ZKSECURITY

Audit of Hinkal Protocol Smart Contracts and Circom Circuits

September 6th, 2024

Introduction

On September 2nd, 2024, zkSecurity was engaged to conduct a security audit of the Hinkal Protocol’s Solidity smart contracts and Circom circuits. Over the course of one week, two dedicated consultants meticulously reviewed the Solidity smart contracts and Circom circuits, aiming to identify potential bugs and security vulnerabilities. The audit was conducted following a one-week pre-audit engagement between a zkSecurity consultant and the Hinkal team, focused on identifying security vulnerabilities and potential threats to the protocol.

During the audit, several observations and findings were identified, which have been communicated to the Hinkal Protocol team. The detailed findings and their implications are discussed in the subsequent sections of this report.

Scope

The audit, conducted by two consultants from zkSecurity, focused on a comprehensive review of the Solidity smart contracts and Circom circuits associated with the Hinkal Protocol. The assessment covered the following files:

`./libs/circom/`

- `AccessTokenChecker.circom`
- `ConditionalOverflowPreventer.circom`
- `ForceNotEqual.circom`
- `ForceMaxAllowedTimestamp.circom`
- `MerkleRootCalculator.circom`
- `NullifierCalculator.circom`
- `OriginalCommitmentCalculator.circom`
- `OriginalOrStakeCommitmentCalculator.circom`
- `OverflowPreventer.circom`
- `PointCompressor.circom`
- `ShouldFill.circom`
- `Signature.circom`
- `StakeCommitmentCalculator.circom`
- `StakeInputChecker.circom`
- `StakeProver.circom`
- `StakeProverPermissionless.circom`
- `StealthAddressCalculator.circom`
- `StealthAddressCompressor.circom`
- `SwapperM.circom`
- `SwapperME.circom`
- `receiver.circom`

`./hardhat/contracts/`

- `CircomDataBuilder.sol`
- `CrossChainAccessToken.sol`
- `ERC20TokenRegistry.sol`
- `Hinkal.sol`
- `HinkalBase.sol`
- `HinkalHelper.sol`
- `HinkalRelayWrapper.sol`
- `HinkalWrapper.sol`
- `Merkle.sol`
- `MerkleBase.sol`
- `MerkleRemovable.sol`
- `OwnerHinkal.sol`
- `RelayStore.sol`
- `Signer.sol`
- `Transferer.sol`
- `TransfererBase.sol`
- `VerifierFacade.sol`

`external-actions/hinkal-staking/`

- `HinkalStakeExternalAction.sol`
- `HinkalStakeDataDecoder.sol`
- `hToken.sol`

`xchain/connext/`

- `ConnextAction.sol`
- `ConnextRescueBuffer.sol`

The commit we audited was `117e8ad`.

Methodology

zkSecurity employed a comprehensive set of methodologies and heuristics to audit both the Solidity smart contracts and Circom circuits of the Hinkal Protocol. This section outlines the key approaches taken during the audit.

Firstly, we conducted an extensive review of the Solidity smart contracts, focusing on common vulnerabilities including, but not limited to, reentrancy, transaction-ordering and timestamp dependence, arithmetic errors, gas-related issues, access control weaknesses, and logic flaws. On the Circom side, our focus was on identifying under-constrained vulnerabilities due to insufficient constraints (such as variables assigned but not properly constrained and mismatches in arithmetic transformations), logic errors, incorrect use of external circuits, edge cases in modulo arithmetic, and out-of-circuit computations that are not properly constrained. Further, we tried to validate that all the private variables remained private and there were no information leakages.

Additionally, we analyzed the integration between Solidity and Circom, paying special attention to potential errors arising from incorrect assumptions made by the Solidity code about the Circom circuits and vice versa.

We also considered the following specific attack vectors and explored whether vulnerabilities could be identified within the protocol:

- Double Spending: Can users exploit a flaw to double-spend a nullifier?
- Slippage Exploits: Can an attacker exploit slippage risks associated with relayers?
- Reentrancy Attacks: Are the contracts adequately protected against reentrancy attacks?
- Access Token Theft: Can an attacker steal an access token? Is the access token securely bound to a user's public key?
- KYC Bypass: Is it possible for an attacker to acquire an access token without passing KYC?
- Signature Vulnerabilities: Are there risks of signature malleability or replay attacks?
- Cross-Chain Transfer Failures: In Connext xcall, could a cross-chain transfer silently fail while still triggering a fund transfer back to the user in **`` `ConnextAction.sol` ``**?
- Fee Bypass: Can an attacker bypass the relayer fee or Hinkal commission?
- Hook Contracts: The transactHook in **`` `Hinkal.sol` ``** allows user-deployed contracts to execute custom code via **`` `transactHook.beforeTransact()` ``** and **`` `transactHook.afterTransact()` ``**. Could this lead to malicious actions, especially when using both **`` `preHookContract` ``** and **`` `hookContract` ``**?
- Circuit Constraints: Are the Circom circuits properly constrained to prevent under-constrained vulnerabilities?
- ZKP Usage: Do the Solidity contracts correctly utilize the ZKPs generated by Circom? Are there any incorrect assumptions made based solely on the ZKPs?
- Merkle Tree Issues: Are the Merkle Trees correctly constructed? Could funds be locked in the protocol due to incorrect updates?
- Commitment Spending: Can commitments be duplicated?
- Is the new feature of Hinkal, i.e., staking and bridging, working correctly?
- Are the unspendable commitments created in **`` `SwapperME` ``** truly unspendable?
- Does the bridging functionally work properly, and is the access control correctly applied?

Recommendations

Based on the findings of the audit, we recommend the following steps to enhance the security and robustness of the Hinkal Protocol.

While the codebase was found to be generally well-designed and aligned with security best practices across both Solidity and Circom, there is room for improvement in the area of documentation. Specifically, clearer documentation of the interactions between different components—namely, the client-side code (out-of-scope for this audit), the smart contracts responsible for verifying ZKPs and managing on-chain data through the three primary Merkle Trees (the main Hinkal Merkle Tree, Staking Merkle Tree, and AccessToken Merkle Tree), and the Circom code that generates and verifies ZKPs—would significantly enhance code readability and maintainability.

Similarly, the Circom circuits should include more detailed documentation regarding their intended behavior. For example, in the `` `receiver.circom` `` circuit, it appears that the circuit is designed to validate the provided private key and verify the access token. However, the private key is not actually validated against a corresponding public key, which could lead to serious bugs in the future.

We also observed that while the existing tests and integration tests cover basic scenarios, they could be improved by incorporating a wider range of values and edge cases. For instance, the deposit/withdraw/transfer tests could be expanded to ensure that the protocol correctly records deposits and handles unusual ERC20 tokens, such as fee-on-transfer and rebase tokens. If the protocol is intended to support these token types, more comprehensive tests should be included to verify proper functionality.

Further, we identify unnecessary complexity in the Circom circuit and we have recommended ways to make the Circom circuits simpler. Additionally, we identified that there are no tests for the Circom code and we suggest including both positive and negative case for the Circom circuit especially covering all the edge cases.

The Role of Hinkal’s Smart Contracts Owner

The Owner of Hinkal’s smart contracts holds significant capabilities, which introduces potential risks if proper security practices are not followed, or if the Owner’s address (or multisig wallet controlling the Owner role) is compromised. While this risk is acknowledged within Hinkal’s existing threat model, it is crucial to adhere to best practices mitigating the chance of contract control being hijacked.

We have identified a few scenarios where the Owner’s elevated permissions could lead to potential misuse:

- **Bridging**: The Owner can modify parameters such as slippage, potentially causing bridging transactions to fail on the destination chain. This could be exploited to invoke the `` `rescueAsset` `` function, allowing the Owner to retrieve stuck funds before users have the chance to recover their assets.
- **Staking**: The distribution of staking rewards is managed by a function that the Owner controls, meaning the Owner could assign rewards whenever he wants and whatever amounts he wants. We propose moving forward to decentralize that role or implement its logic on-chain.
- **Censorship**: The Owner has the ability to censor users by blacklisting them. This can be done by removing their access token from the Merkle tree, preventing them from interacting with the protocol, even if they have passed the KYC process. The Owner could also deny users access by refusing to provide the necessary signatures.

