MystenLabs

zkLogin: Onboarding the next billion users to web3

Deepak Maram

Jointly with Kostas Chalkias | Yan Ji | Jonas Lindstrøm | Ben Riva | Arnab Roy | Mahdi Sedaghat | Joy Wang

SBC 2024

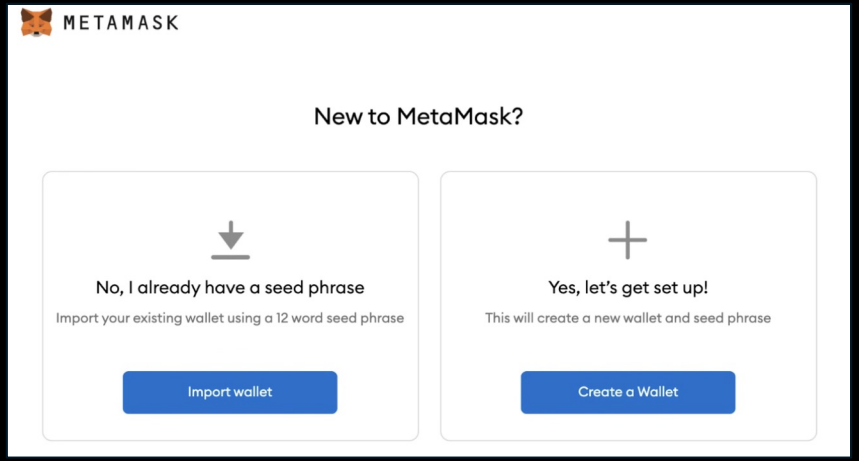
There are fewer than

100 million

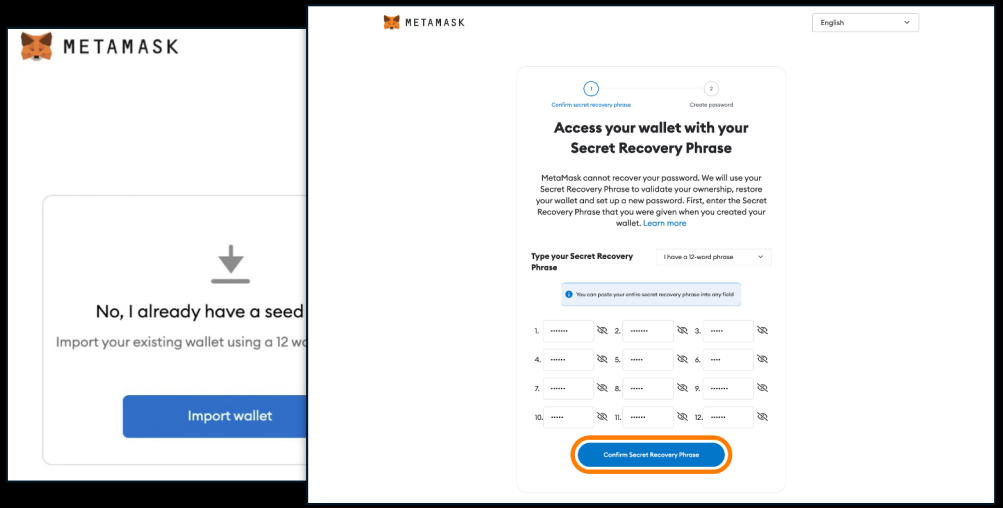
