

Build with Veri.

Veri is built for developers who want to add high-assurance identity verification without asking users to hand over their data.

INTEGRATION GUIDE

1. WHAT IS VERI

Your users have to prove who they are, and today that usually means handing you their email address — which you then have to store, protect, and justify holding. Veri removes that trade. A user proves they control a verified email address at a real domain, and you find out which domain without ever learning the address. Fewer personal details on your side, less to leak, and the same confidence that a real person with a real, verified mailbox is signing in.

After a successful sign-in your app receives a small set of claims. The important ones are `sub`, a stable identifier for that person on your site only; `email_domain`, the real verified domain such as `gmail.com`, which is what you show the user; and `veri_credential_type`, telling you how strong the proof was. You never receive, and cannot request, the user's actual email address — or their wallet address.

"Sign in with Veri" is a standard OpenID Connect provider (authorization code flow, RS256-signed ID tokens, pairwise subjects). Use whatever OIDC

library your stack already has, or plug Veri into an off-the-shelf app that supports a generic OIDC provider.

2. HOW IT WORKS

1. Your app redirects the user to Veri's authorization endpoint.
2. Veri opens the user's vault, which names your app and shows exactly what you will learn. If the user holds more than one credential, they choose which one to share.
3. On approval, Veri redirects back to your registered redirect URI with a one-time authorization code and your `state`.
4. Your server exchanges the code for tokens, authenticating with your client secret.
5. Your app verifies the ID token and reads the credential. Veri mints it from the user's credential as recorded on-chain at that moment.

3. BEFORE YOU START

- A `client_id` and `client_secret`, issued instantly — no waiting and no emailing us:

1. Click **Get client credentials** on getveri.co/developers.
2. Enter your email, your app's name and its redirect URIs.
3. Type in the 6-digit code we email you.
4. Copy your client ID and secret straight into your `.env` file. The secret is shown once.

One set of credentials per email every 24 hours.

- Your redirect URI(s), registered with Veri. Matching is exact string equality — there are no wildcards and no localhost exception, so register every port you develop on — `http://localhost:3000/callback` must be registered as its own entry, and `https://example.com/callback` and



`https://example.com/callback/` are different URIs. Use HTTPS in production; the authorization code travels on this URL.

- PKCE with `s256`. Clients registered today are required to send a `code_challenge`; a request without one is refused.
- An OIDC library for your stack — see Section 4.

4. QUICK START

All four use the same environment variables. `SESSION_SECRET` is any long random string; the Express and FastAPI examples sign their session cookie with it.

`.env`

```
VERI_ISSUER=https://id.veribeta.io
VERI_CLIENT_ID=your_client_id
VERI_CLIENT_SECRET=your_client_secret
VERI_REDIRECT_URI=http://localhost:3000/callback
SESSION_SECRET=a_long_random_string
```

TYPESCRIPT / NODE

Terminal

```
npm install openid-client
```

`openid-client` is the OpenID-Foundation-certified Node implementation.

`server.ts`

```
import { createServer } from "node:http";
import * as client from "openid-client";

// STEP 1 — Discover Veri. Fetches /.well-known/openid-
// configuration and checks
// the document's issuer matches the host it came from. Do this
```

```

ONCE at boot.
const config = await client.discovery(
  new URL(process.env.VERI_ISSUER!),
  process.env.VERI_CLIENT_ID!,
  process.env.VERI_CLIENT_SECRET!,
  undefined,
  // Also verify the id_token's signature. OFF BY DEFAULT: the
  spec lets a client
  // rely on TLS instead. One line, and worth it.
  { execute: [client.enableNonRepudiationChecks] },
);

// STEP 2 – Build the authorization URL with PKCE, state and
// nonce.
const codeVerifier = client.randomPKCECodeVerifier();
const codeChallenge = await
  client.calculatePKCECodeChallenge(codeVerifier);
const state = client.randomState();
const nonce = client.randomNonce();

// Store all three server-side (session or httpOnly cookie). Never
// in the URL.
const pending = new Map([[state, { nonce, codeVerifier }]]);

const authUrl = client.buildAuthorizationUrl(config, {
  redirect_uri: process.env.VERI_REDIRECT_URI!,
  // Request what you read: openid is required; email_domain is
  // Veri's own scope.
  scope: "openid email email_domain",
  code_challenge: codeChallenge,
  code_challenge_method: "S256",
  state,
  nonce,
});
console.log("Sign in:", authUrl.toString());

// STEP 3 – Handle the callback.
createServer(async (req, res) => {
  const url = new URL(req.url!, "http://localhost:3000");
  if (url.pathname !== "/callback") return
  res.writeHead(404).end();

  const session = pending.get(url.searchParams.get("state") ??
  "");
  if (!session) return res.writeHead(400).end("Unknown state");

  // STEP 4 – Exchange the code AND verify the token in one
  // certified call.
  // Checks the signature, iss, aud, exp, plus the nonce and PKCE
  // bindings.