To address these concerns, we propose the following solutions:

- **Censorship Mitigation**: Implement a trustless mechanism for managing KYC through *zkEmail* or a similar system. In this model, the KYC provider sends the user an email with their address and a certificate confirming their KYC status. The user then submits this proof to the protocol to claim their access token. If the user is later blacklisted, the Owner must provide a ZKP (Zero-Knowledge Proof) of an email from the KYC provider verifying the user’s blacklisted status.
- **Bridging Security**: Before allowing the `` `rescueAsset` `` function to execute, enforce a requirement to attempt the recovery of any locked transaction IDs. This ensures that users are given the opportunity to reclaim their assets before the Owner can intervene.

- **Elevated Power Oversight**: For operations that grant the Owner elevated privileges, such as unlocking funds, we recommend establishing a security council. This council would be composed of multiple trusted parties, distributing control across a wider group and reducing the risk of a single point of failure.

By implementing these measures, Hinkal can further enhance the security and trustlessness of the protocol, ensuring that the chances of Owner’s powers being exploited are limited.

Basic Concepts and Background

In this section, we will analyze the architecture of the Hinkal protocol and explain some basic concepts and their implementation.

The Hinkal Protocol

The Hinkal Protocol is a privacy-focused protocol that enables users to perform various DeFi interactions, such as deposits, withdrawals, swaps, bridging, and staking, without on-chain traceability. Although transfers are currently disabled, Hinkal is designed to support transfers with a few changes in the code. Note that DeFi interactions are implemented using a set of whitelisted external actions, which are enabled by the contract owner and interact with reputable DeFi protocols.

Hinkal achieves regulatory compliance by requiring users and addresses to undergo a KYC process before interacting with the protocol. Once approved, users generate an access token that is stored in an on-chain, removable Merkle tree, and then they can deposit funds into a shielded address. All subsequent transactions within the Hinkal ecosystem are private, leveraging a large anonymity pool.

Users initiate deposits, swaps, or withdrawals with integrated DEXs through a commitment and nullifier protocol. All operations are based on a UTXO model and are recorded as commitments in an on-chain Merkle tree. Users can then generate a Zero-Knowledge Proof (ZKP) that proves the existence of their commitment within the tree, allowing them to withdraw or swap funds without revealing which specific commitment they own. Nullifiers prevent double-spending by ensuring each commitment is used only once. The ZKPs are generated off-chain, and users can leverage approved relayers to further enhance privacy by using “clean” addresses to receive withdrawals. Note that the output UTXOs and commitments for swaps have to be generated on-chain because the exact token prices are not known beforehand.

Additionally, Hinkal supports cross-chain bridging using Connext, allowing users to bridge tokens or Ether to Hinkal on other blockchains. These cross-chain transactions maintain privacy by involving only shielded addresses that cannot be traced back to the original depositors. To simplify cross-chain interactions, Hinkal supports cross-chain access tokens, eliminating the need for users to repeat the KYC process on each chain.

Hinkal also offers a staking protocol, allowing users to stake tokens in exchange for wrapped hTokens and earn rewards. This staking mechanism benefits the protocol by increasing the size of its anonymity pool, further enhancing user privacy.

Each of these features is described in detail in the following sections.

Merkle Tree and Removable Merkle Tree Implementations

The Merkle tree data structure is a fundamental piece of the protocol: it provides a mechanism to commit to a vector of elements and provide proofs of membership, which are verifiable only by knowing the root hash of the tree. In the protocol, a variant of the standard Merkle tree construction is implemented to handle dynamic insertions and deletions. The tree is assumed to be filled from left to right, with leaf insertions being done at the end of the leaves list.

The interface `MerkleBase.sol` provides a common interface for Merkle trees data structures. One notable function is `rootHashExists()`, which is used extensively in the protocol. The Merkle tree data structure keeps the last `MAX_ROOT_NUMBER` roots (by default this is set to 25) to avoid denial of service by frontrunners, trying to constantly change the Merkle tree root before any transaction is made, invalidating the original transaction.

The `Merkle.sol` contract provides an implementation of `MerkleBase` with a write-only dynamic Merkle tree. As such, there is no need to hold the entire tree in memory, and the contract just holds a mapping `tree` that holds the "right frontier" of node values. The root is computed recursively as such: the hash of one generic internal node is computed as follows

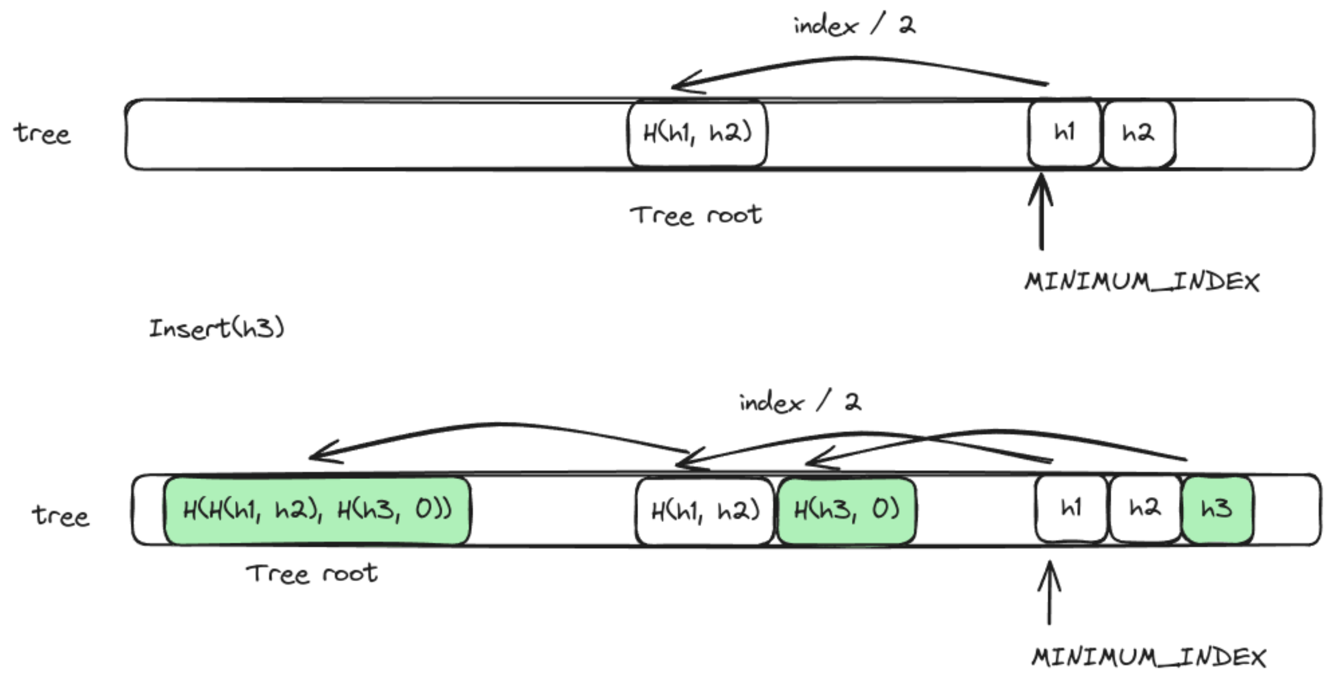
- `Poseidon(left, right)` if the node has both a `left` and `right` children in the tree
- `Poseidon(left, 0)` if the node does not have a right child. In this case, the hash is "bubbled up" the tree, hashing with zero.

The `MerkleRemovable.sol` contract provides another implementation for `MerkleBase`, but is this case also the deletion operation is supported in the tree. This means that it is no longer possible to store only the right frontier, but the full tree mapping has to be stored with all internal nodes. The tree is stored using a binary-heap style mapping to a vector. In particular

- the root is stored at key `0`
- given a node `i`:
  - its left child is stored at key `2*i`
  - its right child is stored at key `2*i + 1`

Leaves are inserted from left to right, starting from key `MINIMUM_INDEX`, which is the index of the leftmost leaf of the tree. To perform an insertion, the tree is updated from the inserted leaf upwards until it reaches a sufficient depth to store all the nodes, computed as the next integer of the logarithm in base 2 of the number of leaves present.

The diagram below illustrates an example of an insertion operation in this data structure.



Access Tokens and User Authentication for Compliance

To ensure regulatory compliance, the Hinkal Protocol employs a KYC process to verify that only compliant users can interact with the protocol. While deposits are restricted to users who have undergone KYC, withdrawals can be directed to any recipient, regardless of KYC status.

