zkLogin

zkLogin is a Sui primitive that provides the ability for you to send transactions from a Sui address using an OAuth credential, without publicly linking the two.

zkLogin is designed with the following goals in mind:

Are you a builder who wants to integrate zkLogin into your application or wallet? Dive into the zkLogin Integration Guide .

If you want to understand how zkLogin works, including how the zero-knowledge proof is generated, and how Sui verifies a zkLogin transaction, see this section .

If you are curious about the security model and the privacy considerations of zkLogin, visit this page .

More questions? See the FAQ section .

The following table lists the OpenID providers that can support zkLogin or are currently being reviewed to determine whether they can support zkLogin.

In rough sketches, the zkLogin protocol relies on the following:

(Step 0) We use Groth16 for our protocol's zkSNARK instantiation, requiring a singular generation of a structured common reference string (CRS) linked to the circuit. A ceremony is conducted to generate the CRS, which is used to produce the proving key in the ZK Proving Service, the verifying key in Sui Authority. See the Ceremony section for more details.

(Step 1-3) The user begins by logging into an OpenID Provider (OP) to obtain a JWT containing a defined nonce. In particular, the user generates an ephemeral KeyPair (eph_sk, eph_pk) and embed eph_pk , along with expiry times (max_epoch) and randomness (jwt_randomness), into the nonce (see definition). After the user completes the OAuth login flow, an JWT can be found in the redirect URL in the application.

(Step 4-5) The application frontend then sends the JWT to a salt service. The salt service returns the unique user_salt based on iss, aud, sub upon validation of the JWT.

(Step 6-7) The user sends the ZK proving service the JWT, user salt, ephemeral public key, jwt randomness, key claim name (i.e. sub). The proving service generates a Zero-Knowledge Proof that takes these as private inputs and does the following: (a) Checks the nonce is derived correctly as defined (b) Checks that key claim value matches the corresponding field in the JWT, (c) Verifies the RSA signature from OP on the JWT, and (d) the address is consistent with the key claim value and user salt.

(Step 8): The application computes the user address based on iss, aud, sub, aud. This step can be done independently as long as the application has a valid JWT.

(Step 9-10) A transaction is signed using the ephemeral private key to generate an ephemeral signature. Finally, the user submits the transaction along with the ephemeral signature, ZK proof and other inputs to Sui.

(After Step 10) After submitted on chain, Sui Authorities verify the ZK proof against the provider JWKs from storage (agreed upon in consensus) and also the ephemeral signature.

Application frontend: This describes the wallet or frontend application that supports zkLogin. This frontend is responsible for storing the ephemeral private key, directing users to complete the OAuth login flow, creating and signing a zkLogin transaction.

Salt Backup Service: This is a backend service responsible for returning a salt per unique user. See zkLogin Integration Guide for other strategies to maintain salt.

ZK Proving Service: This is a backend service responsible for generating ZK proofs based on JWT, JWT randomness, user salt and max epoch. This proof is submitted on-chain along with the ephemeral signature for a zkLogin transaction.

The address is computed on the following inputs:

The address flag: zk_login_flag = 0x05 for zkLogin address. This serves as a domain separator as a signature scheme defined in crypto agility .

kc_name_F = hashBytesToField(kc_name, maxKCNameLen) : Name of the key claim, e.g., sub . The sequence of bytes is mapped to a field element in BN254 using hashBytesToField (defined below).

kc_value_F = hashBytesToField(kc_value, maxKCValueLen) : The value of the key claim mapped using hashBytesToField .

aud_F = hashBytesToField(aud, maxAudValueLen) : The relying party (RP) identifier. See definition .

iss : The OpenID Provider (OP) identifier. See definition .

user_salt : A value introduced to unlink the OAuth identifier with the on-chain address.

Finally, we derive zk_login_address = Blake2b_256(zk_login_flag, iss_L, iss, addr_seed) where addr_seed = Poseidon_BN254(kc_name_F, kc_value_F, aud_F, Poseidon_BN254(user_salt) .

See below for all relevant OpenID terminology defined in spec and how they are used in zkLogin, along with definitions for protocol details.

OAuth 2.0 authorization server that is capable of authenticating the end-user and providing claims to an RP about the authentication event and the end-user. This is identified in the iss field in JWT payload. Check the table of available OPs for the entities zkLogin currently supports.

OAuth 2.0 client application requiring end-user authentication and claims from an OpenID provider. This is assigned by OP when the developer creates the application. This is identified in the aud field in JWT payload. This refers to any zkLogin enabled wallet or application.

Locally unique and never reassigned identifier within the issuer for the end user, which the RP is intended to consume. Sui uses this as the key claim to derive user address.

A JSON data structure that represents a set of public keys for an OP. A public endpoint (as in https://www.googleapis.com/oauth2/v3/certs) can be queried to retrieve the valid public keys corresponding to kid for the provider. Upon matching with the kid in the header of a JWT, the JWT can be verified against the payload and its corresponding JWK. In Sui, all authorities call the JWK endpoints independently, and update the latest view of JWKs for all supported providers during protocol upgrades. The correctness of JWKs is guaranteed by the quorum (2f+1) of validator stake.

JWT is in the redirect URI to RP after the user completes the OAuth login flow (as in https://redirect.com?id_token=$JWT_TOKEN). The JWT contains a header , payload , and a signature . The signature is an RSA signature verified against jwt_message = header + . + payload and its JWK identified by kid . The payload contains a JSON of many claims that is a name-value pair. See below for the specific claims that are relevant to the zkLogin protocol.

Header

Payload

For a zkLogin transaction, the iat and exp claims (timestamp) are not used. Instead, the nonce specifies expiry times.

The claim used to derive a users' address is termed the "key claim", such as sub or email. Naturally, it's ideal to use claims that are fixed once and never changed again. zkLogin currently supports sub as the key claim because OpenID spec mandates that providers do not change this identifier. In the future, this can be extended to use email, username, and so on.

To preserve privacy of the OAuth artifacts, a zero-knowledge proof of possession of the artifacts is provided. zkLogin employs the Groth16 zkSNARK to instantiate the zero-knowledge proofs, as it is the most efficient general-purpose zkSNARK in terms of proof size and verification efficiency.

However, Groth16 needs a computation-specific common reference string (CRS) to be setup by a trusted party. With zkLogin expected to ensure the safe-keeping of high value transactions and the integrity of critical smart contracts, we cannot base the security of the system on the honesty of a single entity. Hence, to generate the CRS for the zkLogin circuit, it is vital to run a protocol which bases its security on the assumed honesty of a small fraction of a large number of parties.

The Sui zkLogin ceremony is essentially a cryptographic multi-party computation (MPC) performed by a diverse group of participants to generate this CRS. We follow the MPC protocol MMORPG described by Bowe, Gabizon and Miers. The protocol roughly proceeds in 2 phases. The first phase results in a series of powers of a secret quantity tau in the exponent of an elliptic curve element. Since this phase is circuit-agnostic, we adopted the result of the existing community contributed perpetual powers of tau . Our ceremony was the second phase, which is specific to the zkLogin circuit.

The MMORPG protocol is a sequential protocol, which allows an indefinite number of parties to participate in sequence, without the need of any prior synchronization or ordering. Each party needs to download the output of the previous party, generate entropy of its own and then layer it on top of the received result, producing its own contribution, which is then relayed to the next party. The

protocol guarantees security, if at least one of the participants follows the protocol faithfully, generates strong entropy and discards it reliably.

We sent invitations to 100+ people with diverse backgrounds and affiliations: Sui validators, cryptographers, web3 experts, world-renowned academicians, and business leaders. We planned the ceremony to take place on the dates September 12-18, 2023, but allowed participants to join when they wanted with no fixed slots.

Since the MPC is sequential, each contributor needed to wait till the previous contributor finished in order to receive the previous contribution, follow the MPC steps and produce their own contribution. Due to this structure, we provisioned a queue where participants waited, while those who joined before them finished. To authenticate participants, we sent a unique activation code to each of them. The activation code was the secret key of a signing key pair, which had a dual purpose: it allowed the coordination server to associate the participant's email with the contribution, and to verify the contribution with the corresponding public key.

Participants had two options to contribute: through a browser or a docker. The browser option was more user-friendly for contributors to participate as everything happens in the browser. The Docker option required Docker setup but is more transparent —the Dockerfile and contributor source code are open-sourced and the whole process is verifiable. Moreover, the browser option utilizes snarkjs while the Docker option utilizes Kobi's implementation . This provided software variety and contributors could choose to contribute by whichever method they trust. In addition, participants could generate entropy via entering random text or making random cursor movements.

The zkLogin circuit and the ceremony client code were made open source and the links were made available to the participants to review before the ceremony. In addition, we also posted these developer docs and an audit report on the circuit from zkSecurity. We adopted challenge #0081 (resulting from 80 community contributions) from perpetual powers of tau in phase 1, which is circuit agnostic. We applied the output of the Drand random beacon at epoch #3298000 to remove bias. For phase 2, our ceremony had 111 contributions, 82 from browser and 29 from docker. Finally, we applied the output of the Drand random beacon at epoch #3320606 to remove bias from contributions. All intermediate files can be reproduced following instructions here for phase 1 and here for phase 2.

The final CRS along with the transcript of contribution of every participant is available in a public repository. Contributors received both the hash of the previous contribution they were working on and the resulting hash after their contribution, displayed on-screen and sent via email. They can compare these hashes with the transcripts publicly available on the ceremony site. In addition, anyone is able to check that the hashes are computed correctly and each contribution is properly incorporated in the finalized parameters.

Eventually, the final CRS was used to generate the proving key and verifying key. The proving key is used to generate zero knowledge proof for zkLogin, stored with the ZK proving service. The verifying key was deployed as part of the validator software (protocol version 25 in release 1.10.1) that is used to verify the zkLogin transaction on Sui.

The following sections walk through all zkLogin artifacts, their security assumptions, and the consequences of loss or exposure.

The JWT's validity is scoped on the client ID (aud) to prevent phishing attacks. The same origin policy for the proof prevents the JWT obtained for a malicious application from being used for zkLogin. The JWT for the client ID is sent directly to the application frontend through the redirect URL. A leaked JWT for the specific client ID can compromise user privacy, as these tokens frequently hold sensitive information like usernames and emails. Furthermore, if a backend salt server is responsible for user salt management, the JWT could potentially be exploited to retrieve the user's salt, which introduces additional risks.

However, a JWT leak does not mean loss of funds as long as the corresponding ephemeral private key is safe.

The user salt is required to get access to the zkLogin wallet. This value is essential for both ZK proof generation and zkLogin address derivation.

The leak of user salt does not mean loss of funds, but it enables the attacker to associate the user's subject identifier (i.e. sub) with the Sui address. This can be problematic depending on whether pairwise or public subject identifiers are in use. In particular, there is no problem if pairwise IDs are used (e.g., Facebook) as the subject identifier is unique per RP. However, with public reusable IDs (e.g., Google and Twitch), the globally unique sub value can be used to identify users.

The ephemeral private key's lifespan is tied to the maximum epoch specified in nonce for creating a valid ZK proof. Should it be misplaced, a new ephemeral private key can be generated for transaction signing, accompanied by a freshly generated ZK proof using a new nonce. However, if the ephemeral private key is compromised, acquiring the user salt and the valid ZK proof would be necessary to move funds.

