zkLogin Integration Guide

Here is the high-level flow the wallet or frontend application must implement to support zkLogin-enabled transactions:

Let's dive into the specific implementation details.

To use the zkLogin TypeScript SDK in your project, run the following command in your project root:

If you want to use the latest experimental version:

Generate an ephemeral key pair. Follow the same process as you would generating a key pair in a traditional wallet. See Sui SDK for details.

Set the expiration time for the ephemeral key pair. The wallet decides whether the maximum epoch is the current epoch or later. The wallet also determines whether this is adjustable by the user.

Assemble the OAuth URL with configured client ID, redirect URL, ephemeral public key and nonce: This is what the application sends the user to complete the login flow with a computed nonce .

The auth flow URL can be constructed with $CLIENT_ID , $REDIRECT_URL and $NONCE .

For some providers ("Yes" for "Auth Flow Only"), the JWT can be found immediately in the redirect URL after the auth flow.

For other providers ("No" for "Auth Flow Only"), the auth flow only returns a code ($AUTH_CODE) in redirect URL. To retrieve the JWT, an additional POST call is required with "Token Exchange URL".

Upon successful redirection, the OpenID provider attaches the JWT as a URL parameter. The following is an example using the Google flow.

The id_token param is the JWT in encoded format. You can validate the correctness of the encoded token and investigate its structure by pasting it in the jwt.io website.

To decode the JWT you can use a library like: jwt_decode: and map the response to the provided type JwtPayload :

zkLogin uses the user salt to compute the zkLogin Sui address (see definition). The salt must be a 16-byte value or a integer smaller than 2n**128n . There are several options for the application to maintain the user salt:

Here's an example request and response for the Mysten Labs-maintained salt server (using option 4). If you want to use the Mysten Labs salt server, please refer to Enoki docs and contact us. Only valid JWT authenticated with whitelisted client IDs are accepted.

User salt is used to disconnect the OAuth identifier (sub) from the on-chain Sui address to avoid linking Web2 credentials with Web3 credentials. While losing or misusing the salt could enable this link, it wouldn't compromise fund control or zkLogin asset authority. See more discussion here .

Once the OAuth flow completes, the JWT can be found in the redirect URL. Along with the user salt, the zkLogin address can be derived as follows:

The next step is to fetch the ZK proof. This is an attestation (proof) over the ephemeral key pair that proves the ephemeral key pair is valid.

First, generate the extended ephemeral public key to use as an input to the ZKP.

You need to fetch a new ZK proof if the previous ephemeral key pair is expired or is otherwise inaccessible.

Because generating a ZK proof can be resource-intensive and potentially slow on the client side, it's advised that wallets utilize a backend service endpoint dedicated to ZK proof generation.

There are two options:

If you want to use the Mysten hosted ZK Proving Service for Mainnet, please refer to Enoki docs and contact us for accessing it.

Use the prover-dev endpoint (https://prover-dev.mystenlabs.com/v1) freely for testing on Devnet. Note that you can submit proofs generated with this endpoint for Devnet zkLogin transactions only; submitting them to Testnet or Mainnet fails.

You can use BigInt or Base64 encoding for extendedEphemeralPublicKey , jwtRandomness , and salt . The following examples show two sample requests with the first using BigInt encoding and the second using Base64.

Response:

To avoid possible CORS errors in Frontend apps, it is suggested to delegate this call to a backend service.

The response can be mapped to the inputs parameter type of getZkLoginSignature of zkLogin SDK.

Install Git Large File Storage (an open-source Git extension for large file versioning) before downloading the zkey.

Download the Groth16 proving key zkey file . There are zkeys available for all Sui networks. See the Ceremony section for more details on how the main proving key is generated.

Main zkey (for Mainnet and Testnet)

Test zkey (for Devnet)

To verify the download contains the correct zkey file, you can run the following command to check the Blake2b hash: b2sum ${file_name}.zkey .

For the next step, you need two Docker images from the mysten/zklogin repository (tagged as prover and prover-fe). To simplify, a docker compose file is available that automates this process. Run docker compose with the downloaded zkey from the same directory as the YAML file.

A few important things to note:

The backend service (mysten/zklogin :prover-stable) is compute-heavy. Use at least the minimum recommended 16 cores and 16GB RAM. Using weaker instances can lead to timeout errors with the message "Call to rapidsnark service took longer than 15s". You can adjust the environment variable PROVER_TIMEOUT to set a different timeout value, for example, PROVER_TIMEOUT=30 for a timeout of 30 seconds.

If you want to compile the prover from scratch (for performance reasons), please see our fork of rapidsnark . You'd need to compile and launch the prover in server mode.

Setting DEBUG=* turns on all logs in the prover-fe service some of which may contain PII. Consider using DEBUG=zkLogin :info ,jwks in production environments.

First, sign the transaction bytes with the ephemeral private key using the key pair generated previously. This is the same as traditional KeyPair signing . Make sure that the transaction sender is also defined.

Next, generate an address seed by combining userSalt , sub (subject ID), and aud (audience).

Set the address seed and the partial zkLogin signature to be the inputs parameter.

You can now serialize the zkLogin signature by combining the ZK proof (inputs), the maxEpoch , and the ephemeral signature (userSignature).

Finally, execute the transaction.

As previously documented, each ZK proof is tied to an ephemeral key pair. So you can reuse the proof to sign any number of transactions until the ephemeral key pair expires (until the current epoch crosses maxEpoch).

You might want to cache the ephemeral key pair along with the ZKP for future uses.

However, the ephemeral key pair needs to be treated as a secret akin to a key pair in a traditional wallet. This is because if both the ephemeral private key and ZK proof are revealed to an attacker, then they can typically sign any transaction on behalf of the user (using the same process described previously).