The process begins with the user interacting with an authorized KYC provider to obtain a KYC certificate linked to their address. The user then submits the certificate, along with an **access token**, to the Hinkal server. The access token is derived by hashing the user's `` `shieldedPrivateKey` `` and combining it with their `` `PublicKey` `` (which itself is a hash of the `` `shieldedPrivateKey` ``).

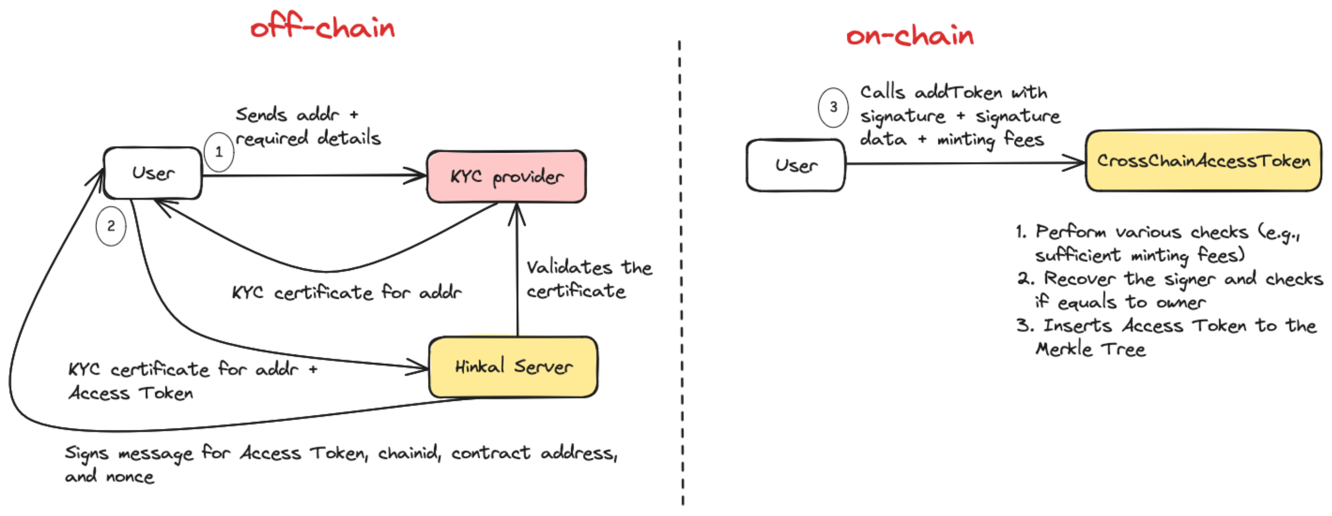
The Hinkal server communicates with the KYC provider to validate the certificate's authenticity. Importantly, mapping an access token to the user's real-world identity requires cooperation between both the KYC provider and the Hinkal server, adding an extra layer of privacy that could be circumvented if required. This scenario can happen if regulatory authorities require from both the Hinkal server and the KYC provider to reveal the info they store in order to remain compliant. Even in this case, compliant users won't be aaffected.

Once validation is successful, the Hinkal server signs a message containing the access token, the chain ID, the address of the Hinkal contract responsible for authentication, and a nonce to prevent replay attacks. The user can then interact with the `` `CrossChainAccessToken` `` contract by submitting this signed message, along with the signature data and minting fees. On-chain verification ensures the signature is valid, and it subsequently adds the access token to the Merkle tree.

The `` `CrossChainAccessToken` `` contract supports cross-chain functionality, allowing users to bridge their access tokens across different blockchains. If a user's access token becomes invalid due to non-compliance, the contract owner has the authority to remove it from the Merkle tree.

After this process is complete, every time the user wants to interact with the Hinkal Protocol, they must provide a zero-knowledge proof (ZKP) that they possess the `shieldedPrivateKey` corresponding to an access token in the Merkle tree. Since only the root hash of the Merkle tree is given as public input, the access token used in the zero-knowledge proof remains concealed, ensuring privacy.

The diagram below illustrates the process of creating and adding access tokens to the Hinkal Protocol.


off-chain
on-chain
Sends addr + required details
1
User
KYC provider
2
KYC certificate for addr
Validates the certificate
KYC certificate for addr + Access Token
Hinkal Server
Signs message for Access Token, chainid, contract address, and nonce
3
Calls addToken with signature + signature data + minting fees
User
CrossChainAccessToken
1. Perform various checks (e.g., sufficient minting fees)
2. Recover the signer and checks if equals to owner
3. Inserts Access Token to the Merkle Tree

Shielded Addresses

Stealth addresses are a key feature across the Hinkal protocol, primarily used to obscure the link between depositors and their funds. Each stealth address is typically used only once, ensuring privacy by breaking the connection between the depositor and the flow of the depositor's transactions. The computation of stealth addresses happens off-chain and is derived from a user's stealth private key (aka `shieldedPrivateKey`). The algorithm for generating these addresses can be found in `StealthAddressCalculator.circom` as well as in the protocol's TypeScript code.

In essence, a stealth address is computed using the user's private key (i.e., the `shieldedPrivateKey`) and a public randomization factor, utilizing elliptic curve operations. In addition to their role in creating/spending UTXO commitments (discussed in the next section), stealth addresses are also used by the protocol's UI to query blockchain events. This allows users to track the assets they own by using the viewing key of the stealth address to identify relevant events emitted on the blockchain.

ZKP UTXOs, Commitments, and Nullifiers

The Hinkal protocol operates on a UTXO (Unspent Transaction Output) model, where each UTXO is structured as follows:

- `erc20Address:` The address of the token the UTXO represents.
- `amount`: The token amount stored in the UTXO.
- `stealthAddress`: The stealth address that owns the UTXO.
- `timestamp`: The time the UTXO was created.

- **`tokenId`**: An internal token ID used by Hinkal contracts.

For each UTXO, a cryptographic commitment is created either off-chain or on-chain (for some outgoing UTXOs, such as in swaps). This commitment is essentially a Poseidon hash of the UTXO's attributes, including the **`amount`**, **`erc20TokenAddress`**, the public key of the stealth address, and the **`timestamp`**.

The nullifier for a commitment is another Poseidon hash, calculated from the nullifier signature and the commitment. The signature itself is the Poseidon hash of the **`shieldedPrivateKey`** and the commitment. The nullifier ensures that each UTXO can only be spent once.

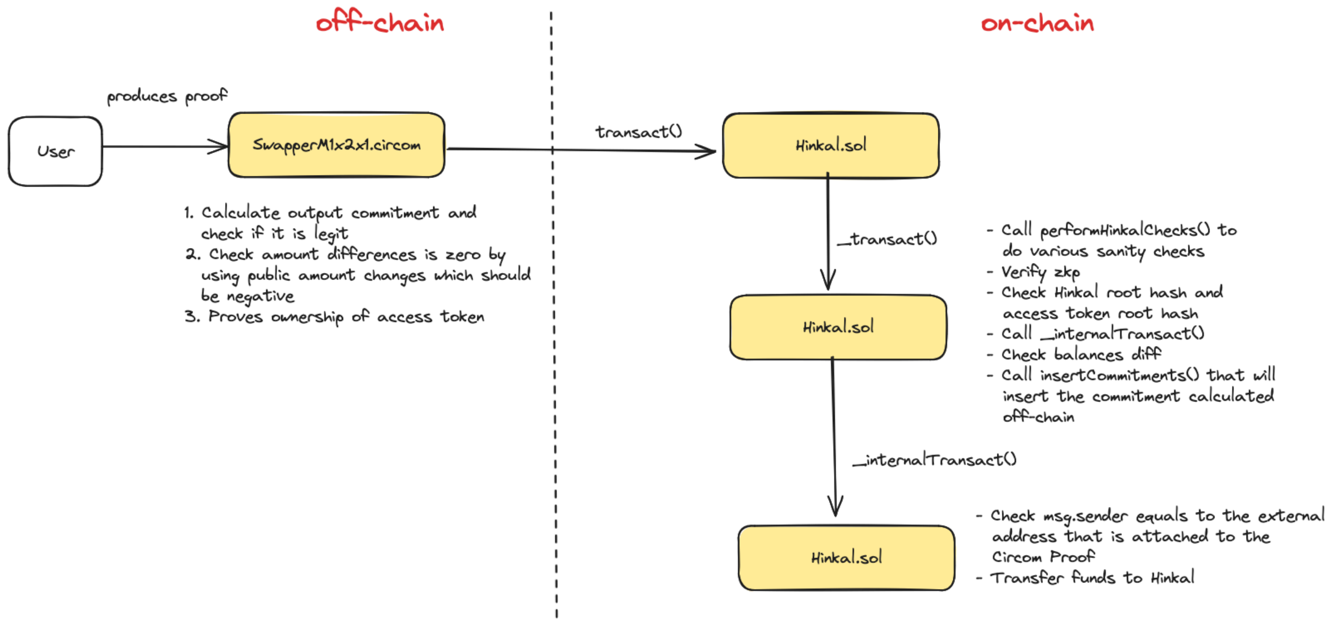
When users wish to spend a UTXO, they must generate a zero-knowledge proof (ZKP) demonstrating that they possess the **`shieldedPrivateKey`** corresponding to a commitment stored in Hinkal's Merkle tree. They must also provide the correct nullifier for the commitment. The Circom and Solidity codebases enforce checks/constraints to ensure the correctness of the spent amounts. Additionally, in Solidity, nullifiers are logged in a data structure to prevent double-spending on commitments.