Obtaining the proof itself cannot create a valid zkLogin transaction because an ephemeral signature over the transaction is also needed.

By default, there is no linking between the OAuth subject identifier (i.e. sub) and a Sui address. This is the purpose of the user salt.

The JWT is not published on-chain by default. The revealed values include iss , aud and kid so that the public input hash can be computed, any sensitive fields such as sub are used as private inputs when generating the proof.

The ZK proving service and the salt service (if maintained) can link the user identity since the user salt and JWT are known, but the two services are stateless by design.

In the future, the user can opt in to verify their OAuth identity associated with an Sui address on-chain.

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt (see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds

between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value. See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

OpenID providers

The following table lists the OpenID providers that can support zkLogin or are currently being reviewed to determine whether they can support zkLogin.

In rough sketches, the zkLogin protocol relies on the following:

(Step 0) We use Groth16 for our protocol's zkSNARK instantiation, requiring a singular generation of a structured common reference string (CRS) linked to the circuit. A ceremony is conducted to generate the CRS, which is used to produce the proving key in the ZK Proving Service, the verifying key in Sui Authority. See the Ceremony section for more details.

(Step 1-3) The user begins by logging into an OpenID Provider (OP) to obtain a JWT containing a defined nonce. In particular, the user generates an ephemeral KeyPair (eph_sk, eph_pk) and embed eph_pk , along with expiry times (max_epoch) and randomness (jwt_randomness), into the nonce (see definition). After the user completes the OAuth login flow, an JWT can be found in the redirect URL in the application.

(Step 4-5) The application frontend then sends the JWT to a salt service. The salt service returns the unique user_salt based on iss, aud, sub upon validation of the JWT.

(Step 6-7) The user sends the ZK proving service the JWT, user salt, ephemeral public key, jwt randomness, key claim name (i.e. sub). The proving service generates a Zero-Knowledge Proof that takes these as private inputs and does the following: (a) Checks the nonce is derived correctly as defined (b) Checks that key claim value matches the corresponding field in the JWT, (c) Verifies the RSA signature from OP on the JWT, and (d) the address is consistent with the key claim value and user salt.

(Step 8): The application computes the user address based on iss, aud, sub, aud. This step can be done independently as long as the application has a valid JWT.

(Step 9-10) A transaction is signed using the ephemeral private key to generate an ephemeral signature. Finally, the user submits the transaction along with the ephemeral signature, ZK proof and other inputs to Sui.

(After Step 10) After submitted on chain, Sui Authorities verify the ZK proof against the provider JWKs from storage (agreed upon in consensus) and also the ephemeral signature.

Application frontend: This describes the wallet or frontend application that supports zkLogin. This frontend is responsible for storing the ephemeral private key, directing users to complete the OAuth login flow, creating and signing a zkLogin transaction.

Salt Backup Service: This is a backend service responsible for returning a salt per unique user. See zkLogin Integration Guide for other strategies to maintain salt.

ZK Proving Service: This is a backend service responsible for generating ZK proofs based on JWT, JWT randomness, user salt and max epoch. This proof is submitted on-chain along with the ephemeral signature for a zkLogin transaction.

The address is computed on the following inputs:

The address flag: zk_login_flag = 0x05 for zkLogin address. This serves as a domain separator as a signature scheme defined in crypto agility .

kc_name_F = hashBytesToField(kc_name, maxKCNameLen) : Name of the key claim, e.g., sub . The sequence of bytes is mapped to a field element in BN254 using hashBytesToField (defined below).

kc_value_F = hashBytesToField(kc_value, maxKCValueLen) : The value of the key claim mapped using hashBytesToField .

aud_F = hashBytesToField(aud, maxAudValueLen) : The relying party (RP) identifier. See definition .

iss : The OpenID Provider (OP) identifier. See definition .

user_salt : A value introduced to unlink the OAuth identifier with the on-chain address.

Finally, we derive zk_login_address = Blake2b_256(zk_login_flag, iss_L, iss, addr_seed) where addr_seed = Poseidon_BN254(kc_name_F, kc_value_F, aud_F, Poseidon_BN254(user_salt) .

See below for all relevant OpenID terminology defined in spec and how they are used in zkLogin, along with definitions for protocol details.

OAuth 2.0 authorization server that is capable of authenticating the end-user and providing claims to an RP about the authentication event and the end-user. This is identified in the iss field in JWT payload. Check the table of available OPs for the entities zkLogin currently supports.

OAuth 2.0 client application requiring end-user authentication and claims from an OpenID provider. This is assigned by OP when the developer creates the application. This is identified in the aud field in JWT payload. This refers to any zkLogin enabled wallet or application.

Locally unique and never reassigned identifier within the issuer for the end user, which the RP is intended to consume. Sui uses this as the key claim to derive user address.

A JSON data structure that represents a set of public keys for an OP. A public endpoint (as in https://www.googleapis.com/oauth2/v3/certs) can be queried to retrieve the valid public keys corresponding to kid for the provider. Upon matching with the kid in the header of a JWT, the JWT can be verified against the payload and its corresponding JWK. In Sui, all authorities call the JWK endpoints independently, and update the latest view of JWKs for all supported providers during protocol upgrades. The correctness of JWKs is guaranteed by the quorum (2f+1) of validator stake.

JWT is in the redirect URI to RP after the user completes the OAuth login flow (as in https://redirect.com?id_token=$JWT_TOKEN). The JWT contains a header , payload , and a signature . The signature is an RSA signature verified against jwt_message = header + . + payload and its JWK identified by kid . The payload contains a JSON of many claims that is a name-value pair. See below for the specific claims that are relevant to the zkLogin protocol.

Header

Payload

For a zkLogin transaction, the iat and exp claims (timestamp) are not used. Instead, the nonce specifies expiry times.

The claim used to derive a users' address is termed the "key claim", such as sub or email. Naturally, it's ideal to use claims that are fixed once and never changed again. zkLogin currently supports sub as the key claim because OpenID spec mandates that providers do not change this identifier. In the future, this can be extended to use email, username, and so on.

To preserve privacy of the OAuth artifacts, a zero-knowledge proof of possession of the artifacts is provided. zkLogin employs the Groth16 zkSNARK to instantiate the zero-knowledge proofs, as it is the most efficient general-purpose zkSNARK in terms of proof size and verification efficiency.

However, Groth16 needs a computation-specific common reference string (CRS) to be setup by a trusted party. With zkLogin expected to ensure the safe-keeping of high value transactions and the integrity of critical smart contracts, we cannot base the security of the system on the honesty of a single entity. Hence, to generate the CRS for the zkLogin circuit, it is vital to run a protocol which bases its security on the assumed honesty of a small fraction of a large number of parties.

The Sui zkLogin ceremony is essentially a cryptographic multi-party computation (MPC) performed by a diverse group of participants to generate this CRS. We follow the MPC protocol MMORPG described by Bowe, Gabizon and Miers. The protocol roughly proceeds in 2 phases. The first phase results in a series of powers of a secret quantity tau in the exponent of an elliptic curve element. Since this phase is circuit-agnostic, we adopted the result of the existing community contributed perpetual powers of tau . Our ceremony was the second phase, which is specific to the zkLogin circuit.

The MMORPG protocol is a sequential protocol, which allows an indefinite number of parties to participate in sequence, without the need of any prior synchronization or ordering. Each party needs to download the output of the previous party, generate entropy of its own and then layer it on top of the received result, producing its own contribution, which is then relayed to the next party. The protocol guarantees security, if at least one of the participants follows the protocol faithfully, generates strong entropy and discards it reliably.

We sent invitations to 100+ people with diverse backgrounds and affiliations: Sui validators, cryptographers, web3 experts, world-renowned academicians, and business leaders. We planned the ceremony to take place on the dates September 12-18, 2023, but allowed participants to join when they wanted with no fixed slots.

Since the MPC is sequential, each contributor needed to wait till the previous contributor finished in order to receive the previous contribution, follow the MPC steps and produce their own contribution. Due to this structure, we provisioned a queue where participants waited, while those who joined before them finished. To authenticate participants, we sent a unique activation code to each of them. The activation code was the secret key of a signing key pair, which had a dual purpose: it allowed the coordination server to associate the participant's email with the contribution, and to verify the contribution with the corresponding public key.

Participants had two options to contribute: through a browser or a docker. The browser option was more user-friendly for contributors to participate as everything happens in the browser. The Docker option required Docker setup but is more transparent —the Dockerfile and contributor source code are open-sourced and the whole process is verifiable. Moreover, the browser option utilizes snarkjs while the Docker option utilizes Kobi's implementation . This provided software variety and contributors could choose to contribute by whichever method they trust. In addition, participants could generate entropy via entering random text or

making random cursor movements.

The zkLogin circuit and the ceremony client code were made open source and the links were made available to the participants to review before the ceremony. In addition, we also posted these developer docs and an audit report on the circuit from zkSecurity. We adopted challenge #0081 (resulting from 80 community contributions) from perpetual powers of tau in phase 1, which is circuit agnostic. We applied the output of the Drand random beacon at epoch #3298000 to remove bias. For phase 2, our ceremony had 111 contributions, 82 from browser and 29 from docker. Finally, we applied the output of the Drand random beacon at epoch #3320606 to remove bias from contributions. All intermediate files can be reproduced following instructions here for phase 1 and here for phase 2.

The final CRS along with the transcript of contribution of every participant is available in a public repository. Contributors received both the hash of the previous contribution they were working on and the resulting hash after their contribution, displayed on-screen and sent via email. They can compare these hashes with the transcripts publicly available on the ceremony site. In addition, anyone is able to check that the hashes are computed correctly and each contribution is properly incorporated in the finalized parameters.

Eventually, the final CRS was used to generate the proving key and verifying key. The proving key is used to generate zero knowledge proof for zkLogin, stored with the ZK proving service. The verifying key was deployed as part of the validator software (protocol version 25 in release 1.10.1) that is used to verify the zkLogin transaction on Sui.

The following sections walk through all zkLogin artifacts, their security assumptions, and the consequences of loss or exposure.

The JWT's validity is scoped on the client ID (aud) to prevent phishing attacks. The same origin policy for the proof prevents the JWT obtained for a malicious application from being used for zkLogin. The JWT for the client ID is sent directly to the application frontend through the redirect URL. A leaked JWT for the specific client ID can compromise user privacy, as these tokens frequently hold sensitive information like usernames and emails. Furthermore, if a backend salt server is responsible for user salt management, the JWT could potentially be exploited to retrieve the user's salt, which introduces additional risks.

However, a JWT leak does not mean loss of funds as long as the corresponding ephemeral private key is safe.

The user salt is required to get access to the zkLogin wallet. This value is essential for both ZK proof generation and zkLogin address derivation.

The leak of user salt does not mean loss of funds, but it enables the attacker to associate the user's subject identifier (i.e. sub) with the Sui address. This can be problematic depending on whether pairwise or public subject identifiers are in use. In particular, there is no problem if pairwise IDs are used (e.g., Facebook) as the subject identifier is unique per RP. However, with public reusable IDs (e.g., Google and Twitch), the globally unique sub value can be used to identify users.