Consequently, you should not store them persistently in a storage location that is not secure, on any platform. For example, on browsers, use session storage instead of local storage to store the ephemeral key pair and the ZK proof. This is because session storage automatically clears its data when the browser session ends, while data in local storage persists indefinitely.

Compared to traditional signatures, zkLogin signatures take a longer time to generate. For example, the prover that Mysten Labs maintains typically takes about three seconds to return a proof, which runs on a machine with 16 vCPUs and 64 GB RAM. Using more powerful machines, such as those with physical CPUs or graphics processing units (GPUs), can reduce the proving time further.

Carefully consider how many requests your application needs to make to the prover. Broadly speaking, the right metric to consider is the number of active user sessions and not the number of signatures. This is because you can cache the same ZK proof and reuse it across the session, as previously explained. For example, if you expect a million active user sessions per day, then you need a prover that can handle one or two requests per second (RPS), assuming evenly distributed traffic.

The prover that Mysten Labs maintains is set to auto-scale to handle traffic surges. If you are not sure whether Mysten Labs can handle a specific number of requests or expect a sudden spike in the number of prover requests your application needs to make, please reach out to us on Discord . Our plan is to horizontally scale the prover to handle any RPS you require.

Install the zkLogin TypeScript SDK

To use the zkLogin TypeScript SDK in your project, run the following command in your project root:

If you want to use the latest experimental version:

Generate an ephemeral key pair. Follow the same process as you would generating a key pair in a traditional wallet. See Sui SDK for details.

Set the expiration time for the ephemeral key pair. The wallet decides whether the maximum epoch is the current epoch or later. The wallet also determines whether this is adjustable by the user.

Assemble the OAuth URL with configured client ID, redirect URL, ephemeral public key and nonce: This is what the application sends the user to complete the login flow with a computed nonce .

The auth flow URL can be constructed with $CLIENT_ID , $REDIRECT_URL and $NONCE .

For some providers ("Yes" for "Auth Flow Only"), the JWT can be found immediately in the redirect URL after the auth flow.

For other providers ("No" for "Auth Flow Only"), the auth flow only returns a code ($AUTH_CODE) in redirect URL. To retrieve the JWT, an additional POST call is required with "Token Exchange URL".

Upon successful redirection, the OpenID provider attaches the JWT as a URL parameter. The following is an example using the Google flow.

The id_token param is the JWT in encoded format. You can validate the correctness of the encoded token and investigate its structure by pasting it in the jwt.io website.

To decode the JWT you can use a library like: jwt_decode: and map the response to the provided type JwtPayload :

zkLogin uses the user salt to compute the zkLogin Sui address (see definition). The salt must be a 16-byte value or a integer smaller than 2n**128n . There are several options for the application to maintain the user salt:

Here's an example request and response for the Mysten Labs-maintained salt server (using option 4). If you want to use the Mysten Labs salt server, please refer to Enoki docs and contact us. Only valid JWT authenticated with whitelisted client IDs are accepted.

User salt is used to disconnect the OAuth identifier (sub) from the on-chain Sui address to avoid linking Web2 credentials with Web3 credentials. While losing or misusing the salt could enable this link, it wouldn't compromise fund control or zkLogin asset authority. See more discussion here .

Once the OAuth flow completes, the JWT can be found in the redirect URL. Along with the user salt, the zkLogin address can be derived as follows:

The next step is to fetch the ZK proof. This is an attestation (proof) over the ephemeral key pair that proves the ephemeral key pair is valid.

First, generate the extended ephemeral public key to use as an input to the ZKP.

You need to fetch a new ZK proof if the previous ephemeral key pair is expired or is otherwise inaccessible.

Because generating a ZK proof can be resource-intensive and potentially slow on the client side, it's advised that wallets utilize a backend service endpoint dedicated to ZK proof generation.

There are two options:

If you want to use the Mysten hosted ZK Proving Service for Mainnet, please refer to Enoki docs and contact us for accessing it.

Use the prover-dev endpoint (https://prover-dev.mystenlabs.com/v1) freely for testing on Devnet. Note that you can submit proofs generated with this endpoint for Devnet zkLogin transactions only; submitting them to Testnet or Mainnet fails.

You can use BigInt or Base64 encoding for extendedEphemeralPublicKey , jwtRandomness , and salt . The following examples show two sample requests with the first using BigInt encoding and the second using Base64.

Response:

To avoid possible CORS errors in Frontend apps, it is suggested to delegate this call to a backend service.

The response can be mapped to the inputs parameter type of getZkLoginSignature of zkLogin SDK.

Install Git Large File Storage (an open-source Git extension for large file versioning) before downloading the zkey.

Download the Groth16 proving key zkey file . There are zkeys available for all Sui networks. See the Ceremony section for more details on how the main proving key is generated.

Main zkey (for Mainnet and Testnet)

Test zkey (for Devnet)

To verify the download contains the correct zkey file, you can run the following command to check the Blake2b hash: b2sum ${file_name}.zkey .

For the next step, you need two Docker images from the mysten/zklogin repository (tagged as prover and prover-fe). To simplify, a docker compose file is available that automates this process. Run docker compose with the downloaded zkey from the same directory as the YAML file.

A few important things to note:

The backend service (mysten/zklogin :prover-stable) is compute-heavy. Use at least the minimum recommended 16 cores and 16GB RAM. Using weaker instances can lead to timeout errors with the message "Call to rapidsnark service took longer than 15s". You can adjust the environment variable PROVER_TIMEOUT to set a different timeout value, for example, PROVER_TIMEOUT=30 for a timeout of 30 seconds.

If you want to compile the prover from scratch (for performance reasons), please see our fork of rapidsnark . You'd need to compile and launch the prover in server mode.

Setting DEBUG=* turns on all logs in the prover-fe service some of which may contain PII. Consider using DEBUG=zkLogin :info ,jwks in production environments.