active crypto wallets

and there are
BILLIONS
of web2 accounts

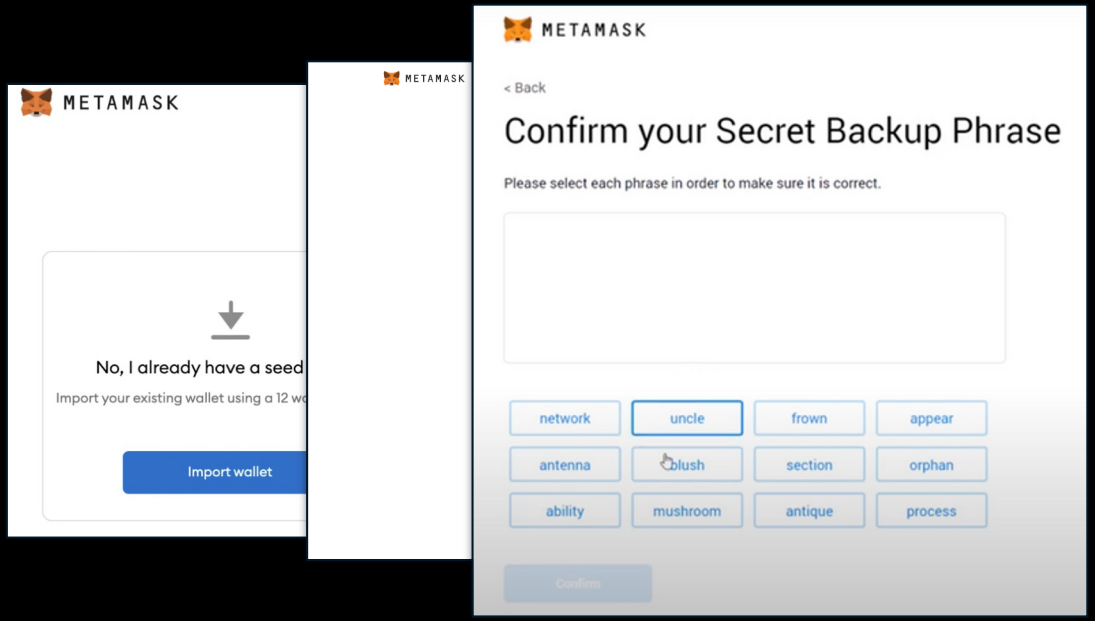
Web3 has an onboarding problem



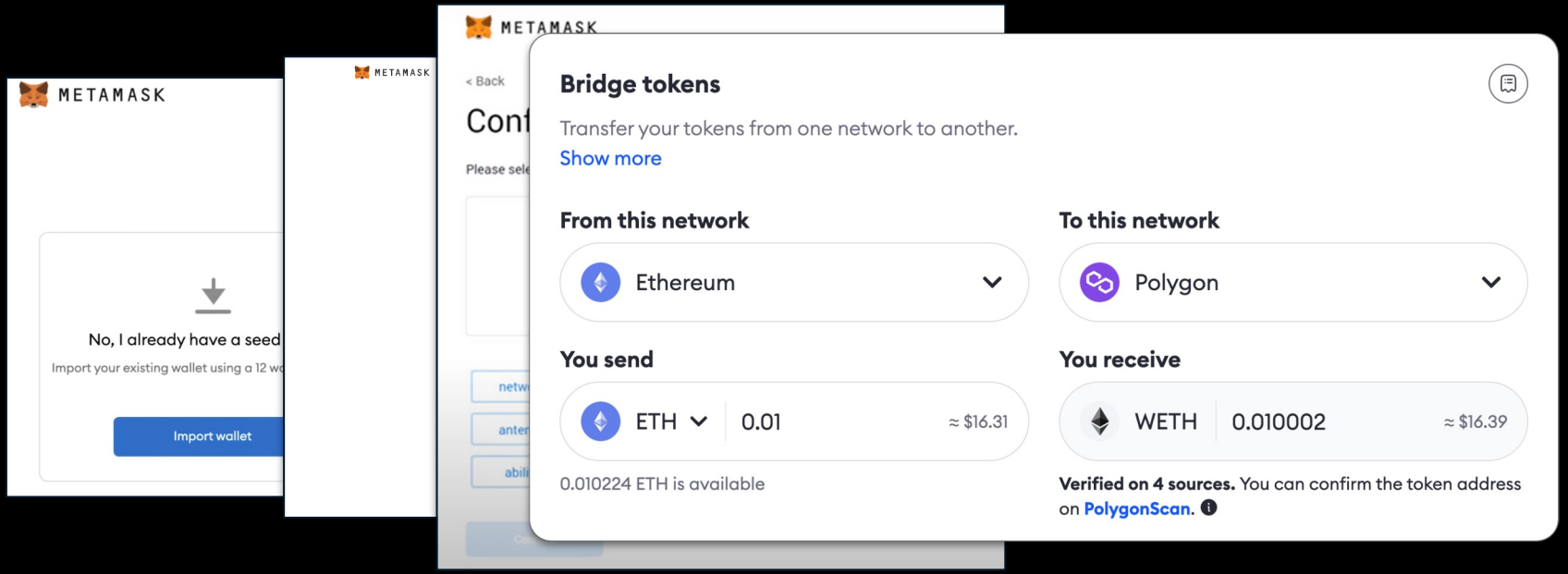
Web3 has an onboarding problem



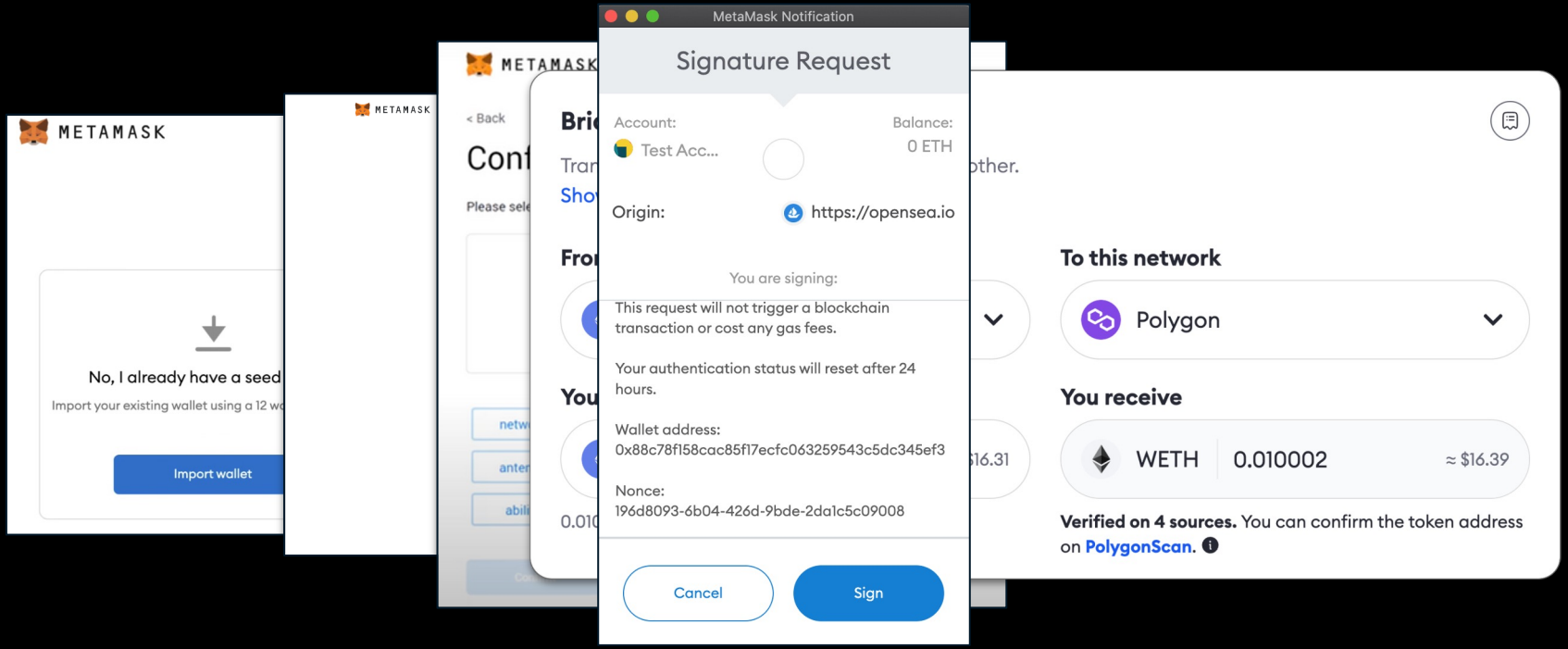
Web3 has an onboarding problem



Web3 has an onboarding problem