```

```

const tokens = await client.authorizationCodeGrant(config, url,
{
  expectedState: url.searchParams.get("state")!,
  expectedNonce: session.nonce,
  pkceCodeVerifier: session.codeVerifier,
});

// STEP 5 – Read the credential. See Section 6 for what each
claim means.
const claims = tokens.claims()!;
console.log("email_domain:", claims.email_domain);
console.log("credential type:", claims.veri_credential_type);
console.log("user key:", { issuer:
config.serverMetadata().issuer, sub: claims.sub });

res.end(`Verified ${claims.email_domain} address`);
}).listen(3000);

```

Terminal – run it

```

npm pkg set type=module          # the file uses import and top-
level await
node --env-file=.env server.ts    # Node 22.18+ runs .ts directly;
older Node: npx tsx server.ts

```

PYTHON (FASTAPI)

Terminal

```

pip install authlib httpx fastapi uvicorn itsdangerous

```

main.py

```

import os
from authlib.integrations.starlette_client import OAuth
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse, JSONResponse
from starlette.middleware.sessions import SessionMiddleware

app = FastAPI()
# Carries state, nonce and the PKCE verifier across the redirect.
app.add_middleware(SessionMiddleware,
secret_key=os.environ["SESSION_SECRET"])

```

```

# STEP 1 – Discover Veri. Authlib reads the discovery document,
# caches the JWKS,
# and validates the ID token on exchange.
oauth = OAuth()
oauth.register(
    name="veri",
    server_metadata_url=f"{os.environ['VERI_ISSUER']}/.well-known/openid-configuration",
    client_id=os.environ["VERI_CLIENT_ID"],
    client_secret=os.environ["VERI_CLIENT_SECRET"],
    client_kwargs={
        "scope": "openid email email_domain",
        # PKCE. Authlib sends NONE unless you say this.
        "code_challenge_method": "S256",
    },
)

@app.get("/login")
async def login(request: Request):
    # STEP 2 – Authlib mints state, nonce and the verifier,
    # stores them in the
    # session cookie, and redirects.
    return await oauth.veri.authorize_redirect(request,
os.environ["VERI_REDIRECT_URI"])

@app.get("/callback")
async def callback(request: Request):
    try:
        # STEPS 3+4 – Exchange the code AND validate the
        # token: signature, iss,
        # aud, exp, and the nonce stored at /login.
        token = await
oauth.veri.authorize_access_token(request)
    except Exception as exc:
        return JSONResponse({"error": getattr(exc, "error",
"verification_failed")}, 400)

    # STEP 5 – Read the credential.
    claims = token["userinfo"]
    user_key = {"issuer": os.environ["VERI_ISSUER"], "sub":
claims["sub"]}
    request.session["user"] = user_key

    return HTMLResponse(f"""
        <p>Verified <strong>{claims.get('email_domain')}
</strong> address</p>
        <p>Proof: {claims.get('veri_credential_type')}</p>
    """)

```

Terminal

```
npm install next-auth@beta
```

NextAuth v5 is still 5.0.0-beta.x; v4 is the current stable. The provider object below is the same shape on v4 – only the file layout differs.