Depositing and withdrawing in Hinkal

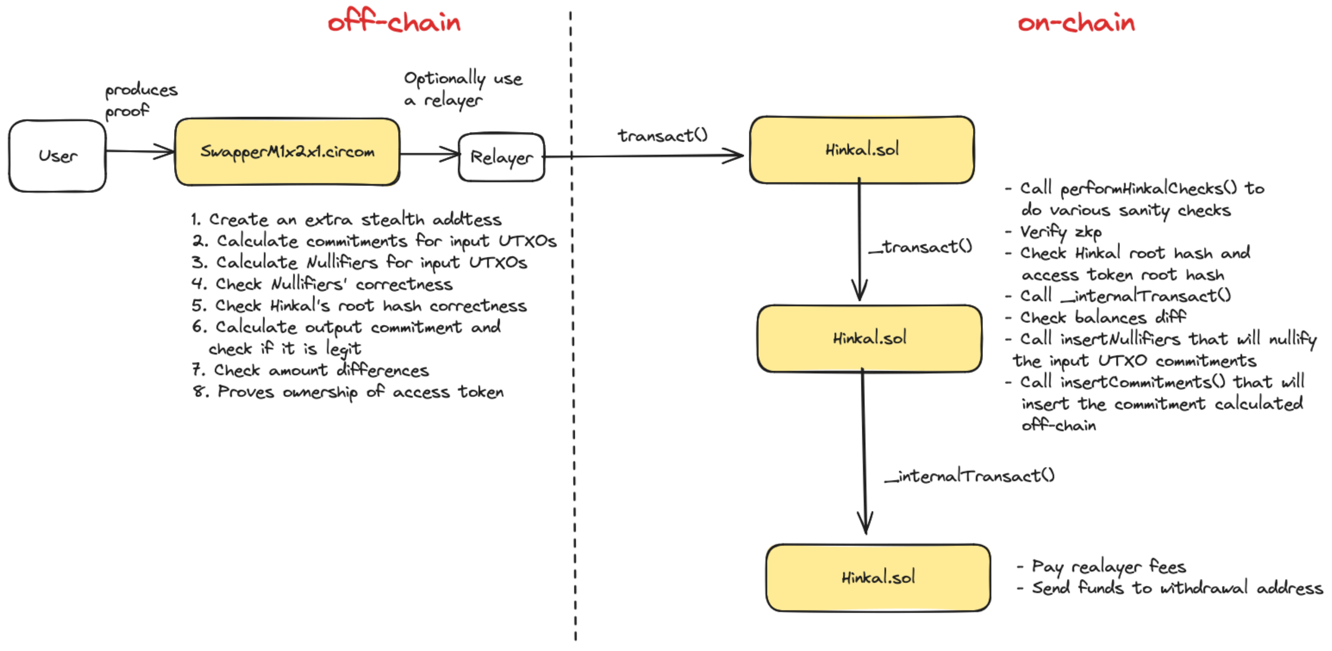
Two of the main operations in the Hinkal Protocol are depositing and withdrawing funds. To interact with Hinkal, users must first create and mint an access token. Once this is completed, users generate a Zero-Knowledge Proof (ZKP) off-chain and submit it on-chain through the Hinkal contract using the **transact** function. The contract performs various checks, including validating the proof, verifying the provided root hashes of both the Hinkal and Access Token Merkle trees, and ensuring that balances across the protocol are accurate.

During this process, the contract nullifies any relevant commitments by tracking the provided nullifiers, ensuring they cannot be reused. It also inserts any newly produced commitments into Hinkal's Merkle tree. For deposits, the user's funds are transferred to the Hinkal contract. In the case of withdrawals, the contract pays any fees to the relayer (if used) before transferring the remaining funds to the designated recipient. Additionally, a calldata hash is submitted alongside the ZKP to prevent frontrunning attacks, ensuring that transaction details—such as the relayer—cannot be altered maliciously.

The figures below illustrate the deposit and withdrawal processes in detail.


off-chain
on-chain
produces proof
User
SwapperM1x2x1.circom
transact()
Hinkal.sol
1. Calculate output commitment and check if it is legit
2. Check amount differences is zero by using public amount changes which should be negative
3. Proves ownership of access token
_transact()
- Call performHinkalChecks() to do various sanity checks
- Verify zkp
- Check Hinkal root hash and access token root hash
- Call _internalTransact()
- Check balances diff
- Call insertCommitments() that will insert the commitment calculated off-chain
Hinkal.sol
_internalTransact()
Hinkal.sol
- Check msg.sender equals to the external address that is attached to the Circom Proof
- Transfer funds to Hinkal

Deposits in Hinkal.


off-chain
on-chain
produces proof
Optionally use a relayer
User
SwapperM1x2x1.circom
Relayer
transact()
Hinkal.sol
1. Create an extra stealth addtess
2. Calculate commitments for input UTXOs
3. Calculate Nullifiers for input UTXOs
4. Check Nullifiers' correctness
5. Check Hinkal's root hash correctness
6. Calculate output commitment and check if it is legit
7. Check amount differences
8. Proves ownership of access token
_transact()
- Call performHinkalChecks() to do various sanity checks
- Verify zkp
- Check Hinkal root hash and access token root hash
- Call _internalTransact()
- Check balances diff
- Call insertNullifiers that will nullify the input UTXO commitments
- Call insertCommitments() that will insert the commitment calculated off-chain
Hinkal.sol
_internalTransact()
Hinkal.sol
- Pay realayer fees
- Send funds to withdrawal address

Withdrawals in Hinkal.

Swapping in Hinkal

The swapping process in Hinkal is similar to deposits and withdrawals, with a few key differences. Instead of using `` `SwapperM1x2x1.circom` ``, users will instantiate the `` `SwapperM` `` circuit with multiple tokens. Additionally, instead of calling the `` `_internalTransact()` `` function, the protocol calls `` `_internalRunExternalAction()` ``, which forwards the swap operation to `` `ExternalActionSwap:runAction()` `` (out of scope for this audit). This function executes the swap and returns the output UTXOs, which cannot be computed off-chain due to the variability of real-time token prices.

Using these UTXOs, the output commitments are then generated on-chain and inserted into Hinkal's Merkle tree. Unlike deposits and withdrawals, where output commitments are created off-chain during ZKP generation, swap-related commitments are computed directly on-chain after the swap is executed.

Bridging in Hinkal

Cross-chain bridging allows users to transfer funds between instances of the Hinkal contract across the chains where it is deployed. First, the user needs to produce two zero-knowledge proofs:

- The first one proves that the user has some funds locked in the source chain in form of UTXOs, and provides nullifier checks for that UTXOs. This proof is produced using one of the **`swapperM*`** circuits, like any other external action.
- The second one is a proof for ownership of the destination stealth address on the destination chain. This proof is produced using the **`receiver`** circuit.