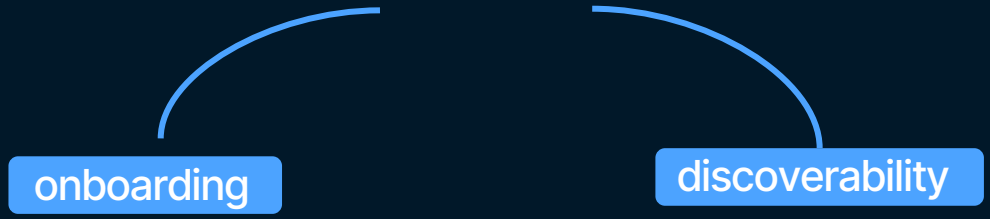


Web3 has an onboarding problem



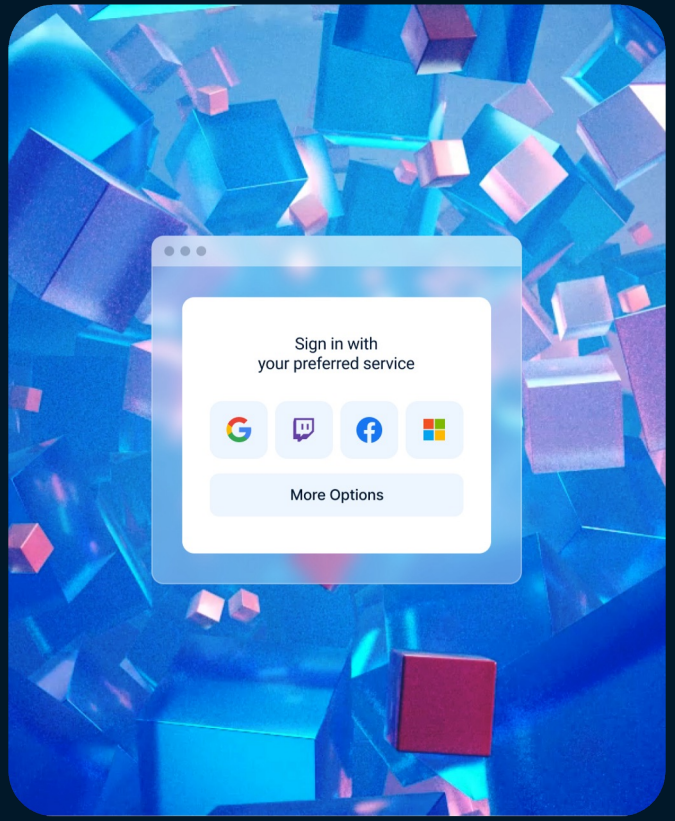
Mnemonics and keys are not going to get us mass adoption.

Complexity is the killer of adoption.
The ultimate killer dApp for blockchain, is accessibility.

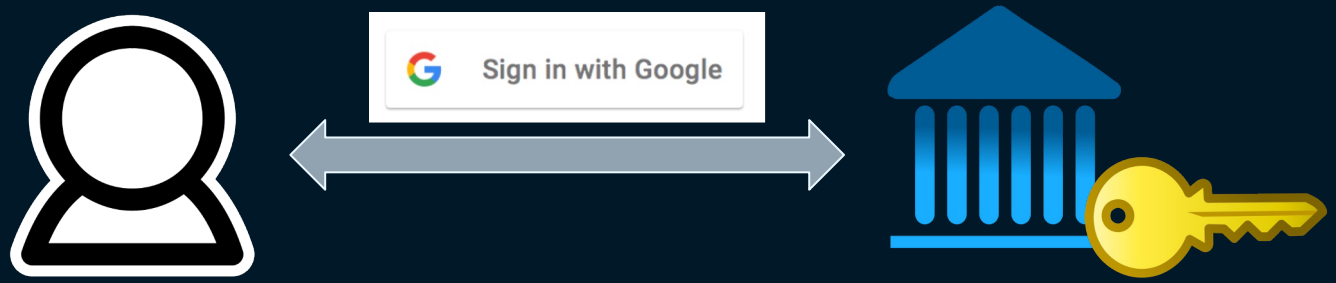


Can we make it as easy as signing in with Google, Facebook and co?

- People don’t want to use separate passwords for each and every app, each and every web2 service
- Extremely likely they already have a Google, Facebook, Amazon account
- Solution: use OAuth to leverage these already existing accounts