The ephemeral private key's lifespan is tied to the maximum epoch specified in nonce for creating a valid ZK proof. Should it be misplaced, a new ephemeral private key can be generated for transaction signing, accompanied by a freshly generated ZK proof using a new nonce. However, if the ephemeral private key is compromised, acquiring the user salt and the valid ZK proof would be necessary to move funds.

Obtaining the proof itself cannot create a valid zkLogin transaction because an ephemeral signature over the transaction is also needed.

By default, there is no linking between the OAuth subject identifier (i.e. sub) and a Sui address. This is the purpose of the user salt. The JWT is not published on-chain by default. The revealed values include iss , aud and kid so that the public input hash can be computed, any sensitive fields such as sub are used as private inputs when generating the proof.

The ZK proving service and the salt service (if maintained) can link the user identity since the user salt and JWT are known, but the two services are stateless by design.

In the future, the user can opt in to verify their OAuth identity associated with an Sui address on-chain.

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt (see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a

new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value. See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

How zkLogin works

In rough sketches, the zkLogin protocol relies on the following:

(Step 0) We use Groth16 for our protocol's zkSNARK instantiation, requiring a singular generation of a structured common reference string (CRS) linked to the circuit. A ceremony is conducted to generate the CRS, which is used to produce the proving key in the ZK Proving Service, the verifying key in Sui Authority. See the Ceremony section for more details.

(Step 1-3) The user begins by logging into an OpenID Provider (OP) to obtain a JWT containing a defined nonce. In particular, the user generates an ephemeral KeyPair (eph_sk, eph_pk) and embed eph_pk , along with expiry times (max_epoch) and randomness (jwt_randomness), into the nonce (see definition). After the user completes the OAuth login flow, an JWT can be found in the redirect URL in the application.

(Step 4-5) The application frontend then sends the JWT to a salt service. The salt service returns the unique user_salt based on iss, aud, sub upon validation of the JWT.

(Step 6-7) The user sends the ZK proving service the JWT, user salt, ephemeral public key, jwt randomness, key claim name (i.e. sub). The proving service generates a Zero-Knowledge Proof that takes these as private inputs and does the following: (a) Checks the nonce is derived correctly as defined (b) Checks that key claim value matches the corresponding field in the JWT, (c) Verifies the RSA signature from OP on the JWT, and (d) the address is consistent with the key claim value and user salt.

(Step 8): The application computes the user address based on iss, aud, sub, aud. This step can be done independently as long as the application has a valid JWT.

(Step 9-10) A transaction is signed using the ephemeral private key to generate an ephemeral signature. Finally, the user submits the transaction along with the ephemeral signature, ZK proof and other inputs to Sui.

(After Step 10) After submitted on chain, Sui Authorities verify the ZK proof against the provider JWKs from storage (agreed upon in consensus) and also the ephemeral signature.

Application frontend: This describes the wallet or frontend application that supports zkLogin. This frontend is responsible for storing the ephemeral private key, directing users to complete the OAuth login flow, creating and signing a zkLogin transaction.

Salt Backup Service: This is a backend service responsible for returning a salt per unique user. See zkLogin Integration Guide for other strategies to maintain salt.

ZK Proving Service: This is a backend service responsible for generating ZK proofs based on JWT, JWT randomness, user salt and max epoch. This proof is submitted on-chain along with the ephemeral signature for a zkLogin transaction.

The address is computed on the following inputs:

The address flag: zk_login_flag = 0x05 for zkLogin address. This serves as a domain separator as a signature scheme defined in crypto agility .

kc_name_F = hashBytesToField(kc_name, maxKCNameLen) : Name of the key claim, e.g., sub . The sequence of bytes is mapped to a field element in BN254 using hashBytesToField (defined below).

kc_value_F = hashBytesToField(kc_value, maxKCValueLen) : The value of the key claim mapped using hashBytesToField .

aud_F = hashBytesToField(aud, maxAudValueLen) : The relying party (RP) identifier. See definition .

iss : The OpenID Provider (OP) identifier. See definition .

user_salt : A value introduced to unlink the OAuth identifier with the on-chain address.

Finally, we derive zk_login_address = Blake2b_256(zk_login_flag, iss_L, iss, addr_seed) where addr_seed = Poseidon_BN254(kc_name_F, kc_value_F, aud_F, Poseidon_BN254(user_salt) .

See below for all relevant OpenID terminology defined in spec and how they are used in zkLogin, along with definitions for protocol details.

OAuth 2.0 authorization server that is capable of authenticating the end-user and providing claims to an RP about the authentication event and the end-user. This is identified in the iss field in JWT payload. Check the table of available OPs for the entities zkLogin currently supports.

OAuth 2.0 client application requiring end-user authentication and claims from an OpenID provider. This is assigned by OP when the developer creates the application. This is identified in the aud field in JWT payload. This refers to any zkLogin enabled wallet or application.

Locally unique and never reassigned identifier within the issuer for the end user, which the RP is intended to consume. Sui uses this as the key claim to derive user address.

A JSON data structure that represents a set of public keys for an OP. A public endpoint (as in https://www.googleapis.com/oauth2/v3/certs) can be queried to retrieve the valid public keys corresponding to kid for the provider. Upon matching with the kid in the header of a JWT, the JWT can be verified against the payload and its corresponding JWK. In Sui, all authorities call the JWK endpoints independently, and update the latest view of JWKs for all supported providers during protocol upgrades. The correctness of JWKs is guaranteed by the quorum (2f+1) of validator stake.

JWT is in the redirect URI to RP after the user completes the OAuth login flow (as in https://redirect.com?id_token=$JWT_TOKEN). The JWT contains a header , payload , and a signature . The signature is an RSA signature verified against jwt_message = header + . + payload and its JWK identified by kid . The payload contains a JSON of many claims that is a name-value pair. See below for the specific claims that are relevant to the zkLogin protocol.

Header

Payload

For a zkLogin transaction, the iat and exp claims (timestamp) are not used. Instead, the nonce specifies expiry times.

The claim used to derive a users' address is termed the "key claim", such as sub or email. Naturally, it's ideal to use claims that are fixed once and never changed again. zkLogin currently supports sub as the key claim because OpenID spec mandates that providers do not change this identifier. In the future, this can be extended to use email, username, and so on.

To preserve privacy of the OAuth artifacts, a zero-knowledge proof of possession of the artifacts is provided. zkLogin employs the Groth16 zkSNARK to instantiate the zero-knowledge proofs, as it is the most efficient general-purpose zkSNARK in terms of proof size and verification efficiency.

However, Groth16 needs a computation-specific common reference string (CRS) to be setup by a trusted party. With zkLogin expected to ensure the safe-keeping of high value transactions and the integrity of critical smart contracts, we cannot base the security of the system on the honesty of a single entity. Hence, to generate the CRS for the zkLogin circuit, it is vital to run a protocol which bases its security on the assumed honesty of a small fraction of a large number of parties.

The Sui zkLogin ceremony is essentially a cryptographic multi-party computation (MPC) performed by a diverse group of participants to generate this CRS. We follow the MPC protocol MMORPG described by Bowe, Gabizon and Miers. The protocol roughly proceeds in 2 phases. The first phase results in a series of powers of a secret quantity tau in the exponent of an elliptic curve element. Since this phase is circuit-agnostic, we adopted the result of the existing community contributed perpetual powers of tau . Our ceremony was the second phase, which is specific to the zkLogin circuit.

The MMORPG protocol is a sequential protocol, which allows an indefinite number of parties to participate in sequence, without the need of any prior synchronization or ordering. Each party needs to download the output of the previous party, generate entropy of its own and then layer it on top of the received result, producing its own contribution, which is then relayed to the next party. The protocol guarantees security, if at least one of the participants follows the protocol faithfully, generates strong entropy and discards it reliably.

We sent invitations to 100+ people with diverse backgrounds and affiliations: Sui validators, cryptographers, web3 experts, world-renowned academicians, and business leaders. We planned the ceremony to take place on the dates September 12-18, 2023, but allowed participants to join when they wanted with no fixed slots.

Since the MPC is sequential, each contributor needed to wait till the previous contributor finished in order to receive the previous contribution, follow the MPC steps and produce their own contribution. Due to this structure, we provisioned a queue where participants waited, while those who joined before them finished. To authenticate participants, we sent a unique activation code to each of them. The activation code was the secret key of a signing key pair, which had a dual purpose: it allowed the coordination server to associate the participant's email with the contribution, and to verify the contribution with the corresponding public key.

Participants had two options to contribute: through a browser or a docker. The browser option was more user-friendly for contributors to participate as everything happens in the browser. The Docker option required Docker setup but is more transparent —the Dockerfile and contributor source code are open-sourced and the whole process is verifiable. Moreover, the browser option utilizes snarkjs while the Docker option utilizes Kobi's implementation . This provided software variety and contributors could choose to contribute by whichever method they trust. In addition, participants could generate entropy via entering random text or making random cursor movements.

The zkLogin circuit and the ceremony client code were made open source and the links were made available to the participants to review before the ceremony. In addition, we also posted these developer docs and an audit report on the circuit from zkSecurity. We adopted challenge #0081 (resulting from 80 community contributions) from perpetual powers of tau in phase 1, which is circuit agnostic. We applied the output of the Drand random beacon at epoch #3298000 to remove bias. For phase 2, our ceremony had 111 contributions, 82 from browser and 29 from docker. Finally, we applied the output of the Drand random beacon at epoch #3320606 to remove bias from contributions. All intermediate files can be reproduced following instructions here for phase 1 and here for phase 2.

The final CRS along with the transcript of contribution of every participant is available in a public repository. Contributors received both the hash of the previous contribution they were working on and the resulting hash after their contribution, displayed on-screen and sent via email. They can compare these hashes with the transcripts publicly available on the ceremony site. In addition, anyone is able to check that the hashes are computed correctly and each contribution is properly incorporated in the finalized parameters.

Eventually, the final CRS was used to generate the proving key and verifying key. The proving key is used to generate zero knowledge proof for zkLogin, stored with the ZK proving service. The verifying key was deployed as part of the validator software (protocol version 25 in release 1.10.1) that is used to verify the zkLogin transaction on Sui.

The following sections walk through all zkLogin artifacts, their security assumptions, and the consequences of loss or exposure.

The JWT's validity is scoped on the client ID (aud) to prevent phishing attacks. The same origin policy for the proof prevents the JWT obtained for a malicious application from being used for zkLogin. The JWT for the client ID is sent directly to the application frontend through the redirect URL. A leaked JWT for the specific client ID can compromise user privacy, as these tokens frequently hold sensitive information like usernames and emails. Furthermore, if a backend salt server is responsible for user salt management, the JWT could potentially be exploited to retrieve the user's salt, which introduces additional risks.

However, a JWT leak does not mean loss of funds as long as the corresponding ephemeral private key is safe.

The user salt is required to get access to the zkLogin wallet. This value is essential for both ZK proof generation and zkLogin address derivation.

The leak of user salt does not mean loss of funds, but it enables the attacker to associate the user's subject identifier (i.e. sub) with the Sui address. This can be problematic depending on whether pairwise or public subject identifiers are in use. In particular, there is no problem if pairwise IDs are used (e.g., Facebook) as the subject identifier is unique per RP. However, with public reusable IDs (e.g., Google and Twitch), the globally unique sub value can be used to identify users.