The relayer will initiate a normal external action on the source chain, and the main contract **`Hinkal.sol`** will verify ownership of funds, nullify UTXOs, and call the external action in the **`ConnextAction.sol`** contract (which implements the **`IExternalAction`** interface) using the **`runAction()`** function. The relevant data for the external action is stored in **`externalActionMetadata`**, which is then cast into an **`ActionData`** object.

```
struct ActionData {
    bool isReceive;
    bool unwrapNative;
    uint32 destination;
    uint slippage;
    uint rootHashHinkal;
    uint rootHashAccessToken;
    InternalProof internalProof;
}
```

In particular this structure contains the **`internalProof`** which is the **`receiver`** zero-knowledge proof which will be sent cross-chain and will be verified on the destination chain. The **`ConnextAction.sol`** contract acts as the external action contract for both the bridge send and receive transactions. The two flows are distinguished by the **`isReceive`** flag in the action data structure.

The **`ConnextAction.sol`** contract will, at this point, send a cross-chain invocation using a third-party bridging service called connext. Invocation is done by calling the **`connext.xcall()`** function, which will invoke the **`xReceive()`** function on the **`connextRescueBuffer.sol`** contract deployed on the destination chain.

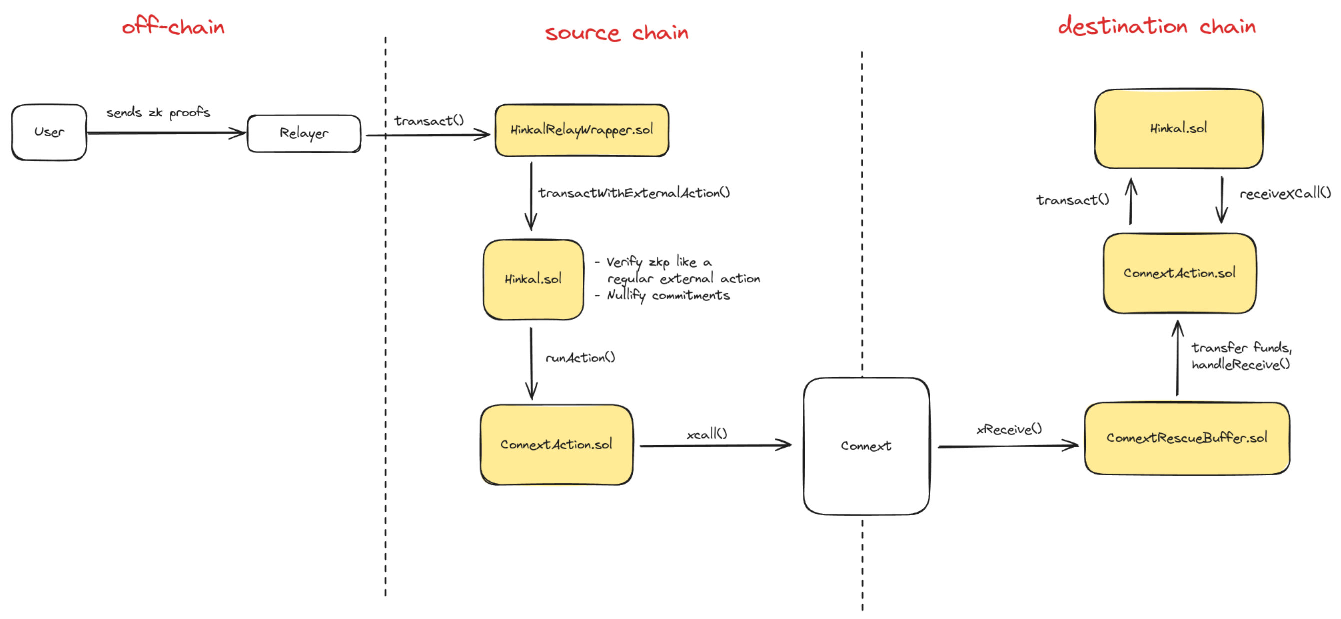
According to the connext documentation developers should use defensive strategies when implementing the **`IXReceive`** interface. Indeed, if a cross-chain receive transaction fails, then funds could get stuck on the receiver contract. The **`connextRescueBuffer.sol`** aims to address this issue by providing a safe buffer for funds. The Hinkal owner can re-try the failed transaction with a different **`callData`** (i.e., calling **`rescue`**) or directly withdraw them (i.e., calling **`rescueAsset`**). The rescue buffer keeps a mapping between the **`transferId`** provided by Connext and some information about the funds, in particular the amount, asset, and the original call data payload.

The contract then calls the `` `transferCall()` `` function: if it succeeds then the mapping is removed, otherwise it is kept and a `` `FailedTransfer` `` event is emitted.

At this point, the rescue buffer contract calls the `` `handleReceive()` `` function in the `` `ConnextAxtion.sol` `` contract (on the destination chain), also transferring the funds. This function basically prepares some input arguments and calls back into the normal `` `Hinkal.sol:transact()` `` transaction flow. The action ID is set to `` `RECEIVE_ONLY_ACTION_ID` ``, and the external action address is set to `` `this` `` (the `` `ConnextAction.sol` `` contract address). Crucially, the inner proof is also passed to the Hinkal contract as the main ZKP to be verified.

The Hinkal contract performs verification of the zero-knowledge proof using the `` `receiver.circom` `` circuit, performs all the standard checks for the transact flow, and calls back the `` `ConnextAction.sol` `` contract as an external action. This call will generate the UTXO and will transfer back the funds from `` `ConnextAction.sol` `` contract to `` `Hinkal.sol` `` contract in the destination chain. The bridge of funds will be completed after Hinkal adds the out commitments in the new chain that are now spendable. Notice that if at any point the reception flow fails (e.g., because the inner proof is not valid), then the funds would still be locked in the rescue buffer.

Below is a diagram of the bridging flow.



Staking in Hinkal

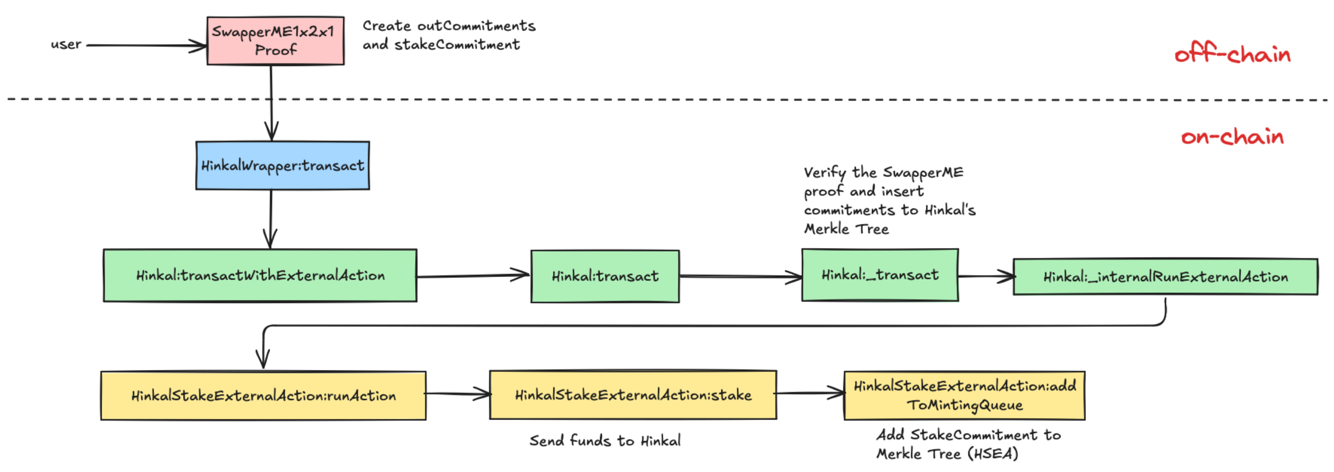
Hinkal's staking feature is designed to reward users who contribute to the protocol's anonymity pool. Staking and deposit operations are managed through the `` `HinkalStakeExternalAction` `` contract, utilizing the `` `SwapperME1x2x1.circom` `` circuit. During a deposit, two commitments are generated: one for the standard Hinkal Merkle tree and another for the Hinkal staking Merkle tree. The first is a regular commitment, as previously described, while the second is an unspendable commitment placed in the staking Merkle tree, which cannot be minted.

Similarly, when users stake their assets, they generate an unspendable deposit commitment in the Hinkal Merkle tree and a spendable commitment in the staking Merkle tree. These commitments are indistinguishable from external observers, enhancing the overall anonymity set of the Hinkal protocol.