Naive solution: OAuth + Custodial wallet

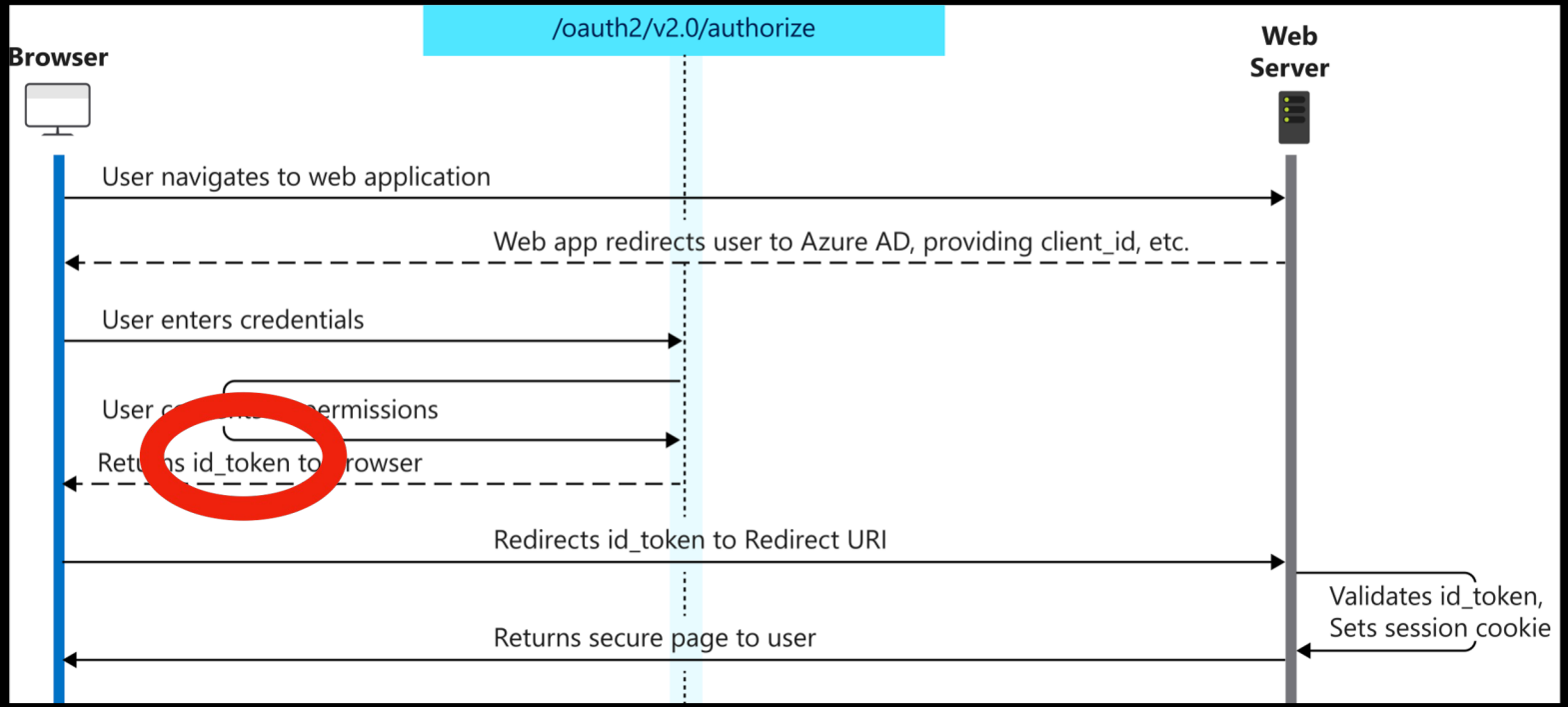


Can we avoid the trusted custodian?

zkLogin:
OAuth + Zero Knowledge Proof

Non-custodial
User-friendly
Privacy-preserving

OpenID Connect (an extension of OAuth 2.0)



JWT: JSON Web Token

Base64-encoded, RSA-signed

JWT as an alternative to a private key?