First, sign the transaction bytes with the ephemeral private key using the key pair generated previously. This is the same as traditional KeyPair signing . Make sure that the transaction sender is also defined.

Next, generate an address seed by combining userSalt , sub (subject ID), and aud (audience).

Set the address seed and the partial zkLogin signature to be the inputs parameter.

You can now serialize the zkLogin signature by combining the ZK proof (inputs), the maxEpoch , and the ephemeral signature (userSignature).

Finally, execute the transaction.

As previously documented, each ZK proof is tied to an ephemeral key pair. So you can reuse the proof to sign any number of transactions until the ephemeral key pair expires (until the current epoch crosses maxEpoch).

You might want to cache the ephemeral key pair along with the ZKP for future uses.

However, the ephemeral key pair needs to be treated as a secret akin to a key pair in a traditional wallet. This is because if both the ephemeral private key and ZK proof are revealed to an attacker, then they can typically sign any transaction on behalf of the user (using the same process described previously).

Consequently, you should not store them persistently in a storage location that is not secure, on any platform. For example, on browsers, use session storage instead of local storage to store the ephemeral key pair and the ZK proof. This is because session storage automatically clears its data when the browser session ends, while data in local storage persists indefinitely.

Compared to traditional signatures, zkLogin signatures take a longer time to generate. For example, the prover that Mysten Labs maintains typically takes about three seconds to return a proof, which runs on a machine with 16 vCPUs and 64 GB RAM. Using more powerful machines, such as those with physical CPUs or graphics processing units (GPUs), can reduce the proving time further.

Carefully consider how many requests your application needs to make to the prover. Broadly speaking, the right metric to consider is the number of active user sessions and not the number of signatures. This is because you can cache the same ZK proof and reuse it across the session, as previously explained. For example, if you expect a million active user sessions per day, then you need a prover that can handle one or two requests per second (RPS), assuming evenly distributed traffic.

The prover that Mysten Labs maintains is set to auto-scale to handle traffic surges. If you are not sure whether Mysten Labs can handle a specific number of requests or expect a sudden spike in the number of prover requests your application needs to make, please reach out to us on Discord . Our plan is to horizontally scale the prover to handle any RPS you require.

Get JWT

Generate an ephemeral key pair. Follow the same process as you would generating a key pair in a traditional wallet. See Sui SDK for details.

Set the expiration time for the ephemeral key pair. The wallet decides whether the maximum epoch is the current epoch or later. The wallet also determines whether this is adjustable by the user.

Assemble the OAuth URL with configured client ID, redirect URL, ephemeral public key and nonce: This is what the application sends the user to complete the login flow with a computed nonce .

The auth flow URL can be constructed with $CLIENT_ID , $REDIRECT_URL and $NONCE .

For some providers ("Yes" for "Auth Flow Only"), the JWT can be found immediately in the redirect URL after the auth flow.

For other providers ("No" for "Auth Flow Only"), the auth flow only returns a code ($AUTH_CODE) in redirect URL. To retrieve the JWT, an additional POST call is required with "Token Exchange URL".

Upon successful redirection, the OpenID provider attaches the JWT as a URL parameter. The following is an example using the Google flow.

The id_token param is the JWT in encoded format. You can validate the correctness of the encoded token and investigate its structure by pasting it in the jwt.io website.

To decode the JWT you can use a library like: jwt_decode: and map the response to the provided type JwtPayload :

zkLogin uses the user salt to compute the zkLogin Sui address (see definition). The salt must be a 16-byte value or a integer smaller than 2n**128n . There are several options for the application to maintain the user salt:

Here's an example request and response for the Mysten Labs-maintained salt server (using option 4). If you want to use the Mysten Labs salt server, please refer to Enoki docs and contact us. Only valid JWT authenticated with whitelisted client IDs are accepted.

User salt is used to disconnect the OAuth identifier (sub) from the on-chain Sui address to avoid linking Web2 credentials with Web3 credentials. While losing or misusing the salt could enable this link, it wouldn't compromise fund control or zkLogin asset authority. See more discussion here .

Once the OAuth flow completes, the JWT can be found in the redirect URL. Along with the user salt, the zkLogin address can be derived as follows:

The next step is to fetch the ZK proof. This is an attestation (proof) over the ephemeral key pair that proves the ephemeral key pair is valid.

First, generate the extended ephemeral public key to use as an input to the ZKP.

You need to fetch a new ZK proof if the previous ephemeral key pair is expired or is otherwise inaccessible.

Because generating a ZK proof can be resource-intensive and potentially slow on the client side, it's advised that wallets utilize a backend service endpoint dedicated to ZK proof generation.

There are two options:

If you want to use the Mysten hosted ZK Proving Service for Mainnet, please refer to Enoki docs and contact us for accessing it.

Use the prover-dev endpoint (https://prover-dev.mystenlabs.com/v1) freely for testing on Devnet. Note that you can submit proofs generated with this endpoint for Devnet zkLogin transactions only; submitting them to Testnet or Mainnet fails.

You can use BigInt or Base64 encoding for extendedEphemeralPublicKey , jwtRandomness , and salt . The following examples show two sample requests with the first using BigInt encoding and the second using Base64.

Response:

To avoid possible CORS errors in Frontend apps, it is suggested to delegate this call to a backend service.

The response can be mapped to the inputs parameter type of getZkLoginSignature of zkLogin SDK.

Install Git Large File Storage (an open-source Git extension for large file versioning) before downloading the zkey.

Download the Groth16 proving key zkey file . There are zkeys available for all Sui networks. See the Ceremony section for more details on how the main proving key is generated.

Main zkey (for Mainnet and Testnet)

Test zkey (for Devnet)

To verify the download contains the correct zkey file, you can run the following command to check the Blake2b hash: b2sum ${file_name}.zkey .