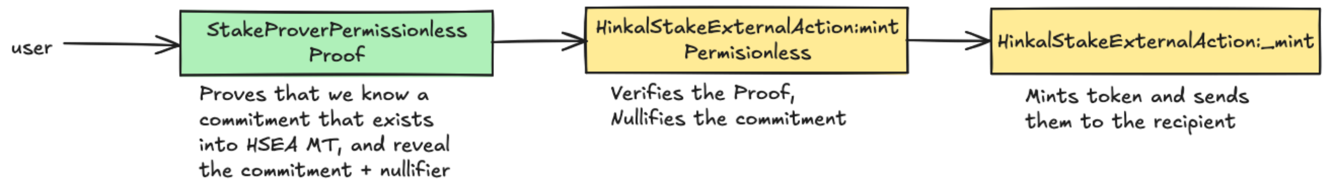
Upon staking, users can mint **hTokens** (wrapped tokens) by spending their staking commitments. These hTokens can be used in DeFi protocols like any other token. Additionally, the Hinkal owner has the ability to distribute rewards to hToken holders. Finally, users can unstake their hTokens, converting them back into regular tokens and transferring them to any desired address.

The figures below illustrate all the operations users can perform within the `` `HinkalStakeExternalAction` `` contract.

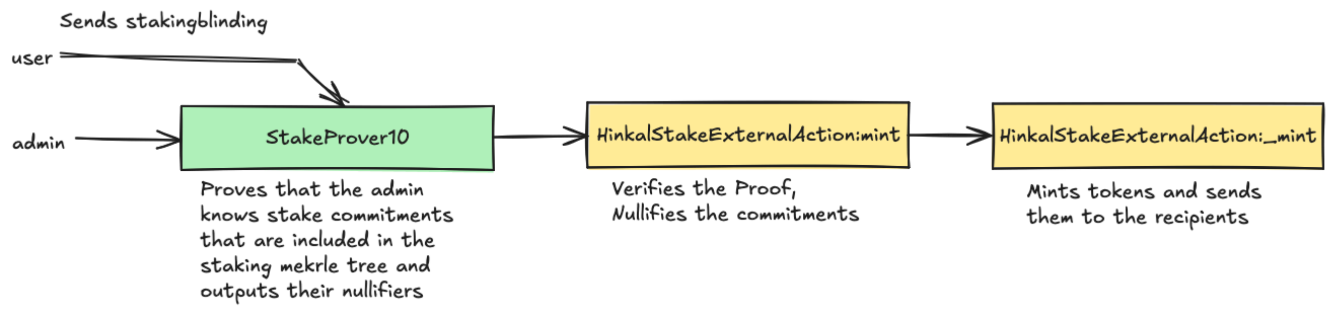
Deposit or Stake


user
SwapperME1x2x1 Proof
Create outCommitments and stakeCommitment
off-chain
on-chain
HinkalWrapper:transact
Hinkal:transactWithExternalAction
Hinkal:transact
Verify the SwapperME proof and insert commitments to Hinkal's Merkle Tree
Hinkal:_transact
Hinkal:_internalRunExternalAction
HinkalStakeExternalAction:runAction
HinkalStakeExternalAction:stake
Send funds to Hinkal
HinkalStakeExternalAction:addToMintingQueue
Add StakeCommitment to Merkle Tree (HSEA)

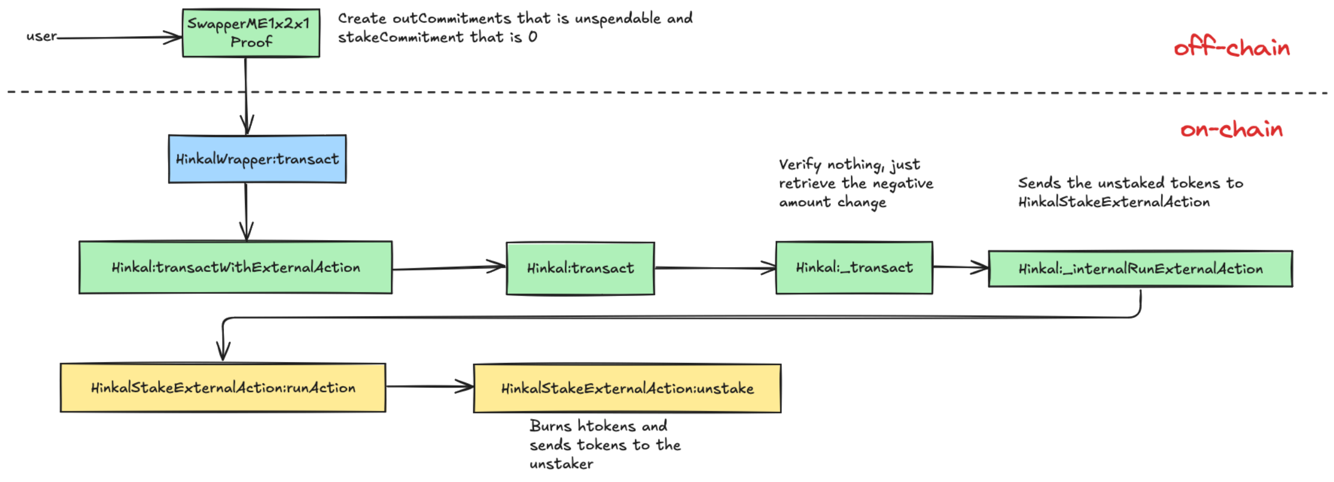
Permisionless Minting Stake


user
StakeProverPermissionless Proof
Proves that we know a commitment that exists into HSEA MT, and reveal the commitment + nullifier
HinkalStakeExternalAction:mint Permisionless
Verifies the Proof, Nullifies the commitment
HinkalStakeExternalAction:_mint
Mints token and sends them to the recipient

Admin Minting Stake


Sends stakingblinding
user
admin
StakeProver10
Proves that the admin knows stake commitments that are included in the staking mekrle tree and outputs their nullifiers
HinkalStakeExternalAction:mint
Verifies the Proof, Nullifies the commitments
HinkalStakeExternalAction:_mint
Mints tokens and sends them to the recipients

Unstake


user
SwapperME1x2x1 Proof
Create outCommitments that is unspendable and stakeCommitment that is 0
off-chain
on-chain
HinkalWrapper:transact
Hinkal:transactWithExternalAction
Hinkal:transact
Verify nothing, just retrieve the negative amount change
Hinkal:_transact
Sends the unstaked tokens to HinkalStakeExternalAction
Hinkal:_internalRunExternalAction
HinkalStakeExternalAction:runAction
HinkalStakeExternalAction:unstake
Burns htokens and sends tokens to the unstaker

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME
00	hardhat/contracts/CrossChainAccessToken.sol	The blacklisting mechanism is not enforced
01	hardhat/contracts/{Hinkal,external-actions/hinkal-staking/HinkalStakeExternalAction}.sol	encryptedOutput and encryptedStakeCommitmentContent are not binding to the verified ZKP proofs.
02	hardhat/contracts/CrossChainAccessToken.sol	accessTokens mapping is not enforced to hold accurate data.
03	libs/circom/MerkleRootCalculator.circom	Merkle tree construction allows crafting a Merkle proof for membership of the intermediate nodes
04	hardhat/contracts/HinkalWrapper.sol	Attacker can get ETH that was accidentally sent to HinkalWrapper
05	libs/circom/SwapperME.circom	SwapperME circuit is unnecessarily complex

# 00 - The blacklisting mechanism is not enforced

● **hardhat/contracts/CrossChainAccessToken.sol**

Medium

Status. Fixed.