The ephemeral private key's lifespan is tied to the maximum epoch specified in nonce for creating a valid ZK proof. Should it be misplaced, a new ephemeral private key can be generated for transaction signing, accompanied by a freshly generated ZK proof using a new nonce. However, if the ephemeral private key is compromised, acquiring the user salt and the valid ZK proof would be necessary to move funds.

Obtaining the proof itself cannot create a valid zkLogin transaction because an ephemeral signature over the transaction is also needed.

By default, there is no linking between the OAuth subject identifier (i.e. sub) and a Sui address. This is the purpose of the user salt. The JWT is not published on-chain by default. The revealed values include iss , aud and kid so that the public input hash can be computed, any sensitive fields such as sub are used as private inputs when generating the proof.

The ZK proving service and the salt service (if maintained) can link the user identity since the user salt and JWT are known, but the two services are stateless by design.

In the future, the user can opt in to verify their OAuth identity associated with an Sui address on-chain.

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in

order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt (see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value. See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

The complete zkLogin flow

(Step 0) We use Groth16 for our protocol's zkSNARK instantiation, requiring a singular generation of a structured common reference string (CRS) linked to the circuit. A ceremony is conducted to generate the CRS, which is used to produce the proving key in the ZK Proving Service, the verifying key in Sui Authority. See the Ceremony section for more details.

(Step 1-3) The user begins by logging into an OpenID Provider (OP) to obtain a JWT containing a defined nonce. In particular, the user generates an ephemeral KeyPair (eph_sk, eph_pk) and embed eph_pk , along with expiry times (max_epoch) and randomness (jwt_randomness), into the nonce (see definition). After the user completes the OAuth login flow, an JWT can be found in the redirect URL in the application.

(Step 4-5) The application frontend then sends the JWT to a salt service. The salt service returns the unique user_salt based on iss, aud, sub upon validation of the JWT.

(Step 6-7) The user sends the ZK proving service the JWT, user salt, ephemeral public key, jwt randomness, key claim name (i.e. sub). The proving service generates a Zero-Knowledge Proof that takes these as private inputs and does the following: (a) Checks the nonce is derived correctly as defined (b) Checks that key claim value matches the corresponding field in the JWT, (c) Verifies the RSA signature from OP on the JWT, and (d) the address is consistent with the key claim value and user salt.

(Step 8): The application computes the user address based on iss, aud, sub, aud. This step can be done independently as long as the application has a valid JWT.

(Step 9-10) A transaction is signed using the ephemeral private key to generate an ephemeral signature. Finally, the user submits the transaction along with the ephemeral signature, ZK proof and other inputs to Sui.

(After Step 10) After submitted on chain, Sui Authorities verify the ZK proof against the provider JWKs from storage (agreed upon in consensus) and also the ephemeral signature.

Application frontend: This describes the wallet or frontend application that supports zkLogin. This frontend is responsible for storing the ephemeral private key, directing users to complete the OAuth login flow, creating and signing a zkLogin transaction.

Salt Backup Service: This is a backend service responsible for returning a salt per unique user. See zkLogin Integration Guide for other strategies to maintain salt.

ZK Proving Service: This is a backend service responsible for generating ZK proofs based on JWT, JWT randomness, user salt and max epoch. This proof is submitted on-chain along with the ephemeral signature for a zkLogin transaction.

The address is computed on the following inputs:

The address flag: zk_login_flag = 0x05 for zkLogin address. This serves as a domain separator as a signature scheme defined in

crypto agility .

kc_name_F = hashBytesToField(kc_name, maxKCNameLen) : Name of the key claim, e.g., sub . The sequence of bytes is mapped to a field element in BN254 using hashBytesToField (defined below).

kc_value_F = hashBytesToField(kc_value, maxKCValueLen) : The value of the key claim mapped using hashBytesToField .

aud_F = hashBytesToField(aud, maxAudValueLen) : The relying party (RP) identifier. See definition .

iss : The OpenID Provider (OP) identifier. See definition .

user_salt : A value introduced to unlink the OAuth identifier with the on-chain address.

Finally, we derive zk_login_address = Blake2b_256(zk_login_flag, iss_L, iss, addr_seed) where addr_seed = Poseidon_BN254(kc_name_F, kc_value_F, aud_F, Poseidon_BN254(user_salt) .

See below for all relevant OpenID terminology defined in spec and how they are used in zkLogin, along with definitions for protocol details.

OAuth 2.0 authorization server that is capable of authenticating the end-user and providing claims to an RP about the authentication event and the end-user. This is identified in the iss field in JWT payload. Check the table of available OPs for the entities zkLogin currently supports.

OAuth 2.0 client application requiring end-user authentication and claims from an OpenID provider. This is assigned by OP when the developer creates the application. This is identified in the aud field in JWT payload. This refers to any zkLogin enabled wallet or application.

Locally unique and never reassigned identifier within the issuer for the end user, which the RP is intended to consume. Sui uses this as the key claim to derive user address.

A JSON data structure that represents a set of public keys for an OP. A public endpoint (as in https://www.googleapis.com/oauth2/v3/certs) can be queried to retrieve the valid public keys corresponding to kid for the provider. Upon matching with the kid in the header of a JWT, the JWT can be verified against the payload and its corresponding JWK. In Sui, all authorities call the JWK endpoints independently, and update the latest view of JWKs for all supported providers during protocol upgrades. The correctness of JWKs is guaranteed by the quorum (2f+1) of validator stake.

JWT is in the redirect URI to RP after the user completes the OAuth login flow (as in https://redirect.com?id_token=$JWT_TOKEN). The JWT contains a header , payload , and a signature . The signature is an RSA signature verified against jwt_message = header + . + payload and its JWK identified by kid . The payload contains a JSON of many claims that is a name-value pair. See below for the specific claims that are relevant to the zkLogin protocol.

Header

Payload

For a zkLogin transaction, the iat and exp claims (timestamp) are not used. Instead, the nonce specifies expiry times.

The claim used to derive a users' address is termed the "key claim", such as sub or email. Naturally, it's ideal to use claims that are fixed once and never changed again. zkLogin currently supports sub as the key claim because OpenID spec mandates that providers do not change this identifier. In the future, this can be extended to use email, username, and so on.

To preserve privacy of the OAuth artifacts, a zero-knowledge proof of possession of the artifacts is provided. zkLogin employs the Groth16 zkSNARK to instantiate the zero-knowledge proofs, as it is the most efficient general-purpose zkSNARK in terms of proof size and verification efficiency.

However, Groth16 needs a computation-specific common reference string (CRS) to be setup by a trusted party. With zkLogin expected to ensure the safe-keeping of high value transactions and the integrity of critical smart contracts, we cannot base the security of the system on the honesty of a single entity. Hence, to generate the CRS for the zkLogin circuit, it is vital to run a protocol which bases its security on the assumed honesty of a small fraction of a large number of parties.

The Sui zkLogin ceremony is essentially a cryptographic multi-party computation (MPC) performed by a diverse group of participants to generate this CRS. We follow the MPC protocol MMORPG described by Bowe, Gabizon and Miers. The protocol roughly proceeds in 2 phases. The first phase results in a series of powers of a secret quantity tau in the exponent of an elliptic curve element. Since this phase is circuit-agnostic, we adopted the result of the existing community contributed perpetual powers of tau . Our ceremony was the second phase, which is specific to the zkLogin circuit.

The MMORPG protocol is a sequential protocol, which allows an indefinite number of parties to participate in sequence, without the need of any prior synchronization or ordering. Each party needs to download the output of the previous party, generate entropy of its own and then layer it on top of the received result, producing its own contribution, which is then relayed to the next party. The protocol guarantees security, if at least one of the participants follows the protocol faithfully, generates strong entropy and discards it reliably.

We sent invitations to 100+ people with diverse backgrounds and affiliations: Sui validators, cryptographers, web3 experts, world-renowned academicians, and business leaders. We planned the ceremony to take place on the dates September 12-18, 2023, but allowed participants to join when they wanted with no fixed slots.

Since the MPC is sequential, each contributor needed to wait till the previous contributor finished in order to receive the previous contribution, follow the MPC steps and produce their own contribution. Due to this structure, we provisioned a queue where participants waited, while those who joined before them finished. To authenticate participants, we sent a unique activation code to each of them. The activation code was the secret key of a signing key pair, which had a dual purpose: it allowed the coordination server to associate the participant's email with the contribution, and to verify the contribution with the corresponding public key.

Participants had two options to contribute: through a browser or a docker. The browser option was more user-friendly for contributors to participate as everything happens in the browser. The Docker option required Docker setup but is more transparent —the Dockerfile and contributor source code are open-sourced and the whole process is verifiable. Moreover, the browser option utilizes snarkjs while the Docker option utilizes Kobi's implementation . This provided software variety and contributors could choose to contribute by whichever method they trust. In addition, participants could generate entropy via entering random text or making random cursor movements.

The zkLogin circuit and the ceremony client code were made open source and the links were made available to the participants to review before the ceremony. In addition, we also posted these developer docs and an audit report on the circuit from zkSecurity. We adopted challenge #0081 (resulting from 80 community contributions) from perpetual powers of tau in phase 1, which is circuit agnostic. We applied the output of the Drand random beacon at epoch #3298000 to remove bias. For phase 2, our ceremony had 111 contributions, 82 from browser and 29 from docker. Finally, we applied the output of the Drand random beacon at epoch #3320606 to remove bias from contributions. All intermediate files can be reproduced following instructions here for phase 1 and here for phase 2.

The final CRS along with the transcript of contribution of every participant is available in a public repository. Contributors received both the hash of the previous contribution they were working on and the resulting hash after their contribution, displayed on-screen and sent via email. They can compare these hashes with the transcripts publicly available on the ceremony site. In addition, anyone is able to check that the hashes are computed correctly and each contribution is properly incorporated in the finalized parameters.

Eventually, the final CRS was used to generate the proving key and verifying key. The proving key is used to generate zero knowledge proof for zkLogin, stored with the ZK proving service. The verifying key was deployed as part of the validator software (protocol version 25 in release 1.10.1) that is used to verify the zkLogin transaction on Sui.

The following sections walk through all zkLogin artifacts, their security assumptions, and the consequences of loss or exposure.

The JWT's validity is scoped on the client ID (aud) to prevent phishing attacks. The same origin policy for the proof prevents the JWT obtained for a malicious application from being used for zkLogin. The JWT for the client ID is sent directly to the application frontend through the redirect URL. A leaked JWT for the specific client ID can compromise user privacy, as these tokens frequently hold sensitive information like usernames and emails. Furthermore, if a backend salt server is responsible for user salt management, the JWT could potentially be exploited to retrieve the user's salt, which introduces additional risks.

However, a JWT leak does not mean loss of funds as long as the corresponding ephemeral private key is safe.

The user salt is required to get access to the zkLogin wallet. This value is essential for both ZK proof generation and zkLogin address derivation.

The leak of user salt does not mean loss of funds, but it enables the attacker to associate the user's subject identifier (i.e. sub) with the Sui address. This can be problematic depending on whether pairwise or public subject identifiers are in use. In particular, there is no problem if pairwise IDs are used (e.g., Facebook) as the subject identifier is unique per RP. However, with public reusable IDs (e.g., Google and Twitch), the globally unique sub value can be used to identify users.