For the next step, you need two Docker images from the mysten/zklogin repository (tagged as prover and prover-fe). To simplify, a docker compose file is available that automates this process. Run docker compose with the downloaded zkey from the same directory as the YAML file.

A few important things to note:

The backend service (mysten/zklogin :prover-stable) is compute-heavy. Use at least the minimum recommended 16 cores and 16GB RAM. Using weaker instances can lead to timeout errors with the message "Call to rapidsnark service took longer than 15s". You can adjust the environment variable PROVER_TIMEOUT to set a different timeout value, for example, PROVER_TIMEOUT=30 for a timeout of 30 seconds.

If you want to compile the prover from scratch (for performance reasons), please see our fork of rapidsnark . You'd need to compile and launch the prover in server mode.

Setting DEBUG=* turns on all logs in the prover-fe service some of which may contain PII. Consider using DEBUG=zkLogin :info ,jwks in production environments.

First, sign the transaction bytes with the ephemeral private key using the key pair generated previously. This is the same as traditional KeyPair signing . Make sure that the transaction sender is also defined.

Next, generate an address seed by combining userSalt , sub (subject ID), and aud (audience).

Set the address seed and the partial zkLogin signature to be the inputs parameter.

You can now serialize the zkLogin signature by combining the ZK proof (inputs), the maxEpoch , and the ephemeral signature (userSignature).

Finally, execute the transaction.

As previously documented, each ZK proof is tied to an ephemeral key pair. So you can reuse the proof to sign any number of transactions until the ephemeral key pair expires (until the current epoch crosses maxEpoch).

You might want to cache the ephemeral key pair along with the ZKP for future uses.

However, the ephemeral key pair needs to be treated as a secret akin to a key pair in a traditional wallet. This is because if both the ephemeral private key and ZK proof are revealed to an attacker, then they can typically sign any transaction on behalf of the user (using the same process described previously).

Consequently, you should not store them persistently in a storage location that is not secure, on any platform. For example, on browsers, use session storage instead of local storage to store the ephemeral key pair and the ZK proof. This is because session storage automatically clears its data when the browser session ends, while data in local storage persists indefinitely.

Compared to traditional signatures, zkLogin signatures take a longer time to generate. For example, the prover that Mysten Labs maintains typically takes about three seconds to return a proof, which runs on a machine with 16 vCPUs and 64 GB RAM. Using more powerful machines, such as those with physical CPUs or graphics processing units (GPUs), can reduce the proving time further.

Carefully consider how many requests your application needs to make to the prover. Broadly speaking, the right metric to consider is the number of active user sessions and not the number of signatures. This is because you can cache the same ZK proof and reuse it across the session, as previously explained. For example, if you expect a million active user sessions per day, then you need a prover that can handle one or two requests per second (RPS), assuming evenly distributed traffic.

The prover that Mysten Labs maintains is set to auto-scale to handle traffic surges. If you are not sure whether Mysten Labs can handle a specific number of requests or expect a sudden spike in the number of prover requests your application needs to make, please reach out to us on Discord . Our plan is to horizontally scale the prover to handle any RPS you require.

Decoding JWT

Upon successful redirection, the OpenID provider attaches the JWT as a URL parameter. The following is an example using the Google flow.

The id_token param is the JWT in encoded format. You can validate the correctness of the encoded token and investigate its structure by pasting it in the jwt.io website.

To decode the JWT you can use a library like: jwt_decode: and map the response to the provided type JwtPayload :

zkLogin uses the user salt to compute the zkLogin Sui address (see definition). The salt must be a 16-byte value or a integer smaller than 2n**128n . There are several options for the application to maintain the user salt:

Here's an example request and response for the Mysten Labs-maintained salt server (using option 4). If you want to use the Mysten Labs salt server, please refer to Enoki docs and contact us. Only valid JWT authenticated with whitelisted client IDs are accepted.

User salt is used to disconnect the OAuth identifier (sub) from the on-chain Sui address to avoid linking Web2 credentials with Web3 credentials. While losing or misusing the salt could enable this link, it wouldn't compromise fund control or zkLogin asset authority. See more discussion here .

Once the OAuth flow completes, the JWT can be found in the redirect URL. Along with the user salt, the zkLogin address can be derived as follows:

The next step is to fetch the ZK proof. This is an attestation (proof) over the ephemeral key pair that proves the ephemeral key pair is valid.

First, generate the extended ephemeral public key to use as an input to the ZKP.

You need to fetch a new ZK proof if the previous ephemeral key pair is expired or is otherwise inaccessible.

Because generating a ZK proof can be resource-intensive and potentially slow on the client side, it's advised that wallets utilize a backend service endpoint dedicated to ZK proof generation.

There are two options:

If you want to use the Mysten hosted ZK Proving Service for Mainnet, please refer to Enoki docs and contact us for accessing it.

Use the prover-dev endpoint (https://prover-dev.mystenlabs.com/v1) freely for testing on Devnet. Note that you can submit proofs generated with this endpoint for Devnet zkLogin transactions only; submitting them to Testnet or Mainnet fails.

You can use BigInt or Base64 encoding for extendedEphemeralPublicKey , jwtRandomness , and salt . The following examples show two sample requests with the first using BigInt encoding and the second using Base64.

Response:

To avoid possible CORS errors in Frontend apps, it is suggested to delegate this call to a backend service.

The response can be mapped to the inputs parameter type of getZkLoginSignature of zkLogin SDK.

Install Git Large File Storage (an open-source Git extension for large file versioning) before downloading the zkey.

Download the Groth16 proving key zkey file . There are zkeys available for all Sui networks. See the Ceremony section for more details on how the main proving key is generated.

Main zkey (for Mainnet and Testnet)

Test zkey (for Devnet)

To verify the download contains the correct zkey file, you can run the following command to check the Blake2b hash: b2sum ${file_name}.zkey .