Description. The Hinkal protocol uses access tokens to authenticate participants who interact with Hinkal's contracts. The reason why Hinkal requires that authentication mechanism is that its users need to be KYC-ed in order to stay compliant with regulations. The way this process works is as follows:

off-chain steps:

- The user interacts with a KYC provider to get KYC'ed.
- The KYC provider creates a certificate for the user.
- The user proves to the Hinkal server that he has been KYC'ed and sends his `` `AccessToken` ``, which is composed of his `` `shieldedPrivateKey` `` hashed with his `` `PublicKey` `` (which is the hash of his `` `shieldedPrivateKey` ``).
- The server signs a message composed of the user's `` `AccessToken` ``, the `` `blockchain.chainid` ``, the Hinkal's protocol `` `CrossChainAccessToken` `` contract's `` `address` ``, and a `` `nonce` ``.

on-chain steps:

- The user calls the `` `addToken` `` function of `` `CrossChainAccessToken` `` with the following data:

```
struct SignatureData {
    uint8 v;
    bytes32 r;
    bytes32 s;
    uint256 accessKey;
    uint256 nonce;
}
```

- The `` `addToken` `` function does the following checks:

```
require(msg.value >= mintingFee, "minting fee not covered");
require(
    !blacklistAddresses[msg.sender],
    "Address has been blacklisted"
);
require(!usedNonces[signatureData.nonce], "nonce already used");

bytes32 ethSignedMessageHash = getEthSignedMessageHash(
    signatureData.accessKey,
    block.chainid,
    address(this),
    signatureData.nonce
);
address signer = recoverSigner(ethSignedMessageHash, signatureData);
require(
    signer == owner(),
    "Signature must be signed by the owner of the contract"
);
```

- Then it calls the function `insertAccessToken` that will (i) insert the `AccessToken` in the Merkle Tree that stores the access tokens for compliant users, and will also add the `AccessToken`/`msg.sender` pair in the `accessTokens` mapping to map access tokens to addresses.

The main issue in that process is that a malicious user can circumvent the blacklisting mechanism by just calling the `addToken` function by another address.

Impact. This attack will lead to the Hinkal protocol being non-compliant by allowing non-blacklisted addresses to interact with the protocol.

Recommendation. We recommend adding the `msg.sender` address in the signature to bind addresses to access tokens. By doing so, if the user uses another address to call the `addToken` function, then the signature verification will fail.

- Client comment: We only use `blacklistAccessKey` as a blacklisting mechanism. `blacklistAddress` is a legacy function that we do not currently use. Given that we called `blacklistAccessKey` for a specific `accessKey`, the user who holds this access key will not be able to mint the access token again because he will not be able to use the same nonce.

Client Response. Fixed in commit cec95d4f443c6d5faed1d6d066c88bf92031f7d9. We added msg.sender in the signature to bind addresses to access tokens.

# 01 - encryptedOutput and encryptedStakeCommitmentContent are not binding to the verified ZKP proofs.

● **hardhat/contracts/{Hinkal,external-actions/hinkal-staking/HinkalStakeExternalAction}.sol**

Medium

Status. Acknowledged.

Description. All transactions in the Hinkal protocol are routed through `` `Hinkal.sol` ``. The process is the following: First, an off-chain ZKP proof is generated through the Circom circuits. The proof is then verified on-chain using the Solidity smart contracts, and some additional logic follows.

When users transact with Hinkal, they are required to attach some additional data beyond the ZKP and the public inputs and outputs of the proof. Those data are the following:

```
externalActionMetadata -> calldata used with external actions stores information regarding
swaps, route paths, and whether to perform staking / unstaking.
externalActionId -> the id of the external action to use.
externalActionAddress -> the address of the external action to use.
flatFees -> the flat fees the client is willing to pay to the relayer
relay -> the address of the relayer
publicSignalCount -> used to determine which verifier to use.
encryptedOutputs -> encrypted data used to store commitment information.
```

Then the hash of those data is also provided as a public input in the circuit as the signal `` `calldataHash` ``. Then the hash is also computed in the solidity code as follows.

```
function getHashedCalldata(
        CircomData calldata circomData
    ) internal pure returns (uint256) {
    return
        uint256(
            keccak256(
                abi.encode(
                    circomData.publicSignalCount,
                    circomData.relay,
                    circomData.externalAddress,
                    circomData.externalActionId,
                    circomData.externalActionMetadata,
                    circomData.hookData,
                    circomData.encryptedOutputs,
                    circomData.flatFees
                )
            )
        ) % CIRCOM_P;
}
```

Those data are used mainly to prevent front-running issues and to bind the relay, hooks, and fees to the submitted proof computed off-chain by a user.

Further, we understand that the **`encryptedOutputs`** are used for UI/UX purposes. For example, a user can query them to see which of those outputs have not been spent and are owned by a shielded address that they are owning. Nevertheless, those data are not bound to the circuit computations. That could lead to the attack scenario where an adversarial user that is participating in the Hinkal protocol computes off-chain a valid proof and then attaches to the proof the hash of the **`circomData`** where the **`encryptedOutputs`** are not the ones that supposedly should be. Note that the hash will be the same as the one used off-chain for producing the proof. Although all the checks will pass, the emitted **`circomData.encryptedOutputs`** won't be bound to the proof.

Note that an identical issue is happening with **`encryptedStakeCommitmentContent`** in **`HinkalStakeExternalAction`**.

Impact. The attacker could achieve two goals. Firstly, it could trick another user into thinking that he has sent him some tokens because he can attach arbitrary UTXOs in the **`encryptedOutputs`**. Secondly, he can make the detriment any code that depends on the **`encryptedOutputs`** unusable by polluting **`encryptedOutputs`** of valid proofs with many incorrect UTXOs, which would result in users being unable to keep track of their real assets. Note that this issue could not exploit any of the functionality that is in the scope of this audit, i.e., it could not lead to double-spending, etc. Finally, note that users could actually check in the Merkle Tree if there are commitments for the UTXOs of **`encryptedOutputs`**.

Recommendation. We recommend to bind **`encryptedOutputs`** in the Circom circuits. Otherwise, clearly document the above scenario, and implement tools for users to easily check the validity of **`encryptedOutputs`**.

Client Response. We document the attack scenario in Risk Section, Point 6 (https://hinkal-team.gitbook.io/hinkal/hinkal/risks), and indicated a tooling for users to easily check the validity of **`encryptedOutputs`**.

# 02 - accessTokens mapping is not enforced to hold accurate data.

● hardhat/contracts/CrossChainAccessToken.sol

Low

Status. Fixed.

Description. In the code mentioned above ("The blacklisting mechanism is not enforced"), for the same reason as above, the `` `accessTokens` `` mapping does not hold the supposed data. In our understanding, this map holds the data of which `` `accessToken` `` is owned by which address. A user can call the `` `addToken` `` function by another address, and then the access tokens mapping will not hold the correct data.

Impact. This mapping is used in a required statement in `` `_migrateAccessToken` `` function:

```
require(
            accessTokens[accessKey] == msg.sender,
            "User does not own this access token"
);
```

Nevertheless, this should not lead to major issues as non-KYCed users cannot interact with the protocol as they do not have an access token.

Recommendation. It is crucial to add the `` `msg.sender` `` address in the signature to bind addresses to access tokens. This will prevent a user from using another address to call the `` `addToken` `` function, as the signature verification will fail.

Client Response. Fixed in commit cec95d4f443c6d5faed1d6d066c88bf92031f7d9. We added msg.sender in the signature to bind addresses to access tokens.

# 03 - Merkle tree construction allows crafting a Merkle proof for membership of the intermediate nodes

● libs/circom/MerkleRootCalculator.circom

Low

Status. Fixed.

Description. The Circom template `MerkleRootCalculator` aims at reconstructing the Merkle tree root given a leaf and all the required intermediate hashes (i.e., siblings). The gadget computes this function (translated into pseudocode)

```
def MerkleRootCalculator(inCommitment,
                         commitmentSiblings[inputs],
                         commitmentSiblingSides[inputs]):
    hashes[0] = inCommitment
    for i in range(inputs):
        assert(commitmentSiblingSides[i] in [0, 1])

        L, R = hashes[i], commitmentSiblings[i]
        if commitmentSiblingSides[i] == 1:
            R, L = L, R
        hashes[i+1] = Poseidon(L, R)

        if commitmentSiblings[i] == 0:
            hashesProcessed[i] = 0
        else:
            hashesProcessed[i] = hashes[i+1]

    rootHashes[0] = hashes[0]
    for i in range(1, inputs+1):
        if hashesProcessed[i-1] == 0:
            rootHashes[i] = rootHashes[i-1]
        else:
            rootHashes[i] = hashesProcessed[i-1]
    return rootHashes[inputs]
```

