hashlock.

Security Audit

OpenZK

January 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

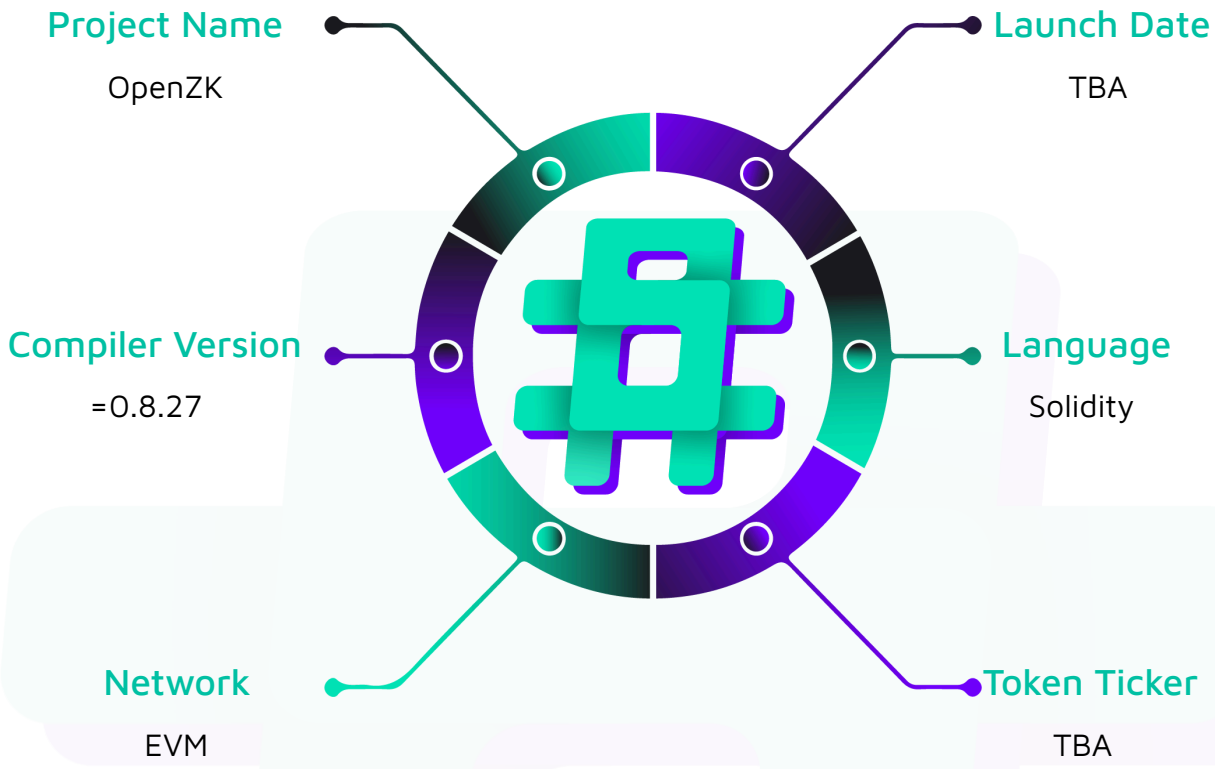
The OpenZK team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