For the next step, you need two Docker images from the mysten/zklogin repository (tagged as prover and prover-fe). To simplify, a docker compose file is available that automates this process. Run docker compose with the downloaded zkey from the same directory as the YAML file.

A few important things to note:

The backend service (mysten/zklogin :prover-stable) is compute-heavy. Use at least the minimum recommended 16 cores and 16GB RAM. Using weaker instances can lead to timeout errors with the message "Call to rapidsnark service took longer than 15s". You can adjust the environment variable PROVER_TIMEOUT to set a different timeout value, for example, PROVER_TIMEOUT=30 for a timeout of 30 seconds.

If you want to compile the prover from scratch (for performance reasons), please see our fork of rapidsnark . You'd need to compile and launch the prover in server mode.

Setting DEBUG=* turns on all logs in the prover-fe service some of which may contain PII. Consider using DEBUG=zkLogin :info ,jwks in production environments.

First, sign the transaction bytes with the ephemeral private key using the key pair generated previously. This is the same as traditional KeyPair signing . Make sure that the transaction sender is also defined.

Next, generate an address seed by combining userSalt , sub (subject ID), and aud (audience).

Set the address seed and the partial zkLogin signature to be the inputs parameter.

You can now serialize the zkLogin signature by combining the ZK proof (inputs), the maxEpoch , and the ephemeral signature (userSignature).

Finally, execute the transaction.

As previously documented, each ZK proof is tied to an ephemeral key pair. So you can reuse the proof to sign any number of transactions until the ephemeral key pair expires (until the current epoch crosses maxEpoch).

You might want to cache the ephemeral key pair along with the ZKP for future uses.

However, the ephemeral key pair needs to be treated as a secret akin to a key pair in a traditional wallet. This is because if both the ephemeral private key and ZK proof are revealed to an attacker, then they can typically sign any transaction on behalf of the user (using the same process described previously).

Consequently, you should not store them persistently in a storage location that is not secure, on any platform. For example, on browsers, use session storage instead of local storage to store the ephemeral key pair and the ZK proof. This is because session storage automatically clears its data when the browser session ends, while data in local storage persists indefinitely.

Compared to traditional signatures, zkLogin signatures take a longer time to generate. For example, the prover that Mysten Labs maintains typically takes about three seconds to return a proof, which runs on a machine with 16 vCPUs and 64 GB RAM. Using more powerful machines, such as those with physical CPUs or graphics processing units (GPUs), can reduce the proving time further.

Carefully consider how many requests your application needs to make to the prover. Broadly speaking, the right metric to consider is the number of active user sessions and not the number of signatures. This is because you can cache the same ZK proof and reuse it across the session, as previously explained. For example, if you expect a million active user sessions per day, then you need a prover that can handle one or two requests per second (RPS), assuming evenly distributed traffic.

The prover that Mysten Labs maintains is set to auto-scale to handle traffic surges. If you are not sure whether Mysten Labs can handle a specific number of requests or expect a sudden spike in the number of prover requests your application needs to make, please reach out to us on Discord . Our plan is to horizontally scale the prover to handle any RPS you require.

User salt management

zkLogin uses the user salt to compute the zkLogin Sui address (see definition). The salt must be a 16-byte value or a integer smaller than 2n**128n . There are several options for the application to maintain the user salt:

Here's an example request and response for the Mysten Labs-maintained salt server (using option 4). If you want to use the Mysten Labs salt server, please refer to Enoki docs and contact us. Only valid JWT authenticated with whitelisted client IDs are accepted.

User salt is used to disconnect the OAuth identifier (sub) from the on-chain Sui address to avoid linking Web2 credentials with Web3 credentials. While losing or misusing the salt could enable this link, it wouldn't compromise fund control or zkLogin asset authority. See more discussion here .

Once the OAuth flow completes, the JWT can be found in the redirect URL. Along with the user salt, the zkLogin address can be derived as follows:

The next step is to fetch the ZK proof. This is an attestation (proof) over the ephemeral key pair that proves the ephemeral key pair is valid.

First, generate the extended ephemeral public key to use as an input to the ZKP.

You need to fetch a new ZK proof if the previous ephemeral key pair is expired or is otherwise inaccessible.

Because generating a ZK proof can be resource-intensive and potentially slow on the client side, it's advised that wallets utilize a backend service endpoint dedicated to ZK proof generation.

There are two options:

If you want to use the Mysten hosted ZK Proving Service for Mainnet, please refer to Enoki docs and contact us for accessing it.

Use the prover-dev endpoint (https://prover-dev.mystenlabs.com/v1) freely for testing on Devnet. Note that you can submit proofs generated with this endpoint for Devnet zkLogin transactions only; submitting them to Testnet or Mainnet fails.

You can use BigInt or Base64 encoding for extendedEphemeralPublicKey , jwtRandomness , and salt . The following examples show two sample requests with the first using BigInt encoding and the second using Base64.

Response:

To avoid possible CORS errors in Frontend apps, it is suggested to delegate this call to a backend service.

The response can be mapped to the inputs parameter type of getZkLoginSignature of zkLogin SDK.

Install Git Large File Storage (an open-source Git extension for large file versioning) before downloading the zkey.

Download the Groth16 proving key zkey file . There are zkeys available for all Sui networks. See the Ceremony section for more details on how the main proving key is generated.

Main zkey (for Mainnet and Testnet)

Test zkey (for Devnet)

To verify the download contains the correct zkey file, you can run the following command to check the Blake2b hash: b2sum ${file_name}.zkey .

For the next step, you need two Docker images from the mysten/zklogin repository (tagged as prover and prover-fe). To simplify, a docker compose file is available that automates this process. Run docker compose with the downloaded zkey from the same directory as the YAML file.

