Verifying Proofs On-Chain

World ID proofs can be fully verified on-chain. After all, the source of truth for the decentralized protocol is on-chain. To verify a World ID proof, your smart contract will embed a call to theverifyProof method of the World ID contract, and then execute the rest of its logic as usual.

The smart contract starter kits (Foundry ,Hardhat) and thefrontend & on-chain monorepo template are great resources to help you get started with World ID. Using one of these repositories is strongly recommended to get started with World ID on-chain.

The following examples demonstrate the most common use case: verifying proofs from only Orb-verified users, for a single action, with a user's wallet address as the signal, while also enabling sybil-resistance. This setup is recommended for most users, as it is the most gas-efficient. For more information on use cases that require more complex setups (such as multiple actions or other types of signals), see theAdvanced On-Chain Verification page.

At the core of the World ID Protocol is the use oSemaphore . Semaphore is a zk-SNARK based privacy protocol that allows for the creation of anonymous credential systems developed by the Ethereum Foundation. Read more aboutThe Protocol andSemaphore .

IDKit Configuration

When verifying proofs on-chain, there are a few changes you have to make to your IDKit configuration. You must ensure that the app created in the Developer Portal is configured as an on-chain app, and you should only accept Orb credentials in IDKit, as World ID Device is not currently supported on-chain.

import { IDKitWidget } from

'@worldcoin/idkit'

const { address } =

useAddress () // get the user's wallet address

< IDKitWidget app_id = "app_GBkZ1KlVUdFTjeMXKlVUdFT"

// must be an app set to on-chain in Developer Portal action = "claim_nft" signal = {address} // proof will only verify if the signal is unchanged, this prevents tampering onSuccess = {onSuccess} // use onSuccess to call your smart contract // no use for handleVerify, so it is removed // use default verification_level (orb-only), as device credentials are not supported on-chain

{({ open }) => < button

onClick

{open}>Verify with World ID</ button

} </ IDKitWidget

Copy Copied!

Contract Addresses

The World ID Router contract is what you should use to verify proofs. It is deployed on multiple chains, and you can find the addresses for each chain in our contractsAddress Book .

hashToField Helper Function

Our contracts use a custom hash function that returns auint256 that is guaranteed to be in the field of the elliptic curve we use. This is necessary to ensure that theuint256 returned by the hash function can be used in our zero-knowledge proofs.

library

ByteHasher { /// @dev Creates a keccak256 hash of a bytestring. /// @param value The bytestring to hash /// @return The hash of the specified value /// @dev >> 8 makes sure that the result is included in our field function

hashToField (bytes

memory value) internal

pure

returns (uint256) { return

uint256 (keccak256 (abi. encodePacked (value)))

8 ; } } Copy Copied!

To use this function, simply import theByteHasher library and callhashToField on your bytestring.

import { ByteHasher } from

'./helpers/ByteHasher.sol' ;

contract HelloWorld { using

ByteHasher

for

bytes ; // {...} abi. encodePacked ('hello world'). hashToField (); // returns the keccak256 hash of 'hello world' as a uint256 // {...} Copy Copied!

Constructor

TheexternalNullifier is the unique identifier of the action performed in Semaphore, and its keccak256 hash (namedexternalNullifierHash) is what is passed to the World ID Router contract. It is a combination of the app ID and the action. You should typically set it in the constructor to save gas (as is done in this example), as it will not change if all users are performing the same action.

We additionally set thegroupId to1 , which limits this example to Orb-verified users only. World ID Device is currently not supported on-chain.

/// @dev This allows us to use our hashToField function on bytes using

ByteHasher

for

bytes ;

/// @notice Thrown when attempting to reuse a nullifier error

InvalidNullifier ();

/// @dev The address of the World ID Router contract that will be used for verifying proofs IWorldID internal

immutable worldId;

/// @dev The keccak256 hash of the externalNullifier (unique identifier of the action performed), combination of appId and action uint256

internal

immutable externalNullifierHash;

/// @dev The World ID group ID (1 for Orb-verified) uint256

internal

immutable groupId =

1 ;

/// @dev Whether a nullifier hash has been used already. Used to guarantee an action is only performed once by a single person mapping (uint256

=>

bool) internal nullifierHashes;

/// @param _worldId The address of the WorldIDRouter that will verify the proofs /// @param _appId The World ID App ID (from Developer Portal) /// @param _actionId The World ID Action (from Developer Portal) constructor (IWorldID

_worldId , string

memory

_appId , string

memory

_action) { worldId = _worldId; externalNullifierHash = abi . encodePacked (abi. encodePacked (_appId). hashToField () , _action) . hashToField (); } Copy Copied!

verifyProof

TheverifyProof method reverts if the proof is invalid, meaning you can just call it as part of your smart contract's logic and execute the rest of your logic after as usual.

Note that calling theverifyProof function by itself does not provide sybil-resistance, or prevent proof reuse -- it just verifies that the proof is valid.

However, this exampledoes implement sybil-resistance by checking that thenullifierHash has not been verified before.

TheverifyProof method takes the arguments below.

- root
-
  - The World ID root to verify against. This is obtained from the IDKit widget, and should be passed as-is.
- groupId
-
  - This must be1
- for Orb-verified users. World ID Device is currently not supported on-chain.
- signalHash
-
  - The keccak256 hash of the signal to verify.
- nullifierHash
-
  - Anonymous user ID for this action. This is obtained from the IDKit widget, and should just be passed as-is.
- externalNullifierHash
-
  - The externalNullifierHash, which identifies which app and action the user is verifying for.
- proof
-
  - The proof to verify. This is obtained from the IDKit widget.

/// @param signal An arbitrary input from the user that cannot be tampered with. In this case, it is the user's wallet address. /// @param root The root (returned by the IDKit widget). /// @param nullifierHash The nullifier hash for this proof, preventing double signaling (returned by the IDKit widget). /// @param proof The zero-knowledge proof that demonstrates the claimer is registered with World ID (returned by the IDKit widget). function

verifyAndExecute (address signal , uint256 root , uint256 nullifierHash , uint256 [8] calldata proof) public { // First, we make sure this person hasn't done this before if (nullifierHashes[nullifierHash]) revert

InvalidNullifier ();

// We now verify the provided proof is valid and the user is verified by World ID worldId. verifyProof (root , groupId ,

// set to "1" in the constructor abi. encodePacked (signal). hashToField () , nullifierHash , externalNullifierHash , proof);

// We now record the user has done this, so they can't do it again (sybil-resistance) nullifierHashes[nullifierHash] =

true ;

// Finally, execute your logic here, knowing the user is verified } Copy Copied!

All arguments are of typeuint256 , with the exception ofproof , which is of typeuint256[8] . Depending on how you're calling your smart contract, you might be required to unpack it into auint256[8] before passing it to the verifyProof method. To unpack it, use the following code:

viem ethers.js import { decodeAbiParameters } from

'viem'

const

unpackedProof

=

decodeAbiParameters ([{ type :

'uint256[8]' }] , proof)[0] Copy Copied!