The ephemeral private key's lifespan is tied to the maximum epoch specified in nonce for creating a valid ZK proof. Should it be misplaced, a new ephemeral private key can be generated for transaction signing, accompanied by a freshly generated ZK proof using a new nonce. However, if the ephemeral private key is compromised, acquiring the user salt and the valid ZK proof would be necessary to move funds.

Obtaining the proof itself cannot create a valid zkLogin transaction because an ephemeral signature over the transaction is also needed.

By default, there is no linking between the OAuth subject identifier (i.e. sub) and a Sui address. This is the purpose of the user salt. The JWT is not published on-chain by default. The revealed values include iss , aud and kid so that the public input hash can be computed, any sensitive fields such as sub are used as private inputs when generating the proof.

The ZK proving service and the salt service (if maintained) can link the user identity since the user salt and JWT are known, but the two services are stateless by design.

In the future, the user can opt in to verify their OAuth identity associated with an Sui address on-chain.

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt (see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value. See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

Entities

Application frontend: This describes the wallet or frontend application that supports zkLogin. This frontend is responsible for storing the ephemeral private key, directing users to complete the OAuth login flow, creating and signing a zkLogin transaction.

Salt Backup Service: This is a backend service responsible for returning a salt per unique user. See zkLogin Integration Guide for other strategies to maintain salt.

ZK Proving Service: This is a backend service responsible for generating ZK proofs based on JWT, JWT randomness, user salt and max epoch. This proof is submitted on-chain along with the ephemeral signature for a zkLogin transaction.

The address is computed on the following inputs:

The address flag: zk_login_flag = 0x05 for zkLogin address. This serves as a domain separator as a signature scheme defined in crypto agility .

kc_name_F = hashBytesToField(kc_name, maxKCNameLen) : Name of the key claim, e.g., sub . The sequence of bytes is mapped to a field element in BN254 using hashBytesToField (defined below).

kc_value_F = hashBytesToField(kc_value, maxKCValueLen) : The value of the key claim mapped using hashBytesToField .

aud_F = hashBytesToField(aud, maxAudValueLen) : The relying party (RP) identifier. See definition .

iss : The OpenID Provider (OP) identifier. See definition .

user_salt : A value introduced to unlink the OAuth identifier with the on-chain address.

Finally, we derive zk_login_address = Blake2b_256(zk_login_flag, iss_L, iss, addr_seed) where addr_seed = Poseidon_BN254(kc_name_F, kc_value_F, aud_F, Poseidon_BN254(user_salt) .

See below for all relevant OpenID terminology defined in spec and how they are used in zkLogin, along with definitions for protocol details.

OAuth 2.0 authorization server that is capable of authenticating the end-user and providing claims to an RP about the authentication event and the end-user. This is identified in the iss field in JWT payload. Check the table of available OPs for the entities zkLogin currently supports.

OAuth 2.0 client application requiring end-user authentication and claims from an OpenID provider. This is assigned by OP when the developer creates the application. This is identified in the aud field in JWT payload. This refers to any zkLogin enabled wallet or application.

Locally unique and never reassigned identifier within the issuer for the end user, which the RP is intended to consume. Sui uses this as the key claim to derive user address.

A JSON data structure that represents a set of public keys for an OP. A public endpoint (as in https://www.googleapis.com/oauth2/v3/certs) can be queried to retrieve the valid public keys corresponding to kid for the provider. Upon matching with the kid in the header of a JWT, the JWT can be verified against the payload and its corresponding JWK. In Sui, all authorities call the JWK endpoints independently, and update the latest view of JWKs for all supported providers during protocol upgrades. The correctness of JWKs is guaranteed by the quorum (2f+1) of validator stake.

JWT is in the redirect URI to RP after the user completes the OAuth login flow (as in https://redirect.com?id_token=$JWT_TOKEN). The JWT contains a header , payload , and a signature . The signature is an RSA signature verified against jwt_message = header + . + payload and its JWK identified by kid . The payload contains a JSON of many claims that is a name-value pair. See below for the specific claims that are relevant to the zkLogin protocol.

Header

Payload

For a zkLogin transaction, the iat and exp claims (timestamp) are not used. Instead, the nonce specifies expiry times.

The claim used to derive a users' address is termed the "key claim", such as sub or email. Naturally, it's ideal to use claims that are

fixed once and never changed again. zkLogin currently supports sub as the key claim because OpenID spec mandates that providers do not change this identifier. In the future, this can be extended to use email, username, and so on.

To preserve privacy of the OAuth artifacts, a zero-knowledge proof of possession of the artifacts is provided. zkLogin employs the Groth16 zkSNARK to instantiate the zero-knowledge proofs, as it is the most efficient general-purpose zkSNARK in terms of proof size and verification efficiency.

However, Groth16 needs a computation-specific common reference string (CRS) to be setup by a trusted party. With zkLogin expected to ensure the safe-keeping of high value transactions and the integrity of critical smart contracts, we cannot base the security of the system on the honesty of a single entity. Hence, to generate the CRS for the zkLogin circuit, it is vital to run a protocol which bases its security on the assumed honesty of a small fraction of a large number of parties.

The Sui zkLogin ceremony is essentially a cryptographic multi-party computation (MPC) performed by a diverse group of participants to generate this CRS. We follow the MPC protocol MMORPG described by Bowe, Gabizon and Miers. The protocol roughly proceeds in 2 phases. The first phase results in a series of powers of a secret quantity tau in the exponent of an elliptic curve element. Since this phase is circuit-agnostic, we adopted the result of the existing community contributed perpetual powers of tau . Our ceremony was the second phase, which is specific to the zkLogin circuit.

The MMORPG protocol is a sequential protocol, which allows an indefinite number of parties to participate in sequence, without the need of any prior synchronization or ordering. Each party needs to download the output of the previous party, generate entropy of its own and then layer it on top of the received result, producing its own contribution, which is then relayed to the next party. The protocol guarantees security, if at least one of the participants follows the protocol faithfully, generates strong entropy and discards it reliably.

We sent invitations to 100+ people with diverse backgrounds and affiliations: Sui validators, cryptographers, web3 experts, world-renowned academicians, and business leaders. We planned the ceremony to take place on the dates September 12-18, 2023, but allowed participants to join when they wanted with no fixed slots.

Since the MPC is sequential, each contributor needed to wait till the previous contributor finished in order to receive the previous contribution, follow the MPC steps and produce their own contribution. Due to this structure, we provisioned a queue where participants waited, while those who joined before them finished. To authenticate participants, we sent a unique activation code to each of them. The activation code was the secret key of a signing key pair, which had a dual purpose: it allowed the coordination server to associate the participant's email with the contribution, and to verify the contribution with the corresponding public key.

Participants had two options to contribute: through a browser or a docker. The browser option was more user-friendly for contributors to participate as everything happens in the browser. The Docker option required Docker setup but is more transparent —the Dockerfile and contributor source code are open-sourced and the whole process is verifiable. Moreover, the browser option utilizes snarkjs while the Docker option utilizes Kobi's implementation . This provided software variety and contributors could choose to contribute by whichever method they trust. In addition, participants could generate entropy via entering random text or making random cursor movements.

The zkLogin circuit and the ceremony client code were made open source and the links were made available to the participants to review before the ceremony. In addition, we also posted these developer docs and an audit report on the circuit from zkSecurity. We adopted challenge #0081 (resulting from 80 community contributions) from perpetual powers of tau in phase 1, which is circuit agnostic. We applied the output of the Drand random beacon at epoch #3298000 to remove bias. For phase 2, our ceremony had 111 contributions, 82 from browser and 29 from docker. Finally, we applied the output of the Drand random beacon at epoch #3320606 to remove bias from contributions. All intermediate files can be reproduced following instructions here for phase 1 and here for phase 2.

The final CRS along with the transcript of contribution of every participant is available in a public repository. Contributors received both the hash of the previous contribution they were working on and the resulting hash after their contribution, displayed on-screen and sent via email. They can compare these hashes with the transcripts publicly available on the ceremony site. In addition, anyone is able to check that the hashes are computed correctly and each contribution is properly incorporated in the finalized parameters.

Eventually, the final CRS was used to generate the proving key and verifying key. The proving key is used to generate zero knowledge proof for zkLogin, stored with the ZK proving service. The verifying key was deployed as part of the validator software (protocol version 25 in release 1.10.1) that is used to verify the zkLogin transaction on Sui.

The following sections walk through all zkLogin artifacts, their security assumptions, and the consequences of loss or exposure.

The JWT's validity is scoped on the client ID (aud) to prevent phishing attacks. The same origin policy for the proof prevents the JWT obtained for a malicious application from being used for zkLogin. The JWT for the client ID is sent directly to the application frontend through the redirect URL. A leaked JWT for the specific client ID can compromise user privacy, as these tokens frequently

hold sensitive information like usernames and emails. Furthermore, if a backend salt server is responsible for user salt management, the JWT could potentially be exploited to retrieve the user's salt, which introduces additional risks.

However, a JWT leak does not mean loss of funds as long as the corresponding ephemeral private key is safe.

The user salt is required to get access to the zkLogin wallet. This value is essential for both ZK proof generation and zkLogin address derivation.

The leak of user salt does not mean loss of funds, but it enables the attacker to associate the user's subject identifier (i.e. sub) with the Sui address. This can be problematic depending on whether pairwise or public subject identifiers are in use. In particular, there is no problem if pairwise IDs are used (e.g., Facebook) as the subject identifier is unique per RP. However, with public reusable IDs (e.g., Google and Twitch), the globally unique sub value can be used to identify users.

The ephemeral private key's lifespan is tied to the maximum epoch specified in nonce for creating a valid ZK proof. Should it be misplaced, a new ephemeral private key can be generated for transaction signing, accompanied by a freshly generated ZK proof using a new nonce. However, if the ephemeral private key is compromised, acquiring the user salt and the valid ZK proof would be necessary to move funds.

Obtaining the proof itself cannot create a valid zkLogin transaction because an ephemeral signature over the transaction is also needed.

By default, there is no linking between the OAuth subject identifier (i.e. sub) and a Sui address. This is the purpose of the user salt. The JWT is not published on-chain by default. The revealed values include iss , aud and kid so that the public input hash can be computed, any sensitive fields such as sub are used as private inputs when generating the proof.

The ZK proving service and the salt service (if maintained) can link the user identity since the user salt and JWT are known, but the two services are stateless by design.

In the future, the user can opt in to verify their OAuth identity associated with an Sui address on-chain.

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or

wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt (see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value. See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

Address definition

The address is computed on the following inputs:

The address flag: zk_login_flag = 0x05 for zkLogin address. This serves as a domain separator as a signature scheme defined in crypto agility .

kc_name_F = hashBytesToField(kc_name, maxKCNameLen) : Name of the key claim, e.g., sub . The sequence of bytes is mapped to a field element in BN254 using hashBytesToField (defined below).

kc_value_F = hashBytesToField(kc_value, maxKCValueLen) : The value of the key claim mapped using hashBytesToField .

aud_F = hashBytesToField(aud, maxAudValueLen) : The relying party (RP) identifier. See definition .

iss : The OpenID Provider (OP) identifier. See definition .

user_salt : A value introduced to unlink the OAuth identifier with the on-chain address.