A few important things to note:

The backend service (mysten/zklogin :prover-stable) is compute-heavy. Use at least the minimum recommended 16 cores and 16GB RAM. Using weaker instances can lead to timeout errors with the message "Call to rapidsnark service took longer than 15s".

You can adjust the environment variable PROVER_TIMEOUT to set a different timeout value, for example, PROVER_TIMEOUT=30 for a timeout of 30 seconds.

If you want to compile the prover from scratch (for performance reasons), please see our fork of rapidsnark . You'd need to compile and launch the prover in server mode.

Setting DEBUG=* turns on all logs in the prover-fe service some of which may contain PII. Consider using DEBUG=zkLogin :info ,jwks in production environments.

First, sign the transaction bytes with the ephemeral private key using the key pair generated previously. This is the same as traditional KeyPair signing . Make sure that the transaction sender is also defined.

Next, generate an address seed by combining userSalt , sub (subject ID), and aud (audience).

Set the address seed and the partial zkLogin signature to be the inputs parameter.

You can now serialize the zkLogin signature by combining the ZK proof (inputs), the maxEpoch , and the ephemeral signature (userSignature).

Finally, execute the transaction.

As previously documented, each ZK proof is tied to an ephemeral key pair. So you can reuse the proof to sign any number of transactions until the ephemeral key pair expires (until the current epoch crosses maxEpoch).

You might want to cache the ephemeral key pair along with the ZKP for future uses.

However, the ephemeral key pair needs to be treated as a secret akin to a key pair in a traditional wallet. This is because if both the ephemeral private key and ZK proof are revealed to an attacker, then they can typically sign any transaction on behalf of the user (using the same process described previously).

Consequently, you should not store them persistently in a storage location that is not secure, on any platform. For example, on browsers, use session storage instead of local storage to store the ephemeral key pair and the ZK proof. This is because session storage automatically clears its data when the browser session ends, while data in local storage persists indefinitely.

Compared to traditional signatures, zkLogin signatures take a longer time to generate. For example, the prover that Mysten Labs maintains typically takes about three seconds to return a proof, which runs on a machine with 16 vCPUs and 64 GB RAM. Using more powerful machines, such as those with physical CPUs or graphics processing units (GPUs), can reduce the proving time further.

Carefully consider how many requests your application needs to make to the prover. Broadly speaking, the right metric to consider is the number of active user sessions and not the number of signatures. This is because you can cache the same ZK proof and reuse it across the session, as previously explained. For example, if you expect a million active user sessions per day, then you need a prover that can handle one or two requests per second (RPS), assuming evenly distributed traffic.

The prover that Mysten Labs maintains is set to auto-scale to handle traffic surges. If you are not sure whether Mysten Labs can handle a specific number of requests or expect a sudden spike in the number of prover requests your application needs to make, please reach out to us on Discord . Our plan is to horizontally scale the prover to handle any RPS you require.

Get the user's Sui address

Once the OAuth flow completes, the JWT can be found in the redirect URL. Along with the user salt, the zkLogin address can be derived as follows:

The next step is to fetch the ZK proof. This is an attestation (proof) over the ephemeral key pair that proves the ephemeral key pair is valid.

First, generate the extended ephemeral public key to use as an input to the ZKP.

You need to fetch a new ZK proof if the previous ephemeral key pair is expired or is otherwise inaccessible.

Because generating a ZK proof can be resource-intensive and potentially slow on the client side, it's advised that wallets utilize a backend service endpoint dedicated to ZK proof generation.

There are two options:

If you want to use the Mysten hosted ZK Proving Service for Mainnet, please refer to Enoki docs and contact us for accessing it.

Use the prover-dev endpoint (https://prover-dev.mystenlabs.com/v1) freely for testing on Devnet. Note that you can submit proofs generated with this endpoint for Devnet zkLogin transactions only; submitting them to Testnet or Mainnet fails.

You can use BigInt or Base64 encoding for extendedEphemeralPublicKey , jwtRandomness , and salt . The following examples show two sample requests with the first using BigInt encoding and the second using Base64.

Response:

To avoid possible CORS errors in Frontend apps, it is suggested to delegate this call to a backend service.

The response can be mapped to the inputs parameter type of getZkLoginSignature of zkLogin SDK.

Install Git Large File Storage (an open-source Git extension for large file versioning) before downloading the zkey.

Download the Groth16 proving key zkey file . There are zkeys available for all Sui networks. See the Ceremony section for more details on how the main proving key is generated.

Main zkey (for Mainnet and Testnet)

Test zkey (for Devnet)

To verify the download contains the correct zkey file, you can run the following command to check the Blake2b hash: b2sum ${file_name}.zkey .

For the next step, you need two Docker images from the mysten/zklogin repository (tagged as prover and prover-fe). To simplify, a docker compose file is available that automates this process. Run docker compose with the downloaded zkey from the same directory as the YAML file.

A few important things to note:

The backend service (mysten/zklogin :prover-stable) is compute-heavy. Use at least the minimum recommended 16 cores and 16GB RAM. Using weaker instances can lead to timeout errors with the message "Call to rapidsnark service took longer than 15s". You can adjust the environment variable PROVER_TIMEOUT to set a different timeout value, for example, PROVER_TIMEOUT=30 for a timeout of 30 seconds.

If you want to compile the prover from scratch (for performance reasons), please see our fork of rapidsnark . You'd need to compile and launch the prover in server mode.

Setting DEBUG=* turns on all logs in the prover-fe service some of which may contain PII. Consider using DEBUG=zkLogin :info ,jwks in production environments.

First, sign the transaction bytes with the ephemeral private key using the key pair generated previously. This is the same as traditional KeyPair signing . Make sure that the transaction sender is also defined.

Next, generate an address seed by combining userSalt , sub (subject ID), and aud (audience).

Set the address seed and the partial zkLogin signature to be the inputs parameter.

