# Introducing Custom OIDC

Charlotte Brandhorst-Satzkorn and Tom D'Netto on March 27, 2023

At Tailscale, we don’t want your users (or us) managing a separate list of usernames and passwords, which is why you must use single sign-on with an identity provider to create and manage your network. Until now, that meant you needed to choose from a handful of trusted identity providers including Google, Okta, GitHub, and Azure AD. Custom OIDC, now in beta (and available for everyone), changes all that.

It doesn’t matter if you’re an enterprise customer with complex identity requirements or a privacy-minded power user self-hosting your own authentication solution — you’re now able to use Tailscale with an OpenID Connect (OIDC) compliant identity provider of your choice. Custom OIDC requires a WebFinger endpoint configured on the domain used for authentication, to prove administrative control of the domain, and for identity provider discovery.

New users can set up custom OIDC and sign in at login.tailscale.com/start/oidc, and existing customers can contact our support team to request account migration.

## Custom OIDC, *The Slightly Questionable Way*

Now the serious part of this blog post is out of the way, it’s time for some shenanigans. For the rest of this post, we’ll describe how one *might* build a janky (Totally Professional, not at all chaotic) identity provider (IdP) to test this functionality, and give you a whirlwind tour of OIDC along the way.

### Spec (non)compliance

WebFinger, OAuth 2.0, and OpenID Connect are all chonky specs with many many pages of SHALL, MUST NOTs, and what have yous. This ain’t that vibe — we wanted to see how many of these requirements we could ignore, and still end up with a working IdP.

### WebFinger

WebFinger is a protocol for discovering information about accounts on a domain, and Tailscale’s control plane uses it to determine the OIDC issuer URL for an account during sign-up. Because WebFinger responses must be served over HTTPS, we can be confident that any valid response is authoritative for that domain.

Fortunately for us, we already had the domain `cat-nip.dev` lying around. So with a lil' bit of code…

```
acme := &autocert.Manager{
  Cache:      autocert.DirCache("/var/lib/yolocerts"),
  Prompt:     autocert.AcceptTOS,
  HostPolicy: autocert.HostWhitelist("cat-nip.dev"),
}
S := &http.Server{
  Handler: http.NewServeMux(),
  TLSConfig: &tls.Config{GetCertificate: acme.GetCertificate}
}
s.ListenAndServeTLS("", "")
```

We now have a ✨*HTTPS server*✨.

What's that? What about WebFinger? Oh yea, *soooooo* technically WebFinger is meant to serve specialized information about the account that was queried, but given everything on this domain is going to use the same OIDC issuer we can just yeet a static response:

```
s.Handler.(*http.ServeMux).Handle("/.well-known/webfinger", func(w http.Response
  w.Header().Set("Content-Type", "application/json")
  w.Write([]byte(`{
    "subject" : "acct:felix@cat-nip.dev",
    "links" :
    [
      {
        "rel" : "http://openid.net/specs/connect/1.0/issuer",
        "href" : "https://cat-nip.dev/"
      }
    ]
  }`))
})
```

## OpenID discovery document

Now that, thanks to WebFinger, Tailscale knows where to look when dancing the OIDC tune, we need to implement enough to tango. The control plane learns our moves by reading the discovery document from our `<OIDC issuer URL>/.well-known/openid-configuration`:

```
{
  "issuer": "https://cat-nip.dev/",
  "authorization_endpoint": "https://cat-nip.dev/authorize",
  "token_endpoint": "https://cat-nip.dev/token",
  "jwks_uri": "https://cat-nip.dev/.well-known/jwks.json",
  "userinfo_endpoint": "https://cat-nip.dev/userinfo",
  "grant_types_supported": ["authorization_code", "refresh_token"],
  "response_types_supported": ["code"],
  "subject_types_supported": ["public"],
  "id_token_signing_alg_values_supported": ["RS256"],
  "scopes_supported": ["openid", "email", "groups", "profile"],
  "token_endpoint_auth_methods_supported": ["client_secret_basic","client_secret
  "claims_supported": ["sub", "email", "email_verified", "preferred_username", '
  "code_challenge_methods_supported": ["plain", "S256"]
}
```

That looks complicated but we're just trying to do #hackz. Hrm, let's just wire up a library that should roughly know what it's doing instead:

```
go get github.com/oauth2-proxy/mockoidc
```

*(A testing library? We did warn you that there would be shenanigans.)*

While our testing library wires us up with reasonable defaults, OpenID Connect IdPs need a little configuration to work. Most notably, we need to pick the OAuth client ID and secret, as well as set up a keypair which can be used for signatures.

```
kp, _ := mockoidc.RandomKeypair(2048)
oidc := &mockoidc.MockOIDC{
    ClientID:     "AzureDiamond",
    ClientSecret: "hunter2",
    AccessTTL:    5 * time.Minute,
    RefreshTTL:   5 * time.Hour,
    CodeChallengeMethodsSupported: []string{"plain", "S256"},
    Keypair:                       kp,
    SessionStore:                  mockoidc.NewSessionStore(),
    UserQueue:                     &mockoidc.UserQueue{},
    ErrorQueue:                    &mockoidc.ErrorQueue{},
    Server: &http.Server{Addr: "cat-nip.dev"},
}
```