Register <https://your-domain.com/api/auth/callback/veri> – Auth.js derives that path from the provider id, so it must match exactly.

auth.ts

```
import NextAuth from "next-auth";

export const { handlers, auth, signIn, signOut } = NextAuth({
  providers: [
    {
      id: "veri", // fixes the callback path:
      /api/auth/callback/veri
      name: "Veri",
      type: "oidc",
      issuer: process.env.VERI_ISSUER, // Auth.js
      discovers + pins iss/aud/exp
      clientId: process.env.VERI_CLIENT_ID,
      clientSecret: process.env.VERI_CLIENT_SECRET,
      authorization: { params: { scope: "openid email
      email_domain" } },
      checks: ["pkce", "state", "nonce"], // do not
      trim this
      profile(claims) {
        return {
          id: claims.sub as string,
          // A pseudonym, not a real inbox – see Section 6.
          email: claims.email as string,
          name: (claims.email_domain as string) ?? "Veri
          user",
          emailDomain: claims.email_domain as string |
          undefined,
          credentialType: claims.veri_credential_type as
          string | undefined,
        };
      },
    },
  ],
});
```

```

callbacks: {
  // profile() runs only on first sign-in – carry what you need
  into the JWT.
  async jwt({ token, user }) {
    if (user) {
      token.emailDomain = (user as any).emailDomain;
      token.credentialType = (user as any).credentialType;
    }
    return token;
  },
  async session({ session, token }) {
    (session as any).veri = {
      issuer: process.env.VERI_ISSUER, // store the issuer
      WITH the sub
      sub: token.sub,
      emailDomain: token.emailDomain,
      credentialType: token.credentialType,
    };
    return session;
  },
},
});

```

components/VeriSignInButton.tsx

```

import { signIn } from "@auth";

export function VeriSignInButton() {
  return (
    <form action={async () => { "use server"; await
    signIn("veri"); }}>
      <button type="submit">Sign in with Veri</button>
    </form>
  );
}

```

app/dashboard/page.tsx – reading the credential in a Server Component

```

import { auth } from "@auth";

export default async function Dashboard() {
  const session = await auth();
  if (!session) return <p>Not signed in.</p>;
  const veri = (session as any).veri;
  return <p>Verified <strong>{veri.emailDomain}</strong>

```

```
address</p>;  
}
```

EXPRESS

Terminal

```
npm install express express-session openid-client
```

Same library as Tab 1, wired as Express middleware. Plain JavaScript; add SESSION_SECRET (any long random string) to your .env.