The logic of the function allows to craft a Merkle proof for every intermediate node in the Merkle tree, thus proving membership of non-member values. The root cause is the fact that the root hash alone does not provide any information to the verifier about the current height of the tree. For example, take the Merkle tree formed by just two levels, where the leaves are, for simplicity, `1`, `2`, `3`, and `4`.

```
Tree leaves:
- 1
- 2
- 3
- 4

Second level:
- 7853200120776062878684798364095072458815029376092732009249414926327459813530
- 14763215145315200506921711489642608356394854266165572616578112107564877678998

Root
- 3330844108758711782672220159612173083623710937399719017074673646455206473965
```

We should be allowed to prove only membership of the leaves; for example, an intended usage would be proving membership of the value `1` by providing the following input.

```
{
    "inCommitment": "1",
    "commitmentSiblings": ["2",
"14763215145315200506921711489642608356394854266165572616578112107564877678998"],
    "commitmentSiblingSides": ["0", "0"]
}
```

We can, however, prove the membership of the root by providing a Merkle proof composed of only zeros.

```
{
    "inCommitment":
"3330844108758711782672220159612173083623710937399719017074673646455206473965",
    "commitmentSiblings": ["0", "0"],
    "commitmentSiblingSides": ["0", "0"]
}
```

Additionally, we can also prove membership of any intermediate node values by providing "half" of the Merkle proof and padding on the right with zeros.

```
{
    "inCommitment":
"7853200120776062878684798364095072458815029376092732009249414926327459813530",
    "commitmentSiblings":
["14763215145315200506921711489642608356394854266165572616578112107564877678998", "0"],
    "commitmentSiblingSides": ["0", "0"]
}
```

In all three cases, the `MerkleRootCalculator` circuit gives output the correct Merkle tree root hash value

```
rootHash = 3330844108758711782672220159612173083623710937399719017074673646455206473965
```

The `AccessTokenChecker` circuit should check that the prover knows an opening to an access token commitment stored in a Merkle tree on-chain. The pseudocode of the checker is as follows.

```
def AccessTokenChecker(shieldedPrivateKey,
                       publicKey,
                       rootHashAccessToken,
                       accessTokenSiblings[treeDepth],
                       accessTokenSiblingSides[treeDepth],
                       isStakeOrUnstake):

    # compute the token commitment from the opening
    calcAccessToken = Poseidon(shieldedPrivateKey, publicKey)

    # compute the Merkle root from commitment and the proof
    calcAccessRootHash = MerkleRootCalculator(
                            inCommitment,
                            commitmentSiblings[inputs],
                            commitmentSiblingSides[inputs])

    # check that the computed root is equal to the real one
    if isStakeOrUnstake == 0:
        assert(calcAccessRootHash == rootHashAccessToken)
```

As a consequence of the previous issue, it is possible to provide valid Merkle proof even without knowing the opening to a commitment in the Merkle tree. In this scenario, the attacker observes the AccesToken Merkle tree on-chain; suppose it's the one above where the token commitments are `1`, `2`, `3`, and `4`. Since the `MerkleRootCalculator` allows for proving membership of internal nodes of the tree, the tree itself provides valid openings for the access token.

A valid input for the `AccessTokenChecker` template is as follows.

```
{
    "shieldedPrivateKey":
"7853200120776062878684798364095072458815029376092732009249414926327459813530",
    "publicKey":
"14763215145315200506921711489642608356394854266165572616578112107564877678998",
    "rootHashAccessToken":
"3330844108758711782672220159612173083623710937399719017074673646455206473965",
    "accessTokenSiblings": ["0", "0"],
    "accessTokenSiblingSides": ["0", "0"],
    "isStakeOrUnstake": "0"
}
```

Notice that every value is either fixed or derived from the (public) Merkle tree on-chain. Notice also that this is enabled by the token commitment being the result of hashing two field elements with the Poseidon hash, which is exactly equal to the Merkle tree hash computation at every node.

Finally, this is vulnerable because there is no constraint in the circuit such that the public key is indeed derived from the private key. Hence, if there is such a check before calling this circuit (which is the case for Receiver and SwapperM), then this vulnerability is not exploitable. This is because in order to exploit it, we need to find intermediate hashes that the one is the Poseidon hash of the other.

Impact. This issue allows the construction of valid Merkle proofs for intermediate nodes present in the tree. We did not find any instance of this being exploitable in practice: **`` `AccessTokenChecker` ``** is always used by having the **`` `publicKey` ``** generated from the **`` `shieldedPrivateKey` ``**. However, we recommend fixing this issue as the **`` `AccessTokenChecker` ``** could be reused in other contexts. If not fixed, we recommended adding a warning of the implicit constraint that **`` `publicKey` ``** must be derived from **`` `shieldedPrivateKey` ``**.

Recommendation. Ensure that the Merkle root calculator computes correctly every hash at each level. This could be implemented by also providing to the Merkle root calculator an additional input specifying how many leaves are currently in the tree. The circuit should then check the length of the proof to be consistent with this value.

Client Response. Fixed in commit `b22861f9e40725cdad96bee5d2b926abd928a90e`. We enforced the constraint that public key should be a derivation from private key in AccessTokenChecker circuit.

# 04 - Attacker can get ETH that was accidentally sent to HinkalWrapper

● hardhat/contracts/HinkalWrapper.sol

Informational

Status. Fixed.

Description. Suppose a user goes into unstake flow and he accidentally sends 1 ETH to `` `HinkalWrapper` `` as `` `msg.value` ``. This message value is stored in `` `HinkalWrapper` `` as a state variable named `` `value` ``. Since unstake flow does not use this ETH balance, it will stay one ether after the user transaction gets executed.

Since `` `HinkalWrapper.getETH()` `` is permissionless, an attacker can call it and steal the user's ETH:

```
function getETH() public {
    uint256 oldValue = value;
    value = 0;
    transferETH(msg.sender, oldValue);
}
```

Impact. If the user mistakenly sends some ETH to `` `HinkalWrapper` ``, an attacker can get them.

Recommendation. Check who is `` `msg.sender` `` in the code:

```
// Add hinkalStakeExternalAction info via constructor

function getETH() public {
    require(msg.sender == sender || msg.sender == address(hinkalStakeExternalAction),
"unexpected caller");
    uint256 oldValue = value;
    value = 0;
    transferETH(msg.sender, oldValue);
}
```

Client Response. Fixed it in commit a0ea50b328d101ca7e86910c274ca505ced5b1dc. We added Roles to HinkalWrapper, so we restrict entities who can call getETH.

# 05 - SwapperME circuit is unnecessarily complex

● libs/circom/SwapperME.circom

Informational

Status. Fixed.

Description. The `SwapperME` circuit produces proofs for use with Hinkal staking features. Specifically, this circuit is responsible for creating and verifying proofs for depositing, staking, and unstaking in the staking extension of the Hinkal protocol.

The circuit has some unnecessary complexity as its only instantiation is the following.

```
component main {public [rootHashHinkal, rootHashAccessToken, outTimeStamp,
    extraRandomization, amountChanges, erc20TokenAddresses,
    calldataHash, inNullifiers, outCommitments ]} = SwapperME(1,2,1,25);
```

Hence, the template variables `tokenCount`, `inputCount`, and `outputCount`, can be removed. Further, the `onChainCreation[tokenCount]` signal is enforced to always be `0` (i.e. `false`). This is enforced in `StakeInputChecker` template by using the following constraint.

```
isOnChainCreationZero[i] === 1;
```

We also recommend removing this signal and simplifying all the affected templates.

Finally, the **unstake** process does not require a ZKP to be verified. In the `SwapperME` circuit, all the important checks are being "disabled", and it essentially just produces a nullifier that does not nullify any **valid** commitment. It also just checks the `amountChanges` correctness, but the user just provides this as a public input, so this check can also be applied in plain Solidity code. We recommend simplifying the circuit by removing the unstake functionality and handling unstake via Solidity. Note that if unstake is being removed, users will still have to go over the Hinkal contract to unstake, and they should **not** have the ability to include nullifiers as this can lead to other issues in case of frontrunning.

Impact. We did not find any vulnerabilities resulting from the complexity of the `SwapperME` circuit.

Recommendation. We recommend refactoring the `SwapperME` circuit to reduce its complexity and reduce the possibility of introducing subtle issues in the future.

Client Response. Fixed it in commit e25c034f232849499308b6e2cd24ab63fc4cc9ec. We simplified SwapperME by removing onChainCreation variable. We did not touch tokenCount, inputCount, and outputCount since we want to have possibility to extend circuits to multi-token staking.