You can now serialize the zkLogin signature by combining the ZK proof (inputs), the maxEpoch , and the ephemeral signature (userSignature).

Finally, execute the transaction.

As previously documented, each ZK proof is tied to an ephemeral key pair. So you can reuse the proof to sign any number of transactions until the ephemeral key pair expires (until the current epoch crosses maxEpoch).

You might want to cache the ephemeral key pair along with the ZKP for future uses.

However, the ephemeral key pair needs to be treated as a secret akin to a key pair in a traditional wallet. This is because if both the ephemeral private key and ZK proof are revealed to an attacker, then they can typically sign any transaction on behalf of the user (using the same process described previously).

Consequently, you should not store them persistently in a storage location that is not secure, on any platform. For example, on browsers, use session storage instead of local storage to store the ephemeral key pair and the ZK proof. This is because session storage automatically clears its data when the browser session ends, while data in local storage persists indefinitely.

Compared to traditional signatures, zkLogin signatures take a longer time to generate. For example, the prover that Mysten Labs maintains typically takes about three seconds to return a proof, which runs on a machine with 16 vCPUs and 64 GB RAM. Using more powerful machines, such as those with physical CPUs or graphics processing units (GPUs), can reduce the proving time further.

Carefully consider how many requests your application needs to make to the prover. Broadly speaking, the right metric to consider is the number of active user sessions and not the number of signatures. This is because you can cache the same ZK proof and reuse it across the session, as previously explained. For example, if you expect a million active user sessions per day, then you need a prover that can handle one or two requests per second (RPS), assuming evenly distributed traffic.

The prover that Mysten Labs maintains is set to auto-scale to handle traffic surges. If you are not sure whether Mysten Labs can handle a specific number of requests or expect a sudden spike in the number of prover requests your application needs to make, please reach out to us on Discord . Our plan is to horizontally scale the prover to handle any RPS you require.

Get the zero-knowledge proof

The next step is to fetch the ZK proof. This is an attestation (proof) over the ephemeral key pair that proves the ephemeral key pair is valid.

First, generate the extended ephemeral public key to use as an input to the ZKP.

You need to fetch a new ZK proof if the previous ephemeral key pair is expired or is otherwise inaccessible.

Because generating a ZK proof can be resource-intensive and potentially slow on the client side, it's advised that wallets utilize a backend service endpoint dedicated to ZK proof generation.

There are two options:

If you want to use the Mysten hosted ZK Proving Service for Mainnet, please refer to Enoki docs and contact us for accessing it.

Use the prover-dev endpoint (https://prover-dev.mystenlabs.com/v1) freely for testing on Devnet. Note that you can submit proofs generated with this endpoint for Devnet zkLogin transactions only; submitting them to Testnet or Mainnet fails.

You can use BigInt or Base64 encoding for extendedEphemeralPublicKey , jwtRandomness , and salt . The following examples show two sample requests with the first using BigInt encoding and the second using Base64.

Response:

To avoid possible CORS errors in Frontend apps, it is suggested to delegate this call to a backend service.

The response can be mapped to the inputs parameter type of getZkLoginSignature of zkLogin SDK.

Install Git Large File Storage (an open-source Git extension for large file versioning) before downloading the zkey.

Download the Groth16 proving key zkey file . There are zkeys available for all Sui networks. See the Ceremony section for more details on how the main proving key is generated.

Main zkey (for Mainnet and Testnet)

Test zkey (for Devnet)

To verify the download contains the correct zkey file, you can run the following command to check the Blake2b hash: b2sum ${file_name}.zkey .

For the next step, you need two Docker images from the mysten/zklogin repository (tagged as prover and prover-fe). To simplify, a docker compose file is available that automates this process. Run docker compose with the downloaded zkey from the same directory as the YAML file.

A few important things to note:

The backend service (mysten/zklogin :prover-stable) is compute-heavy. Use at least the minimum recommended 16 cores and 16GB RAM. Using weaker instances can lead to timeout errors with the message "Call to rapidsnark service took longer than 15s". You can adjust the environment variable PROVER_TIMEOUT to set a different timeout value, for example, PROVER_TIMEOUT=30 for a timeout of 30 seconds.

If you want to compile the prover from scratch (for performance reasons), please see our fork of rapidsnark . You'd need to compile and launch the prover in server mode.

Setting DEBUG=* turns on all logs in the prover-fe service some of which may contain PII. Consider using DEBUG=zkLogin :info ,jwks in production environments.

First, sign the transaction bytes with the ephemeral private key using the key pair generated previously. This is the same as traditional KeyPair signing . Make sure that the transaction sender is also defined.

Next, generate an address seed by combining userSalt , sub (subject ID), and aud (audience).

Set the address seed and the partial zkLogin signature to be the inputs parameter.

You can now serialize the zkLogin signature by combining the ZK proof (inputs), the maxEpoch , and the ephemeral signature (userSignature).

Finally, execute the transaction.

As previously documented, each ZK proof is tied to an ephemeral key pair. So you can reuse the proof to sign any number of transactions until the ephemeral key pair expires (until the current epoch crosses maxEpoch).

You might want to cache the ephemeral key pair along with the ZKP for future uses.

However, the ephemeral key pair needs to be treated as a secret akin to a key pair in a traditional wallet. This is because if both the ephemeral private key and ZK proof are revealed to an attacker, then they can typically sign any transaction on behalf of the user (using the same process described previously).

Consequently, you should not store them persistently in a storage location that is not secure, on any platform. For example, on browsers, use session storage instead of local storage to store the ephemeral key pair and the ZK proof. This is because session storage automatically clears its data when the browser session ends, while data in local storage persists indefinitely.