Finally, we derive zk_login_address = Blake2b_256(zk_login_flag, iss_L, iss, addr_seed) where addr_seed = Poseidon_BN254(kc_name_F, kc_value_F, aud_F, Poseidon_BN254(user_salt) .

See below for all relevant OpenID terminology defined in spec and how they are used in zkLogin, along with definitions for protocol details.

OAuth 2.0 authorization server that is capable of authenticating the end-user and providing claims to an RP about the authentication event and the end-user. This is identified in the iss field in JWT payload. Check the table of available OPs for the entities zkLogin currently supports.

OAuth 2.0 client application requiring end-user authentication and claims from an OpenID provider. This is assigned by OP when the developer creates the application. This is identified in the aud field in JWT payload. This refers to any zkLogin enabled wallet or application.

Locally unique and never reassigned identifier within the issuer for the end user, which the RP is intended to consume. Sui uses this as the key claim to derive user address.

A JSON data structure that represents a set of public keys for an OP. A public endpoint (as in https://www.googleapis.com/oauth2/v3/certs) can be queried to retrieve the valid public keys corresponding to kid for the provider. Upon matching with the kid in the header of a JWT, the JWT can be verified against the payload and its corresponding JWK. In Sui, all authorities call the JWK endpoints independently, and update the latest view of JWKs for all supported providers during protocol upgrades. The correctness of JWKs is guaranteed by the quorum (2f+1) of validator stake.

JWT is in the redirect URI to RP after the user completes the OAuth login flow (as in https://redirect.com?id_token=$JWT_TOKEN). The JWT contains a header , payload , and a signature . The signature is an RSA signature verified against jwt_message = header + . + payload and its JWK identified by kid . The payload contains a JSON of many claims that is a

name-value pair. See below for the specific claims that are relevant to the zkLogin protocol.

Header

Payload

For a zkLogin transaction, the iat and exp claims (timestamp) are not used. Instead, the nonce specifies expiry times.

The claim used to derive a users' address is termed the "key claim", such as sub or email. Naturally, it's ideal to use claims that are fixed once and never changed again. zkLogin currently supports sub as the key claim because OpenID spec mandates that providers do not change this identifier. In the future, this can be extended to use email, username, and so on.

To preserve privacy of the OAuth artifacts, a zero-knowledge proof of possession of the artifacts is provided. zkLogin employs the Groth16 zkSNARK to instantiate the zero-knowledge proofs, as it is the most efficient general-purpose zkSNARK in terms of proof size and verification efficiency.

However, Groth16 needs a computation-specific common reference string (CRS) to be setup by a trusted party. With zkLogin expected to ensure the safe-keeping of high value transactions and the integrity of critical smart contracts, we cannot base the security of the system on the honesty of a single entity. Hence, to generate the CRS for the zkLogin circuit, it is vital to run a protocol which bases its security on the assumed honesty of a small fraction of a large number of parties.

The Sui zkLogin ceremony is essentially a cryptographic multi-party computation (MPC) performed by a diverse group of participants to generate this CRS. We follow the MPC protocol MMORPG described by Bowe, Gabizon and Miers. The protocol roughly proceeds in 2 phases. The first phase results in a series of powers of a secret quantity tau in the exponent of an elliptic curve element. Since this phase is circuit-agnostic, we adopted the result of the existing community contributed perpetual powers of tau . Our ceremony was the second phase, which is specific to the zkLogin circuit.

The MMORPG protocol is a sequential protocol, which allows an indefinite number of parties to participate in sequence, without the need of any prior synchronization or ordering. Each party needs to download the output of the previous party, generate entropy of its own and then layer it on top of the received result, producing its own contribution, which is then relayed to the next party. The protocol guarantees security, if at least one of the participants follows the protocol faithfully, generates strong entropy and discards it reliably.

We sent invitations to 100+ people with diverse backgrounds and affiliations: Sui validators, cryptographers, web3 experts, world-renowned academicians, and business leaders. We planned the ceremony to take place on the dates September 12-18, 2023, but allowed participants to join when they wanted with no fixed slots.

Since the MPC is sequential, each contributor needed to wait till the previous contributor finished in order to receive the previous contribution, follow the MPC steps and produce their own contribution. Due to this structure, we provisioned a queue where participants waited, while those who joined before them finished. To authenticate participants, we sent a unique activation code to each of them. The activation code was the secret key of a signing key pair, which had a dual purpose: it allowed the coordination server to associate the participant's email with the contribution, and to verify the contribution with the corresponding public key.

Participants had two options to contribute: through a browser or a docker. The browser option was more user-friendly for contributors to participate as everything happens in the browser. The Docker option required Docker setup but is more transparent —the Dockerfile and contributor source code are open-sourced and the whole process is verifiable. Moreover, the browser option utilizes snarkjs while the Docker option utilizes Kobi's implementation . This provided software variety and contributors could choose to contribute by whichever method they trust. In addition, participants could generate entropy via entering random text or making random cursor movements.

The zkLogin circuit and the ceremony client code were made open source and the links were made available to the participants to review before the ceremony. In addition, we also posted these developer docs and an audit report on the circuit from zkSecurity. We adopted challenge #0081 (resulting from 80 community contributions) from perpetual powers of tau in phase 1, which is circuit agnostic. We applied the output of the Drand random beacon at epoch #3298000 to remove bias. For phase 2, our ceremony had 111 contributions, 82 from browser and 29 from docker. Finally, we applied the output of the Drand random beacon at epoch #3320606 to remove bias from contributions. All intermediate files can be reproduced following instructions here for phase 1 and here for phase 2.

The final CRS along with the transcript of contribution of every participant is available in a public repository. Contributors received both the hash of the previous contribution they were working on and the resulting hash after their contribution, displayed on-screen and sent via email. They can compare these hashes with the transcripts publicly available on the ceremony site. In addition, anyone is able to check that the hashes are computed correctly and each contribution is properly incorporated in the finalized parameters.

Eventually, the final CRS was used to generate the proving key and verifying key. The proving key is used to generate zero

knowledge proof for zkLogin, stored with the ZK proving service. The verifying key was deployed as part of the validator software (protocol version 25 in release 1.10.1) that is used to verify the zkLogin transaction on Sui.

The following sections walk through all zkLogin artifacts, their security assumptions, and the consequences of loss or exposure.

The JWT's validity is scoped on the client ID (aud) to prevent phishing attacks. The same origin policy for the proof prevents the JWT obtained for a malicious application from being used for zkLogin. The JWT for the client ID is sent directly to the application frontend through the redirect URL. A leaked JWT for the specific client ID can compromise user privacy, as these tokens frequently hold sensitive information like usernames and emails. Furthermore, if a backend salt server is responsible for user salt management, the JWT could potentially be exploited to retrieve the user's salt, which introduces additional risks.

However, a JWT leak does not mean loss of funds as long as the corresponding ephemeral private key is safe.

The user salt is required to get access to the zkLogin wallet. This value is essential for both ZK proof generation and zkLogin address derivation.

The leak of user salt does not mean loss of funds, but it enables the attacker to associate the user's subject identifier (i.e. sub) with the Sui address. This can be problematic depending on whether pairwise or public subject identifiers are in use. In particular, there is no problem if pairwise IDs are used (e.g., Facebook) as the subject identifier is unique per RP. However, with public reusable IDs (e.g., Google and Twitch), the globally unique sub value can be used to identify users.

The ephemeral private key's lifespan is tied to the maximum epoch specified in nonce for creating a valid ZK proof. Should it be misplaced, a new ephemeral private key can be generated for transaction signing, accompanied by a freshly generated ZK proof using a new nonce. However, if the ephemeral private key is compromised, acquiring the user salt and the valid ZK proof would be necessary to move funds.

Obtaining the proof itself cannot create a valid zkLogin transaction because an ephemeral signature over the transaction is also needed.

By default, there is no linking between the OAuth subject identifier (i.e. sub) and a Sui address. This is the purpose of the user salt. The JWT is not published on-chain by default. The revealed values include iss , aud and kid so that the public input hash can be computed, any sensitive fields such as sub are used as private inputs when generating the proof.

The ZK proving service and the salt service (if maintained) can link the user identity since the user salt and JWT are known, but the two services are stateless by design.

In the future, the user can opt in to verify their OAuth identity associated with an Sui address on-chain.

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt (see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value. See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

Terminology and notations

See below for all relevant OpenID terminology defined in spec and how they are used in zkLogin, along with definitions for protocol details.

OAuth 2.0 authorization server that is capable of authenticating the end-user and providing claims to an RP about the authentication event and the end-user. This is identified in the iss field in JWT payload. Check the table of available OPs for the entities zkLogin currently supports.

OAuth 2.0 client application requiring end-user authentication and claims from an OpenID provider. This is assigned by OP when the developer creates the application. This is identified in the aud field in JWT payload. This refers to any zkLogin enabled wallet or application.

Locally unique and never reassigned identifier within the issuer for the end user, which the RP is intended to consume. Sui uses this as the key claim to derive user address.

A JSON data structure that represents a set of public keys for an OP. A public endpoint (as in https://www.googleapis.com/oauth2/v3/certs) can be queried to retrieve the valid public keys corresponding to kid for the provider. Upon matching with the kid in the header of a JWT, the JWT can be verified against the payload and its corresponding JWK. In Sui, all authorities call the JWK endpoints independently, and update the latest view of JWKs for all supported providers during protocol upgrades. The correctness of JWKs is guaranteed by the quorum (2f+1) of validator stake.

JWT is in the redirect URI to RP after the user completes the OAuth login flow (as in https://redirect.com?id_token=$JWT_TOKEN). The JWT contains a header , payload , and a signature . The signature is an RSA signature verified against jwt_message = header + . + payload and its JWK identified by kid . The payload contains a JSON of many claims that is a name-value pair. See below for the specific claims that are relevant to the zkLogin protocol.

Header

Payload

For a zkLogin transaction, the iat and exp claims (timestamp) are not used. Instead, the nonce specifies expiry times.

The claim used to derive a users' address is termed the "key claim", such as sub or email. Naturally, it's ideal to use claims that are fixed once and never changed again. zkLogin currently supports sub as the key claim because OpenID spec mandates that providers do not change this identifier. In the future, this can be extended to use email, username, and so on.

To preserve privacy of the OAuth artifacts, a zero-knowledge proof of possession of the artifacts is provided. zkLogin employs the Groth16 zkSNARK to instantiate the zero-knowledge proofs, as it is the most efficient general-purpose zkSNARK in terms of proof size and verification efficiency.

However, Groth16 needs a computation-specific common reference string (CRS) to be setup by a trusted party. With zkLogin expected to ensure the safe-keeping of high value transactions and the integrity of critical smart contracts, we cannot base the security of the system on the honesty of a single entity. Hence, to generate the CRS for the zkLogin circuit, it is vital to run a protocol which bases its security on the assumed honesty of a small fraction of a large number of parties.

The Sui zkLogin ceremony is essentially a cryptographic multi-party computation (MPC) performed by a diverse group of participants to generate this CRS. We follow the MPC protocol MMORPG described by Bowe, Gabizon and Miers. The protocol roughly proceeds in 2 phases. The first phase results in a series of powers of a secret quantity tau in the exponent of an elliptic curve element. Since this phase is circuit-agnostic, we adopted the result of the existing community contributed perpetual powers of tau . Our ceremony was the second phase, which is specific to the zkLogin circuit.