server.js

```
import express from "express";  
import session from "express-session";  
import * as client from "openid-client";  
  
const app = express();  
// Behind a proxy (Vercel, Render, Railway, Heroku, nginx) the  
// request reaches you as  
// http. Without this, the callback URL rebuilt below says  
// http://, no longer matches  
// your registered https:// redirect URI, and the code exchange  
// fails invalid_grant.  
app.set("trust proxy", 1);  
app.use(session({  
  secret: process.env.SESSION_SECRET,  
  resave: false,  
  saveUninitialized: false,  
  cookie: {  
    httpOnly: true,  
    secure: process.env.NODE_ENV === "production",  
    // Lax, not Strict: the browser returns here via a redirect  
    FROM Veri, and  
    // Strict withholds the cookie on that navigation.  
    sameSite: "lax",  
  },  
}));  
  
// STEP 1 – once, at boot.  
const config = await client.discovery(  
  new URL(process.env.VERI_ISSUER), process.env.VERI_CLIENT_ID,  
  process.env.VERI_CLIENT_SECRET, undefined,  
  { execute: [client.enableNonRepudiationChecks] },  
);
```

```
// STEP 2 – start the flow.
app.get("/auth", async (req, res) => {
  const codeVerifier = client.randomPKCECodeVerifier();
  const codeChallenge = await
client.calculatePKCECodeChallenge(codeVerifier);
  const state = client.randomState();
  const nonce = client.randomNonce();
  req.session.veriFlow = { state, nonce, codeVerifier }; //
server-side only

  res.redirect(client.buildAuthorizationUrl(config, {
    redirect_uri: process.env.VERI_REDIRECT_URI,
    scope: "openid email email_domain",
    code_challenge: codeChallenge, code_challenge_method:
"S256",
    state, nonce,
  })).toString());
});

// STEPS 3-5 – finish it.
app.get("/callback", async (req, res) => {
  const flow = req.session.veriFlow;
  if (!flow) return res.status(400).send("Sign-in expired. Please
try again.");
  delete req.session.veriFlow; //
single use

  const currentUrl = new URL(req.originalUrl,
`${req.protocol}://${req.get("host")}`);
  try {
    const tokens = await client.authorizationCodeGrant(config,
currentUrl, {
      expectedState: flow.state,
      expectedNonce: flow.nonce,
      pkceCodeVerifier: flow.codeVerifier,
    });
    const claims = tokens.claims();
    req.session.user = {
      issuer: process.env.VERI_ISSUER, sub: claims.sub, //
composite key
      emailDomain: claims.email_domain,
      credentialType: claims.veri_credential_type,
    };
    res.redirect("/dashboard");
  } catch (e) {
    console.error("[veri]", e.error ?? e.name,
e.error_description ?? e.message);
    res.status(400).send("Sign-in failed.");
  }
}
```

```
});

// Protect a route.
const requireVeri = (req, res, next) => req.session.user ? next()
: res.redirect("/auth");
app.get("/dashboard", requireVeri, (req, res) =>
  res.send(`Verified ${req.session.user.emailDomain} address`));

// Your "Sign in with Veri" button is just a link to /auth.
app.get("/", (req, res) => res.send(req.session.user
  ? `Signed in. <a href="/logout">Sign out</a>`
  : `<a href="/auth">Sign in with Veri</a>`));

// Signing out ends YOUR session. Veri has no logout endpoint
(Section 9).
app.get("/logout", (req, res) => req.session.destroy(() =>
  res.redirect("/")));

app.listen(3000);
```

Terminal – run it

```
npm pkg set type=module          # the file uses import and top-
level await
node --env-file=.env server.js    # loads .env; or use the dotenv
package
```

5. ENDPOINTS & TOKENS

Everything below is published in the discovery document, which every OIDC library reads for you. Point your library at the issuer and let it discover the rest.

Discovery

```
https://id.veribeta.io/.well-known/openid-configuration
```



ENDPOINT	URL
Authorization	<code>https://id.veribeta.io/api/oidc/authorize</code>
Token	<code>https://id.veribeta.io/api/oidc/token</code>
UserInfo	<code>https://id.veribeta.io/api/oidc/userinfo</code>
JWKS	<code>https://id.veribeta.io/.well-known/jwks.json</code>

SETTING	VALUE
Flow	Authorization code only — <code>response_type=code</code> , <code>grant_type=authorization_code</code> . No implicit or hybrid flow, and no refresh tokens.
Scopes	<code>openid</code> is required. Also supported: <code>email</code> , <code>email_domain</code> , <code>profile</code> . Request <code>openid email email_domain</code> .
PKCE	<code>S256</code> only. <code>plain</code> is rejected.
Client auth	At the token endpoint: <code>client_secret_basic</code> (HTTP Basic, the OIDC default) or <code>client_secret_post</code> (in the body). Both work.
ID token	JWT signed with <code>RS256</code> ; verify it against the JWKS. Valid for 1 hour (<code>exp</code> = <code>iat</code> + 3600).
Access token	Opaque, valid for 1 hour, usable only at the UserInfo endpoint. Returned with <code>token_type: Bearer</code> and <code>expires_in: 3600</code> .
Authorization code	Single use, expires after 10 minutes, and bound to the exact <code>redirect_uri</code> and PKCE verifier of its request.
Subject type	<code>pairwise</code> — see the warning in Section 6.

The UserInfo endpoint returns the same claims as the ID token (except the token-only `iss` , `aud` , `iat` , `exp` and `nonce`) for clients that read profile data

there instead. It accepts the access token as `Authorization: Bearer` on `GET` or `POST`.