Compared to traditional signatures, zkLogin signatures take a longer time to generate. For example, the prover that Mysten Labs maintains typically takes about three seconds to return a proof, which runs on a machine with 16 vCPUs and 64 GB RAM. Using more powerful machines, such as those with physical CPUs or graphics processing units (GPUs), can reduce the proving time further.

Carefully consider how many requests your application needs to make to the prover. Broadly speaking, the right metric to consider is the number of active user sessions and not the number of signatures. This is because you can cache the same ZK proof and reuse it across the session, as previously explained. For example, if you expect a million active user sessions per day, then you need a prover that can handle one or two requests per second (RPS), assuming evenly distributed traffic.

The prover that Mysten Labs maintains is set to auto-scale to handle traffic surges. If you are not sure whether Mysten Labs can handle a specific number of requests or expect a sudden spike in the number of prover requests your application needs to make, please reach out to us on Discord . Our plan is to horizontally scale the prover to handle any RPS you require.

Assemble the zkLogin signature and submit the transaction

First, sign the transaction bytes with the ephemeral private key using the key pair generated previously. This is the same as traditional KeyPair signing . Make sure that the transaction sender is also defined.

Next, generate an address seed by combining userSalt , sub (subject ID), and aud (audience).

Set the address seed and the partial zkLogin signature to be the inputs parameter.

You can now serialize the zkLogin signature by combining the ZK proof (inputs), the maxEpoch , and the ephemeral signature (userSignature).

Finally, execute the transaction.

As previously documented, each ZK proof is tied to an ephemeral key pair. So you can reuse the proof to sign any number of transactions until the ephemeral key pair expires (until the current epoch crosses maxEpoch).

You might want to cache the ephemeral key pair along with the ZKP for future uses.

However, the ephemeral key pair needs to be treated as a secret akin to a key pair in a traditional wallet. This is because if both the ephemeral private key and ZK proof are revealed to an attacker, then they can typically sign any transaction on behalf of the user (using the same process described previously).

Consequently, you should not store them persistently in a storage location that is not secure, on any platform. For example, on browsers, use session storage instead of local storage to store the ephemeral key pair and the ZK proof. This is because session storage automatically clears its data when the browser session ends, while data in local storage persists indefinitely.

Compared to traditional signatures, zkLogin signatures take a longer time to generate. For example, the prover that Mysten Labs maintains typically takes about three seconds to return a proof, which runs on a machine with 16 vCPUs and 64 GB RAM. Using more powerful machines, such as those with physical CPUs or graphics processing units (GPUs), can reduce the proving time further.

Carefully consider how many requests your application needs to make to the prover. Broadly speaking, the right metric to consider is the number of active user sessions and not the number of signatures. This is because you can cache the same ZK proof and reuse it across the session, as previously explained. For example, if you expect a million active user sessions per day, then you need a prover that can handle one or two requests per second (RPS), assuming evenly distributed traffic.

The prover that Mysten Labs maintains is set to auto-scale to handle traffic surges. If you are not sure whether Mysten Labs can handle a specific number of requests or expect a sudden spike in the number of prover requests your application needs to make, please reach out to us on Discord . Our plan is to horizontally scale the prover to handle any RPS you require.

Caching the ephemeral private key and ZK proof

As previously documented, each ZK proof is tied to an ephemeral key pair. So you can reuse the proof to sign any number of transactions until the ephemeral key pair expires (until the current epoch crosses maxEpoch).

You might want to cache the ephemeral key pair along with the ZKP for future uses.

However, the ephemeral key pair needs to be treated as a secret akin to a key pair in a traditional wallet. This is because if both the ephemeral private key and ZK proof are revealed to an attacker, then they can typically sign any transaction on behalf of the user (using the same process described previously).

Consequently, you should not store them persistently in a storage location that is not secure, on any platform. For example, on browsers, use session storage instead of local storage to store the ephemeral key pair and the ZK proof. This is because session storage automatically clears its data when the browser session ends, while data in local storage persists indefinitely.

Compared to traditional signatures, zkLogin signatures take a longer time to generate. For example, the prover that Mysten Labs maintains typically takes about three seconds to return a proof, which runs on a machine with 16 vCPUs and 64 GB RAM. Using more powerful machines, such as those with physical CPUs or graphics processing units (GPUs), can reduce the proving time further.

Carefully consider how many requests your application needs to make to the prover. Broadly speaking, the right metric to consider is the number of active user sessions and not the number of signatures. This is because you can cache the same ZK proof and reuse it across the session, as previously explained. For example, if you expect a million active user sessions per day, then you need a prover that can handle one or two requests per second (RPS), assuming evenly distributed traffic.

The prover that Mysten Labs maintains is set to auto-scale to handle traffic surges. If you are not sure whether Mysten Labs can handle a specific number of requests or expect a sudden spike in the number of prover requests your application needs to make, please reach out to us on Discord . Our plan is to horizontally scale the prover to handle any RPS you require.

Efficiency considerations

Compared to traditional signatures, zkLogin signatures take a longer time to generate. For example, the prover that Mysten Labs maintains typically takes about three seconds to return a proof, which runs on a machine with 16 vCPUs and 64 GB RAM. Using more powerful machines, such as those with physical CPUs or graphics processing units (GPUs), can reduce the proving time further.

Carefully consider how many requests your application needs to make to the prover. Broadly speaking, the right metric to consider is the number of active user sessions and not the number of signatures. This is because you can cache the same ZK proof and reuse it across the session, as previously explained. For example, if you expect a million active user sessions per day, then you need a

prover that can handle one or two requests per second (RPS), assuming evenly distributed traffic.

The prover that Mysten Labs maintains is set to auto-scale to handle traffic surges. If you are not sure whether Mysten Labs can handle a specific number of requests or expect a sudden spike in the number of prover requests your application needs to make, please reach out to us on Discord . Our plan is to horizontally scale the prover to handle any RPS you require.

Related links