Encoded PASTE A TOKEN HERE

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ
zdWIiOiJwaGlsaXBwZUBwcmFnbWF0aWN3ZWJzZWN
1cml0eS5jb20iLCJyb2xlIjoiYWRtaW4iLCJpc3M
iOiJwcmFnbWF0aWN3ZWJzZWN1cml0eS5jb20ifQ.
jW4cq__pkcq-r6H1Ebiq8toW-
4Igstk1ibRgxECUhdExvZTzhvXqfrPewgtRHEApB
WXpUqGqRY6LSj2Gklxt3O6kxUaky-
VTl8jbL0OV5HEQVOnL3VVgPv65ddGRYaCOuyzYcf
6M1fA4PeFme9lL2ZTNtjiE0OjUR3LH1Dptm_u9_
aQRtJ_IU8xiywctV1JLeQcMJFDXCS2N5oU0Gkatu
oJNbjMdSTg3BsU5yUsGLyuPnJTeUWJajin5e0NuB
A1Bc6oLee6KtPAM8-
1ufhHr1fpT78iGyrSQLpiVd2naPA0CvUyZ6W_4ar
nmZDKRF9N9zOR_Jxyfv5xFMi4G67EhA
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "RS256"
}
```

PAYLOAD: DATA

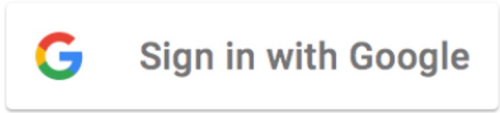
```
  "sub": "philippe@pragmaticwebsecurity.com",
  "role": "admin",
  "iss": "pragmaticwebsecurity.com"
}
```

VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  Lg8ulqDgdXLFwS/lHXV/OKcBOXBrIp
  B4DWQ0c16zLZU7NTe657rWqKlwIDAQ
  AB
  -----END RSA PUBLIC KEY-----
```

A Google-issued JWT (decoded)


```
{
  "alg": "RS256",
  "kid": "96971808796829a972e79a9d1a9fff11cd61b1e3",
  "typ": "JWT"
}
```


```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521            ",
  "nonce": "166379188139080602618705289039940387216697996138036016166781555121812732894 77",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```


you can ask for email
and other personal info

Challenge 1: How to authorize a tx with a JWT?


```
{
  "alg": "RS256",
  "kid": "96971808796829a972e79a9d1a9fff11cd61b1e3",
  "typ": "JWT"
}
```

```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521            ",
  "nonce": "166379188139080602618705289039940387216697996138036016166781555121812732894 77",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```

Inject a fresh pub key into JWT!


```
{
  "alg": "RS256",
  "kid": "96971808796829a972e79a9d1a9fff11cd61b1e3",
  "typ": "JWT"
}
```



```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.ap                    ogleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.a      s.googleusercontent.com",
  "sub": "1104634521                ",
  "nonce": "166379188139080602618705289039940387216697996138036016166781555121812732894 77",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```


replace *nonce* with user provided data:

ephemeral pub key + expiration

We have a DIGITAL CERT over our fresh key + expiration

Challenge 2: How to identify the user without linking identities?


```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521           ",
  "nonce": "166379188139080602618705289039940387216697996138036016166781555121812732894 77",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```


aud = walletID
sub = userID

we could ask for email too

ADDRESS
???

Add a persistent randomizer: salt


```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521            ",
  "nonce": "166379188139080602618705289039940387216697996138036016166781555121812732894 77",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```


aud = walletID
sub = userID

we could ask for email too

ADDRESS

hash(providerID + walletID + userID + salt)

Salt: A persistent per-user secret for unlinkability

Who maintains the salt?

- Client-side on-device management
  - Edge cases, e.g., cross-device sync, device loss need handling

- Server-side management by a “salt service”
  - Each wallet can maintain their own service / delegate it
  - Privacy models: Store salt either in TEE or MPC or in plaintext
  - Auth policies to the service: Either JWT or 2FA

ADDRESS

hash(providerID + walletID + userID + salt)

Salt: A persistent per-user secret for unlinkability

Challenge 3: How to hide the JWT? SNARKs to the rescue!


```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521            ",
  "nonce": "166379188139080602618705289039940387216697996138036016166781555121812732894 77",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```


aud =
walletID
sub = userID

we could ask for email too

nonce =
eph. pubKey
+ expiration

Goal: Prove you have a valid JWT + you know the salt + you injected the ephemeral key into JWT

- Verify JWT's signature using Google's public key
- Verify the ephemeral public key is injected into the JWT's nonce
- Verify that the address is derived correctly from the JWT's userID, walletID, providerID + user's salt