6. READING THE CREDENTIAL

Once your library has verified the token you will see the standard OIDC claims — `iss`, `aud`, `sub`, `iat`, `exp`, `nonce` — plus a few that are specific to Veri and that no OIDC documentation covers. Here is what each one means and what to do with it.



CLAIM	WHAT IT IS	EXAMPLE	WHAT TO DO WITH IT
sub	A pairwise identifier, unique to this user on your site	9f2c81a4e0... (64 hex characters)	Use as your user ID. Store the issuer alongside it. A <code>sub</code> without its issuer cannot be matched if Veri ever migrates issuers.
email	A pseudonymous address at Veri's domain	4f3a9c2e1b7d5a80@id.veribeta.io	Never display it, never email it. It is stable per user per site but is not a real inbox. It exists for platforms that refuse to create an account without an email claim.
email_verified	Always true	true	Veri does not mint a token for a user without a verified credential, so you can trust it.
email_domain	The real verified domain	gmail.com	Show this to the user. "Verified gmail.com address" is the right display string.
name, preferred_username	A pseudonymous handle derived from the email claim	veri-4f3a9c2e	For apps that require a display name or username to create a profile. It reveals nothing about the user.
veri_credential_type	How strong the proof was	email_address_v2	<code>email_address_v2</code> proves control of a specific mailbox; <code>email_domain_v1</code> proves only that they hold an address at the domain. Most apps just check <code>email_verified</code> .
veri_credentials	The credentials disclosed in this sign-in, as objects	[{"type": "email_address_v2", "domain": "gmail.com"}]	Sorted strongest first. If the user chose a credential to share, this holds only that one, and <code>email_domain</code> and <code>veri_credential_type</code> describe it too.

**WARNING**

The sub is different on every site

The same Veri user gets a different `sub` on your site than on any other. That is by design — it means nobody can correlate your users against another service. Store the issuer alongside the `sub` and use the pair as your account key. A future issuer migration changes every `sub`, and if you stored the `sub` alone you will silently stop recognising returning users.

```
const userKey = { issuer: "https://id.veribeta.io", sub:
claims.sub }; // right
const userKey = claims.sub; // wrong
```

Two things worth knowing. `veri_credentials` holds objects, not strings — each has a `type` and a `domain`. And a user's `sub` and pseudonymous email are derived from their vault, not from the credential, so whichever credential they choose to share, they land in the same account on your site.

PREVIEW

Device possession & token holdings

On the staging issuer (`https://id-staging.veribeta.io`), a sign-in can also carry two claims that production does not issue yet. Both are absent on an ordinary sign-in, so treat them as optional.

- `veri_mesh` — what was proven during this sign-in: `credentials` (for example `device-possession_v1`, or `token_holder_v1` with an `asset_id` and `threshold`), `per-check outcomes` (`asserted`, `not_held`, `rejected`, `unavailable`), plus `complete`, `degraded` and `completed_at`. If `degraded` is true, a check could not run — never read that as "the user lacks it".

- `amr` — `["swk"]` (RFC 8176) only when the user proved possession of their vault key from their device. Gate on this if you require the stronger proof.

Preview claims

```
"veri_mesh": {
  "credentials": [{ "type": "device_possession_v1" }],
  "outcomes": [{ "verifier": "device_possession", "status":
"asserted" }],
  "complete": true,
  "degraded": false,
  "completed_at": 1791000000
},
"amr": ["swk"]
```

Verified Token Holder credentials reach a relying party only through `veri_mesh`; they never appear in `veri_credentials`.

7. SECURITY CHECKLIST

1. **PKCE** — always use `S256`. Veri rejects `plain`, and new clients must send a challenge.
2. **Nonce** — generate a random nonce, store it in the session, verify the token's nonce matches.
3. **State** — generate a random state, store it in the session, verify it in the callback to prevent CSRF.
4. **Signature** — verify the ID token's RS256 signature against the JWKS. Some libraries skip this by default when the token arrives over TLS; turn it on.
5. **Issuer** — pin `iss` to `https://id.veribeta.io` and reject anything else.