The MMORPG protocol is a sequential protocol, which allows an indefinite number of parties to participate in sequence, without the need of any prior synchronization or ordering. Each party needs to download the output of the previous party, generate entropy of its own and then layer it on top of the received result, producing its own contribution, which is then relayed to the next party. The protocol guarantees security, if at least one of the participants follows the protocol faithfully, generates strong entropy and discards it reliably.

We sent invitations to 100+ people with diverse backgrounds and affiliations: Sui validators, cryptographers, web3 experts, world-renowned academicians, and business leaders. We planned the ceremony to take place on the dates September 12-18, 2023, but allowed participants to join when they wanted with no fixed slots.

Since the MPC is sequential, each contributor needed to wait till the previous contributor finished in order to receive the previous contribution, follow the MPC steps and produce their own contribution. Due to this structure, we provisioned a queue where participants waited, while those who joined before them finished. To authenticate participants, we sent a unique activation code to each of them. The activation code was the secret key of a signing key pair, which had a dual purpose: it allowed the coordination server to associate the participant's email with the contribution, and to verify the contribution with the corresponding public key.

Participants had two options to contribute: through a browser or a docker. The browser option was more user-friendly for contributors to participate as everything happens in the browser. The Docker option required Docker setup but is more transparent —the Dockerfile and contributor source code are open-sourced and the whole process is verifiable. Moreover, the browser option utilizes snarkjs while the Docker option utilizes Kobi's implementation . This provided software variety and contributors could choose to contribute by whichever method they trust. In addition, participants could generate entropy via entering random text or making random cursor movements.

The zkLogin circuit and the ceremony client code were made open source and the links were made available to the participants to review before the ceremony. In addition, we also posted these developer docs and an audit report on the circuit from zkSecurity. We adopted challenge #0081 (resulting from 80 community contributions) from perpetual powers of tau in phase 1, which is circuit agnostic. We applied the output of the Drand random beacon at epoch #3298000 to remove bias. For phase 2, our ceremony had 111 contributions, 82 from browser and 29 from docker. Finally, we applied the output of the Drand random beacon at epoch #3320606 to remove bias from contributions. All intermediate files can be reproduced following instructions here for phase 1 and here for phase 2.

The final CRS along with the transcript of contribution of every participant is available in a public repository. Contributors received both the hash of the previous contribution they were working on and the resulting hash after their contribution, displayed on-screen and sent via email. They can compare these hashes with the transcripts publicly available on the ceremony site. In addition, anyone is able to check that the hashes are computed correctly and each contribution is properly incorporated in the finalized parameters.

Eventually, the final CRS was used to generate the proving key and verifying key. The proving key is used to generate zero knowledge proof for zkLogin, stored with the ZK proving service. The verifying key was deployed as part of the validator software (protocol version 25 in release 1.10.1) that is used to verify the zkLogin transaction on Sui.

The following sections walk through all zkLogin artifacts, their security assumptions, and the consequences of loss or exposure.

The JWT's validity is scoped on the client ID (aud) to prevent phishing attacks. The same origin policy for the proof prevents the JWT obtained for a malicious application from being used for zkLogin. The JWT for the client ID is sent directly to the application frontend through the redirect URL. A leaked JWT for the specific client ID can compromise user privacy, as these tokens frequently hold sensitive information like usernames and emails. Furthermore, if a backend salt server is responsible for user salt management, the JWT could potentially be exploited to retrieve the user's salt, which introduces additional risks.

However, a JWT leak does not mean loss of funds as long as the corresponding ephemeral private key is safe.

The user salt is required to get access to the zkLogin wallet. This value is essential for both ZK proof generation and zkLogin address derivation.

The leak of user salt does not mean loss of funds, but it enables the attacker to associate the user's subject identifier (i.e. sub) with the Sui address. This can be problematic depending on whether pairwise or public subject identifiers are in use. In particular, there is no problem if pairwise IDs are used (e.g., Facebook) as the subject identifier is unique per RP. However, with public reusable IDs (e.g., Google and Twitch), the globally unique sub value can be used to identify users.

The ephemeral private key's lifespan is tied to the maximum epoch specified in nonce for creating a valid ZK proof. Should it be misplaced, a new ephemeral private key can be generated for transaction signing, accompanied by a freshly generated ZK proof using a new nonce. However, if the ephemeral private key is compromised, acquiring the user salt and the valid ZK proof would be necessary to move funds.

Obtaining the proof itself cannot create a valid zkLogin transaction because an ephemeral signature over the transaction is also needed.

By default, there is no linking between the OAuth subject identifier (i.e. sub) and a Sui address. This is the purpose of the user salt. The JWT is not published on-chain by default. The revealed values include iss , aud and kid so that the public input hash can be computed, any sensitive fields such as sub are used as private inputs when generating the proof.

The ZK proving service and the salt service (if maintained) can link the user identity since the user salt and JWT are known, but the two services are stateless by design.

In the future, the user can opt in to verify their OAuth identity associated with an Sui address on-chain.

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a

good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt (see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value. See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

Ceremony

To preserve privacy of the OAuth artifacts, a zero-knowledge proof of possession of the artifacts is provided. zkLogin employs the Groth16 zkSNARK to instantiate the zero-knowledge proofs, as it is the most efficient general-purpose zkSNARK in terms of proof size and verification efficiency.

However, Groth16 needs a computation-specific common reference string (CRS) to be setup by a trusted party. With zkLogin expected to ensure the safe-keeping of high value transactions and the integrity of critical smart contracts, we cannot base the security of the system on the honesty of a single entity. Hence, to generate the CRS for the zkLogin circuit, it is vital to run a protocol which bases its security on the assumed honesty of a small fraction of a large number of parties.

The Sui zkLogin ceremony is essentially a cryptographic multi-party computation (MPC) performed by a diverse group of participants to generate this CRS. We follow the MPC protocol MMORPG described by Bowe, Gabizon and Miers. The protocol roughly proceeds in 2 phases. The first phase results in a series of powers of a secret quantity tau in the exponent of an elliptic curve element. Since this phase is circuit-agnostic, we adopted the result of the existing community contributed perpetual powers of tau . Our ceremony was the second phase, which is specific to the zkLogin circuit.

The MMORPG protocol is a sequential protocol, which allows an indefinite number of parties to participate in sequence, without the need of any prior synchronization or ordering. Each party needs to download the output of the previous party, generate entropy of its own and then layer it on top of the received result, producing its own contribution, which is then relayed to the next party. The protocol guarantees security, if at least one of the participants follows the protocol faithfully, generates strong entropy and discards it reliably.

We sent invitations to 100+ people with diverse backgrounds and affiliations: Sui validators, cryptographers, web3 experts, world-renowned academicians, and business leaders. We planned the ceremony to take place on the dates September 12-18, 2023, but allowed participants to join when they wanted with no fixed slots.

Since the MPC is sequential, each contributor needed to wait till the previous contributor finished in order to receive the previous contribution, follow the MPC steps and produce their own contribution. Due to this structure, we provisioned a queue where participants waited, while those who joined before them finished. To authenticate participants, we sent a unique activation code to each of them. The activation code was the secret key of a signing key pair, which had a dual purpose: it allowed the coordination server to associate the participant's email with the contribution, and to verify the contribution with the corresponding public key.

Participants had two options to contribute: through a browser or a docker. The browser option was more user-friendly for contributors to participate as everything happens in the browser. The Docker option required Docker setup but is more transparent —the Dockerfile and contributor source code are open-sourced and the whole process is verifiable. Moreover, the browser option utilizes snarkjs while the Docker option utilizes Kobi's implementation . This provided software variety and contributors could choose to contribute by whichever method they trust. In addition, participants could generate entropy via entering random text or making random cursor movements.

The zkLogin circuit and the ceremony client code were made open source and the links were made available to the participants to review before the ceremony. In addition, we also posted these developer docs and an audit report on the circuit from zkSecurity. We adopted challenge #0081 (resulting from 80 community contributions) from perpetual powers of tau in phase 1, which is circuit agnostic. We applied the output of the Drand random beacon at epoch #3298000 to remove bias. For phase 2, our ceremony had 111 contributions, 82 from browser and 29 from docker. Finally, we applied the output of the Drand random beacon at epoch

#3320606 to remove bias from contributions. All intermediate files can be reproduced following instructions here for phase 1 and here for phase 2.

The final CRS along with the transcript of contribution of every participant is available in a public repository. Contributors received both the hash of the previous contribution they were working on and the resulting hash after their contribution, displayed on-screen and sent via email. They can compare these hashes with the transcripts publicly available on the ceremony site. In addition, anyone is able to check that the hashes are computed correctly and each contribution is properly incorporated in the finalized parameters.

Eventually, the final CRS was used to generate the proving key and verifying key. The proving key is used to generate zero knowledge proof for zkLogin, stored with the ZK proving service. The verifying key was deployed as part of the validator software (protocol version 25 in release 1.10.1) that is used to verify the zkLogin transaction on Sui.

The following sections walk through all zkLogin artifacts, their security assumptions, and the consequences of loss or exposure.

The JWT's validity is scoped on the client ID (aud) to prevent phishing attacks. The same origin policy for the proof prevents the JWT obtained for a malicious application from being used for zkLogin. The JWT for the client ID is sent directly to the application frontend through the redirect URL. A leaked JWT for the specific client ID can compromise user privacy, as these tokens frequently hold sensitive information like usernames and emails. Furthermore, if a backend salt server is responsible for user salt management, the JWT could potentially be exploited to retrieve the user's salt, which introduces additional risks.

However, a JWT leak does not mean loss of funds as long as the corresponding ephemeral private key is safe.

The user salt is required to get access to the zkLogin wallet. This value is essential for both ZK proof generation and zkLogin address derivation.

The leak of user salt does not mean loss of funds, but it enables the attacker to associate the user's subject identifier (i.e. sub) with the Sui address. This can be problematic depending on whether pairwise or public subject identifiers are in use. In particular, there is no problem if pairwise IDs are used (e.g., Facebook) as the subject identifier is unique per RP. However, with public reusable IDs (e.g., Google and Twitch), the globally unique sub value can be used to identify users.

The ephemeral private key's lifespan is tied to the maximum epoch specified in nonce for creating a valid ZK proof. Should it be misplaced, a new ephemeral private key can be generated for transaction signing, accompanied by a freshly generated ZK proof using a new nonce. However, if the ephemeral private key is compromised, acquiring the user salt and the valid ZK proof would be necessary to move funds.

Obtaining the proof itself cannot create a valid zkLogin transaction because an ephemeral signature over the transaction is also needed.

By default, there is no linking between the OAuth subject identifier (i.e. sub) and a Sui address. This is the purpose of the user salt. The JWT is not published on-chain by default. The revealed values include iss , aud and kid so that the public input hash can be computed, any sensitive fields such as sub are used as private inputs when generating the proof.

The ZK proving service and the salt service (if maintained) can link the user identity since the user salt and JWT are known, but the two services are stateless by design.

In the future, the user can opt in to verify their OAuth identity associated with an Sui address on-chain.