Our last step to serve the discovery document is wiring the `mockoidc` handler to our server:

```
s.Handler.(*http.ServeMux).Handle("/.well-known/openid-configuration", oidc.Disc
```

Huzzah! 😎

## OAuth, we meet again

The gory details of any OAuth 2.0 IdP are its handlers for `authorize` and `token` endpoints, which complete a login flow for an authorization code, and exchange an authorization code for an access token respectively. Together, these two endpoints implement the bulk of an OIDC identity provider, and combined with the `userinfo` endpoint (which exchanges an access token for information about the user), these endpoints implement everything Tailscale needs from an identity provider.

The OAuth 2 specification is big and complicated, so it'd be great if we could avoid reading all 76 pages of the RFC and rolling this ourselves. Fortunately, `mockoidc` can handle all of this. Love that for us.

```
s.Handler.(*http.ServeMux).Handle("/oidc/authorize", oidc.Authorize)
s.Handler.(*http.ServeMux).Handle("/oidc/token", oidc.Token)
```

```
s.Handler.(*http.ServeMux).Handle("/oidc/userinfo", oidc.Userinfo)
```

You may be asking: where's the actual login flow? Isn't the `authorize` endpoint meant to provide a UI and some logic for checking credentials and actually logging in a user?

That would involve actually building a login flow! We're just testing the OIDC bits here, so let's just be twitchy and do something much simpler.

### Thanks, I hate it

Nothing I say or do will ever make this okay, so I'll just go ahead and post the code:

```
s.Handler.(*http.ServeMux).Handle("/uwu/secret-url/next-login-as", func(w http.R
   oidc.UserQueue.Push(&mockoidc.MockUser{
        Subject:           req.FormValue("user"),
        Email:             req.FormValue("user") + "@cat-nip.dev",
        PreferredUsername: req.FormValue("user"),
        Phone:             "555-987-6543",
        Address:           "123 Main Street",
        Groups:            []string{"engineering", "design"},
        EmailVerified:     true,
    })
})
```

When anyone makes a request to https://cat-nip.dev/uwu/secret-url/next-login-as, they configure that the next login request will proceed using the username specified as the *user* URL parameter. This way we don't need to implement user sessions, handle credentials, or even store state!

This is horribly insecure in dozens of ways: see if you can list them all!

## Will it work?

At this point we have everything in place we need to test our custom OIDC flow – that is, the vaguely minimalist OIDC issuer and the questionably compliant WebFinger server. To test this out, we would need to:

1. Hit the https://cat-nip.dev /uwu/secret-url/next-login-as endpoint to tell our `mockoidc` service the next user that should be logged in.
2. Go to https://login.tailscale.com/start/oidc and sign up for Tailscale.
3. Use the @cat-nip.dev as the email.
4. Tailscale sends a request to https://cat-nip.dev /.well-known/webfinger to get the OIDC issuer associated with @cat-nip.dev.
5. Input the client ID and secret configured on the `mockoidc` server.
6. Tailscale starts the OIDC authentication flow with the `mockoidc` server, redirecting to the authorize endpoint. Since the `mockoidc` server won't ask for a password, it should just log right in.

With any luck, tada! Authentication should succeed and a tailnet will be created. Since the Tailscale's control plane has observed successful authentication, OIDC configuration parameters will be remembered and any user will be able to login with Tailscale, much to the dismay of security experts everywhere (we are sorry).

## Please don't roll your own IdP

If you've gotten this far, it should be very obvious that this is a terrible idea, and as we said, all entirely hypothetical. The whole point of this was to highlight just how bad of an idea this actually was, show you how it works, and encourage you to please, please, use a real IdP… Please.

If you do want to host your own IdP, we advise you to use a solution like Keycloak, Dex, or Ory. Check out the docs on Custom OIDC to set up your not artisanal, handcrafted, or bespoke IdP with Tailscale.

- SSH Keys
- Docker SSH
- DevSecOps
- Multicloud
- NAT Traversal
- MagicDNS
- PAM
- PoLP
- All articles

- Overview
- Pricing
- Downloads
- Documentation
- How It Works
- Compare Tailscale
- Customers
- Integrations
- Changelog
- Use Tailscale Free

- Company
- Newsletter
- Press Kit
- Blog
- Careers
- Contact Sales
- Contact Support
- Community Forum
- Security
- Status
- Twitter
- GitHub
- YouTube

tailscale