6. **Audience** — pin `aud` to your `client_id` and reject tokens meant for others.
7. **Expiry** — check `exp`. Most libraries do this for you; confirm yours does.
8. **Secret** — keep the `client_secret` and the token exchange on your server, never in a browser or app bundle.
9. **Sub storage** — store the issuer alongside the `sub`. See Section 6 for why.
10. **HTTPS** — use HTTPS for your redirect URI in production, never plain HTTP.

8. ERROR HANDLING

Errors reach you in three places, and only the first two come from Veri.

| From the token endpoint

JSON bodies of the form `{ "error": "...", "error_description": "..." }`.



ERROR	WHAT IT MEANS	WHAT TO DO
invalid_client	HTTP 401. Wrong <code>client_id</code> or <code>client_secret</code> , or the client is inactive.	Check your environment variables.
invalid_grant	The code is unknown, already used or expired (10 minutes); the <code>redirect_uri</code> differs from the original request by even a byte; or PKCE verification failed.	Restart the sign-in flow. If it repeats, compare the redirect URI and code verifier you send.
credential_required	The user holds no verified email credential.	Ask them to verify an email in their Veri vault, then sign in again.
invalid_request	A required parameter is missing or malformed.	Fix the request. Your library normally prevents this.
unsupported_grant_type	Anything other than <code>authorization_code</code> .	Use the authorization code flow.
rate_limit_exceeded	HTTP 429. More than 30 token requests a minute for your client.	Back off and retry.

On your callback URL

Veri redirects back only to a registered URI. Handle `error=access_denied` on your callback (it arrives with your `state` and an `error_description`): that is the standard OIDC response to a declined sign-in, and it is how Veri reports a user who declines. Until that rollout completes, a user who declines is not sent back to your app.

Shown to the user instead

If the authorization request itself is invalid — an unknown or inactive `client_id`, an unregistered `redirect_uri`, a missing PKCE challenge, or a `scope` without `openid` — Veri shows the user an error page and does **not** redirect to you, because the redirect URI cannot be trusted. Nothing arrives at your callback. Test your configuration end to end before you ship.

Everything else — an expired ID token, a nonce or state mismatch, a network failure reaching Veri — is raised by your OIDC library, not by Veri. Catch the library's error types rather than matching on error strings, and treat a nonce or state mismatch as a security signal, not a retry.

9. KNOWN LIMITATIONS

Browser-only apps.

Veri requires a confidential client, which means a `client_secret`, and a secret in a web page is a secret anyone can read and use to impersonate your app. A purely static site therefore cannot integrate directly. Add one small serverless function to hold the secret and do the token exchange — the IETF's OAuth 2.0 for Browser-Based Applications recommends this pattern over public clients in the browser anyway.

Python's standard library.

Cloudflare returns 403 to any client whose User-Agent begins with `Python-urllib`, on every Veri OIDC endpoint. This breaks PyJWT's `PyJWKClient`, which fetches through `urllib`, and the failure is a bare `HTTP Error 403: Forbidden` with nothing pointing at the cause. Authlib uses `httpx` and `mozilla-django-oidc` uses `requests`, so both are unaffected — use one of those rather than the stdlib HTTP client.

Sessions are yours.

There are no refresh tokens and no logout endpoint. The ID token proves the sign-in; after that, keep the user signed in with your own session, and sign them out by ending it — the Express example in Section 4 shows a three-line `/logout`.



Mobile apps.

React Native and other mobile frameworks need a different pattern: they cannot hold a client secret either, and they need a custom URI scheme or app link as the redirect. Contact us before you start rather than adapting the web flow.

10. LIVE DEMO

Try a live demo

OPEN THE LIVE DEMO 

See the full Sign in with Veri flow in under 30 seconds. The demo site learns only your verified email domain — never your address.

11. GETTING HELP

- **Questions or problems:** email team@getveri.co — let's solve problems now, not later.
- **Register a client:** use [Get client credentials](#) in Section 3. To change a redirect URI or replace a lost secret, email us your `client_id`.
- This page is the current version of the integration guide. If you were sent a copy, check it against this page.