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt (see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value. See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

Finalization

The final CRS along with the transcript of contribution of every participant is available in a public repository. Contributors received both the hash of the previous contribution they were working on and the resulting hash after their contribution, displayed on-screen and sent via email. They can compare these hashes with the transcripts publicly available on the ceremony site. In addition, anyone is able to check that the hashes are computed correctly and each contribution is properly incorporated in the finalized parameters.

Eventually, the final CRS was used to generate the proving key and verifying key. The proving key is used to generate zero knowledge proof for zkLogin, stored with the ZK proving service. The verifying key was deployed as part of the validator software (protocol version 25 in release 1.10.1) that is used to verify the zkLogin transaction on Sui.

The following sections walk through all zkLogin artifacts, their security assumptions, and the consequences of loss or exposure.

The JWT's validity is scoped on the client ID (aud) to prevent phishing attacks. The same origin policy for the proof prevents the JWT obtained for a malicious application from being used for zkLogin. The JWT for the client ID is sent directly to the application frontend through the redirect URL. A leaked JWT for the specific client ID can compromise user privacy, as these tokens frequently hold sensitive information like usernames and emails. Furthermore, if a backend salt server is responsible for user salt management, the JWT could potentially be exploited to retrieve the user's salt, which introduces additional risks.

However, a JWT leak does not mean loss of funds as long as the corresponding ephemeral private key is safe.

The user salt is required to get access to the zkLogin wallet. This value is essential for both ZK proof generation and zkLogin address derivation.

The leak of user salt does not mean loss of funds, but it enables the attacker to associate the user's subject identifier (i.e. sub) with the Sui address. This can be problematic depending on whether pairwise or public subject identifiers are in use. In particular, there is no problem if pairwise IDs are used (e.g., Facebook) as the subject identifier is unique per RP. However, with public reusable IDs

(e.g., Google and Twitch), the globally unique sub value can be used to identify users.

The ephemeral private key's lifespan is tied to the maximum epoch specified in nonce for creating a valid ZK proof. Should it be misplaced, a new ephemeral private key can be generated for transaction signing, accompanied by a freshly generated ZK proof using a new nonce. However, if the ephemeral private key is compromised, acquiring the user salt and the valid ZK proof would be necessary to move funds.

Obtaining the proof itself cannot create a valid zkLogin transaction because an ephemeral signature over the transaction is also needed.

By default, there is no linking between the OAuth subject identifier (i.e. sub) and a Sui address. This is the purpose of the user salt. The JWT is not published on-chain by default. The revealed values include iss , aud and kid so that the public input hash can be computed, any sensitive fields such as sub are used as private inputs when generating the proof.

The ZK proving service and the salt service (if maintained) can link the user identity since the user salt and JWT are known, but the two services are stateless by design.

In the future, the user can opt in to verify their OAuth identity associated with an Sui address on-chain.

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt

(see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value.

See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

Security and privacy

The following sections walk through all zkLogin artifacts, their security assumptions, and the consequences of loss or exposure.

The JWT's validity is scoped on the client ID (aud) to prevent phishing attacks. The same origin policy for the proof prevents the JWT obtained for a malicious application from being used for zkLogin. The JWT for the client ID is sent directly to the application frontend through the redirect URL. A leaked JWT for the specific client ID can compromise user privacy, as these tokens frequently hold sensitive information like usernames and emails. Furthermore, if a backend salt server is responsible for user salt management, the JWT could potentially be exploited to retrieve the user's salt, which introduces additional risks.

However, a JWT leak does not mean loss of funds as long as the corresponding ephemeral private key is safe.

The user salt is required to get access to the zkLogin wallet. This value is essential for both ZK proof generation and zkLogin address derivation.

The leak of user salt does not mean loss of funds, but it enables the attacker to associate the user's subject identifier (i.e. sub) with the Sui address. This can be problematic depending on whether pairwise or public subject identifiers are in use. In particular, there is no problem if pairwise IDs are used (e.g., Facebook) as the subject identifier is unique per RP. However, with public reusable IDs (e.g., Google and Twitch), the globally unique sub value can be used to identify users.

The ephemeral private key's lifespan is tied to the maximum epoch specified in nonce for creating a valid ZK proof. Should it be misplaced, a new ephemeral private key can be generated for transaction signing, accompanied by a freshly generated ZK proof using a new nonce. However, if the ephemeral private key is compromised, acquiring the user salt and the valid ZK proof would be necessary to move funds.

Obtaining the proof itself cannot create a valid zkLogin transaction because an ephemeral signature over the transaction is also needed.

By default, there is no linking between the OAuth subject identifier (i.e. sub) and a Sui address. This is the purpose of the user salt. The JWT is not published on-chain by default. The revealed values include iss , aud and kid so that the public input hash can be computed, any sensitive fields such as sub are used as private inputs when generating the proof.

The ZK proving service and the salt service (if maintained) can link the user identity since the user salt and JWT are known, but the two services are stateless by design.

In the future, the user can opt in to verify their OAuth identity associated with an Sui address on-chain.

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt (see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value. See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

FAQ

zkLogin can support providers that work with OpenID Connect built on top of the OAuth 2.0 framework. This is a subset of OAuth 2.0 compatible providers. See latest table for all enabled providers. Other compatible providers will be enabled via protocol upgrades in the future.

Traditional private key wallets demand users to consistently recall mnemonics and passphrases, necessitating secure storage to prevent fund loss from private key compromise.

On the other hand, a zkLogin wallet only requires an ephemeral private key storage with session expiry and the OAuth login flow with expiry. Forgetting an ephemeral key does not result in loss of funds, because a user can always sign in again to generate a new ephemeral key and a new ZK proof.

Multi-Party Computation (MPC) and Multisig wallets rely on multiple keys or distributing multiple key shares and then defining a threshold value for accepting a signature.

zkLogin does not split any individual private keys, but ephemeral private keys are registered using a fresh nonce when the user authenticates with the OAuth provider. The primary advantage of zkLogin is that the user does not need to manage any persistent private key anywhere, not even with any private keys management techniques like MPC or Multisig.

You can think of zkLogin as a 2FA scheme for an address, where the first part is user's OAuth account and the second is the user's salt.

Furthermore, because Sui native supports Multisig wallets, one can always include one or more zkLogin signers inside a Multisig wallet for additional security, such as using the zkLogin part as 2FA in k-of-N settings.

Because zkLogin is a 2FA system, an attacker that has compromised your OAuth account cannot access your zkLogin address

unless they have separately compromised your salt.

Yes. You must be able to log into your OAuth account and produce a current JWT in order to use zkLogin.

A forgotten OAuth credential can typically be recovered by resetting the password in that provider. In the unfortunate event where user's OAuth credentials get compromised, an adversary will still require to obtain user_salt , but also learn which wallet is used in order to take over that account. Note that modern user_salt providers may have additional 2FA security measures in place to prevent provision of user's salt even to entities that present a valid, non-expired JWT.

It's also important to highlight that due to the fact that zkLogin addresses do not expose any information about the user's identity or wallet used, targeted attacks by just monitoring the blockchain are more difficult. Finally, on the unfortunate event where one loses access to their OAuth account permanently, access to that wallet is lost. But if recovery from a lost OAuth account is desired, a good suggestion for wallet providers is to support the native Sui Multisig functionality and add a backup method. Note that it's even possible to have a Multisig wallet that all signers are using zkLogin, i.e. an 1-of-2 Multisig zkLogin wallet where the first part is Google and the second Facebook OAuth, respectively.

No. The zkLogin wallet address is derived differently compared to a private key address.

zkLogin address is derived from sub , iss , aud and user_salt .

The address will not change if the user logs in to the same wallet with the same OAuth provider, since sub , iss , aud and user_salt (see definitions) will remain unchanged in the JWT, even though the JWT itself may look different every time the user logs in.

However, if the user logs in with different OAuth providers, your address will change because the iss and aud are defined distinctly per provider.

In addition, each wallet or application maintains its own user_salt , so logging with the same provider for different wallets may also result in different addresses.

See more on address definition .

Yes, this is possible by using a different wallet provider or different user_salt for each account. This is useful for separating funds between different accounts.

A zkLogin wallet is a non-custodial or unhosted wallet.

A custodial or hosted wallet is where a third party (the custodian) controls the private keys on behalf of a wallet user. No such third-party exists for zkLogin wallets.

Instead, a zkLogin wallet can be viewed as a 2-out-of-2 Multisig where the two credentials are the user's OAuth credentials (maintained by the user) and the salt. In other words, neither the OAuth provider, the wallet vendor, the ZK proving service or the salt service provider is a custodian.

No. Proof generation is only required when ephemeral KeyPair expires. Since the nonce is defined by the ephemeral public key (eph_pk) and expiry (max_epoch), the ZK proof is valid until what the expiry is committed to nonce in the JWT. The ZK proof can be cached and the same ephemeral key can be used to sign transactions till it expires.

zkLogin is a Sui native primitive and not a feature of a particular application or wallet. It can be used by any Sui developer, including on mobile.

Yes, the user can follow the OAuth providers' recovery flow. The ephemeral private key can be refreshed and after completing a new OAuth login flow, the user can obtain new ZK proof and sign transactions with the refreshed key.

Due to the way Groth16 works, we impose length restrictions on several fields in the JWT. Some of the fields that are length-restricted include aud, iss, the JWT's header and payload. For example, zkLogin can currently only work with aud values of up to length 120 (this value is not yet final). In general, we tried to make sure that the restrictions are as generous as possible. We have decided on these values after looking at as many JWTs that we could get our hands on.

While providing social login with Web2 credentials for Web3 wallet is not a new concept, the existing solutions have one or more of the trust assumptions:

Some of the existing deployed solutions rely on some of these assumptions. Web3Auth and DAuth social login requires deployment of custom OAuth verifiers to Web3auth Auth Network nodes to verify the JWT. Magic Wallet and Privy also require custom OAuth identity issuer and verifiers to adopt the DID standard. All of the solutions still require persistent private keys management, either with trusted parties like AWS via delegation, Shamir Secret Sharing or MPC.

The key differentiators that zkLogin brings to Sui are:

Native Support in Sui: Unlike other solutions that are blockchain agnostic, zkLogin is deployed just for Sui. This means a zkLogin transaction can be combined with Multisig and sponsored transactions seamlessly.

Self-Custodial without additional trust: We leverage the nonce field in JWT to commit to ephemeral public key, so no persistent private key management is required with any trusted parties. In addition, the JWK itself is an oracle agreed upon by the quorum of stakes by the validators with trusting any source of authority.

Full privacy: Nothing is required to submit on-chain except the ZK proof and the ephemeral signature.

Compatible with Existing Identity Providers: zkLogin is compatible with providers that adopt OpenID Connect. No need to trust any intermediate identity issuers or verifiers other than the OAuth providers themselves.

The following options support a zkLogin signature over either transaction data or personal message using the JWK state on Sui and current epoch.

Use Sui Typescript SDK. This initializes a GraphQL client and calls the endpoint under the hood.

Use the GraphQL endpoint directly: https://sui-[network].mystenlabs.com/graphql , changing [network] to the appropriate value. See the GraphQL documentation for more details. This is recommended if you do not plan to run any servers or handle any JWK rotations.

Use the Sui Keytool CLI . This is recommended for debug usage.

Use a self-hosted server endpoint and call this endpoint, as described in zklogin-verifier . This provides logic flexibility.

Yes. See the Multisig Guide for more details.

Related links