Yellow => private inputs
Blue => public inputs

zkLogin tricks

sample openID JWT token signed by Google / FB

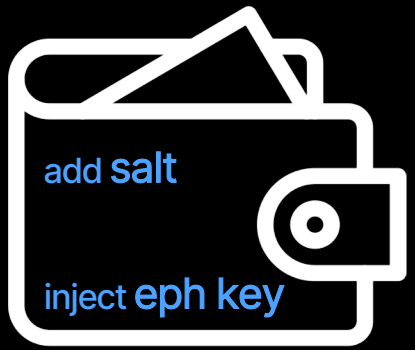
aud = walletID
sub = userID

we could ask for email too

nonce = eph. pubKey + expiration


```
{
  "iss": "https://accounts.google.com",
  "azp": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "aud": "575519204237-msop9ep45u2uo98hapqmngv8d84qdc8k.apps.googleusercontent.com",
  "sub": "1104634521          ",
  "nonce": "166379188139080602618705289039940387216697996138036016166781555121812732894 77",
  "iat": 1682002642,
  "exp": 1682006242,
  "jti": "a8a0728a3ffd5dc81ecfd0ea81d0d33d803eb830"
}
```


+



+ ZK proof =

ADDRESS

~hash(providerID + zkhash(walletID + userID + zkhash(salt)))

&

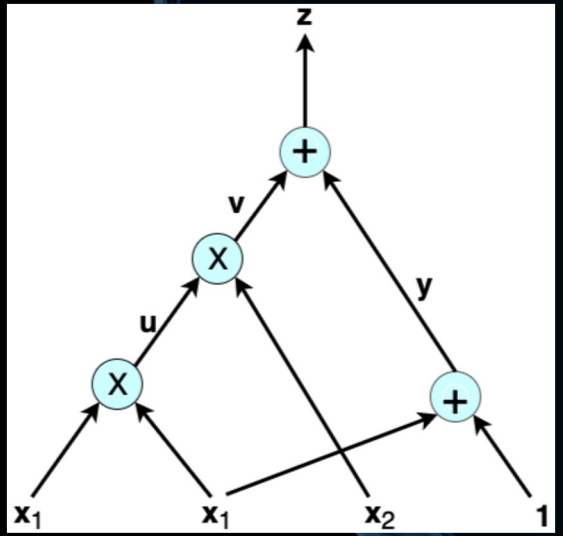
verify ZKproof + verify eph key sig

Challenge 4: Prove + RTT in <3s

- We chose Groth16 due to its small proofs + rich ecosystem + fast prover
- But.. proofs are slow to generate on end-user devices
  - Make ZKP efficient: Hand-optimized circuit that selectively parses relevant parts of the JWT + string slicing tricks + …
  - Delegate proving to an untrusted ZKP service
    - Open problem: How to delegate with privacy?

Circuit details

- Implemented in circom: ~1M R1CS constraints
- Key operations
  - SHA-2 (66%)
  - RSA signature verification (14%) using tricks from [KPS18]
  - JSON parsing, Poseidon hashing, Base64, extra rules (20%)
- Prover based on rapidsnark
  - C++ and Assembly based



zkLogin latency

These numbers correspond only to the first transaction of a session

Operation	zkLogin	Ed25519
Fetch salt from salt service	0.2 s	NA
Fetch ZKP from ZK service	2.78 s	NA
Signature verification	2.04 ms	56.3 μs
E2E transaction confirmation	3.52 s	120.74 ms

Latency for most zkLogin transactions is very similar to traditional ones!


Sui

zkLogin

single-click accounts w/

- Google
- Facebook
- Twitch
- Apple
- Slack
- Microsoft

native authenticator
non-custodial
*discoverable, claimable
invisible wallets
semi-portable, 2FA

zkLogin goodies

Native auth, cheap

Not via smart contracts, same gas cost as regular sig verification.

ID-based wallets

Create email or phone number based accounts.

Can also reveal identity of an existing account (e.g., email) fully or partially (e.g., reveal a suffix like @xyz.edu)

Embedded wallet

Mobile apps or websites can natively integrate zkLogin without the need for a wallet popup!

2FA

Can do a 2-out-of-3 between Google, Facebook and Apple. Salt can also serve as a second factor.

Hard to lose!

Thanks to robust recovery paths of Google, Facebook.

ADDRESS

hash(providerID + zkhash(walletID + userID + zkhash(salt)))

+ ZK proof

MystenLabs

ZK for authentication

How to SNARK sign-in with Google, Apple & FB

Contact

deepak@mystenlabs.com

zkLogin technical paper: https://arxiv.org/abs/2401.11735

Sui zkLogin docs: https://sui.io/zklogin