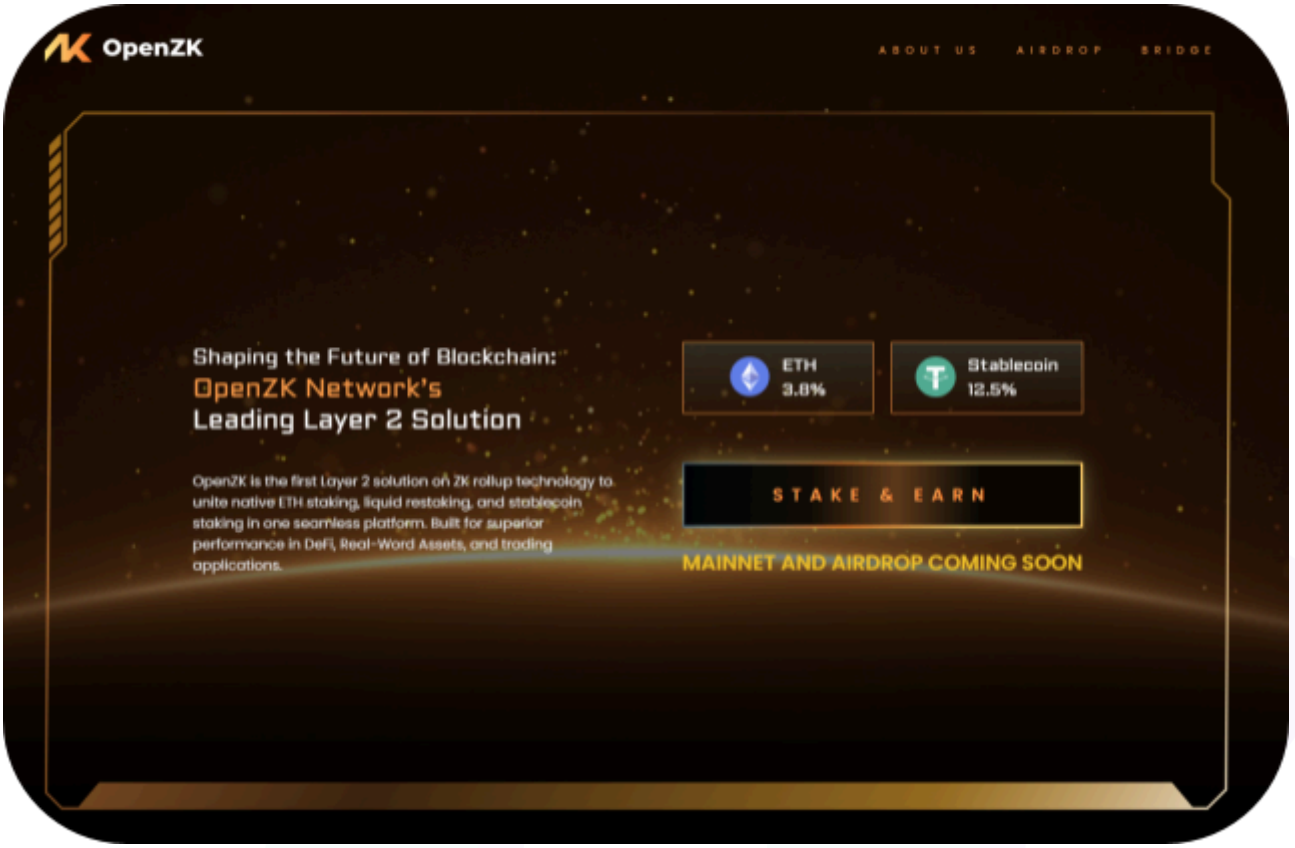
OpenZK is an innovative Layer 2 (L2) network project that leverages Zero-Knowledge (ZK) Rollup technology to enhance Ethereum's scalability and transaction efficiency while maintaining high security. By integrating native ETH staking and restaking capabilities, OpenZK not only bolsters scalability but also creates new revenue streams for users. The platform's leadership includes co-founder Dave Sandor, formerly an Executive Director at Goldman Sachs Asia-Pacific, co-founder and CTO Lucas Cullen, an early contributor to Ethereum's development, and also co-founder Jenna Wayne. OpenZK distinguishes itself with a Dual Gas Fee Mechanism, allowing users to pay gas fees with its native and protocol tokens, enhancing network flexibility and creating sustained demand for its protocol token.

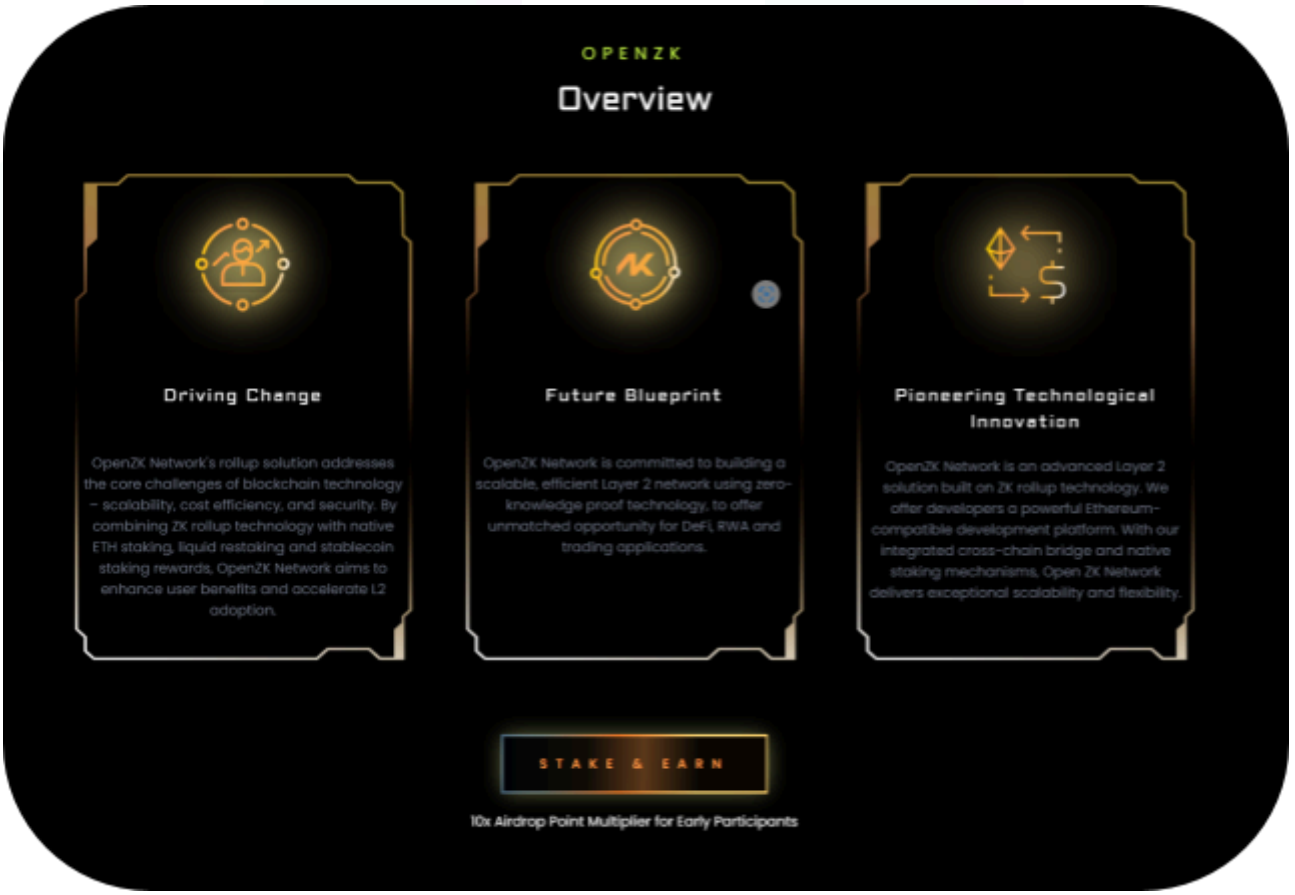
Project Name: OpenZK
Project Type: Defi
Compiler Version: =0.8.27
Website: www.openzk.net
Logo:

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the OpenZK project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	OpenZK Smart Contracts
Platform	**EVM / Solidity**
Audit Date	**January, 2025**
Contract 1	LiquidityManager.sol
Contract 1 MD5 Hash	a75e6a86a0a63a396e2bd1dd3c3a3dfe
Contract 2	RocketPoolVault.sol
Contract 2 MD5 Hash	bf997a149984cabb2c9c1b01f072cea1
Contract 3	RocketPoolVaultBase.sol
Contract 3 MD5 Hash	5d312172298950da4f36385298a9b1bd
Contract 4	UniswapOracle.sol
Contract 4 MD5 Hash	2132eb5fdf597f405b381afde9f26a3b
Contract 5	Token.sol
Contract 5 MD5 Hash	b2497fe1c19c2ca77795d828a893a2fc
Contract 6	ozETH.sol
Contract 6 MD5 Hash	72841b77a6954e150613944fa807ce93
Contract 7	PrivateSale.sol
Contract 7 MD5 Hash	a2aaa28228a598dfd5b91082dfeb5578
Contract 8	PrivateSaleManagement.sol
Contract 8 MD5 Hash	fed4c9adb09447ad4042b8e77449de2d
Contract 9	VestingManager.sol

Contract 9 MD5 Hash	a67b9398877a5faaa06c7ac4d53b0bc7
Contract 10	EigenLayerRETHVault.sol
Contract 10 MD5 Hash	af733d40c39fae6a4acc59fa939bdab4
Git Commit Hash (zkstaking)	530eca5444b3c9a3e31e05b323a7a575a2c45565
Git Commit Hash (presale)	09ad07f4ffccfb6c4dc245e0447df81c8a4ab75c

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts. We initially identified some significant vulnerabilities that have since been addressed.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The general security overview is presented in the Standardised Checks section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

2 High severity vulnerabilities

2 Low severity vulnerabilities

2 Gas Optimisations

6 QAs

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour

PrivateSale.sol
- Allows users to:
  - Deposit ETH/USDT
- Allows DEFAULT_ADMIN_ROLE to:
  - Set an ozkPerUsdQuote value
  - Set the start and end date for the private sale
- Allows FUNDS_OWNER_ROLE to:
  - Withdraw ETH
  - Withdraw any ERC20 tokens
- Allows WHITELIST_MANAGER_ROLE to:
  - Add/Remove a list of addresses from the whitelist
	Contract achieves this functionality.

VestingManager.sol
- Allows users to:
  - Claim their vested wallets
- Allows DEFAULT_ADMIN_ROLE to:
  - Set the start date, the releasedOnStart, and the vestedDays
	Contract achieves this functionality.

LiquidityManager.sol
- Allows users to:
  - Stake/Unstake funds
- Allows owner to:
  - Set the underlying and oracle addresses
  - Add/Remove vaults
  - Rebalance
  - Toggle pause
	Contract achieves this functionality.

RocketPoolVault.sol
- Allows users to:
  - Deposit/Withdraw funds
- Allows owner to:
  - Set the uniswap and balancer portions
  - Set the vesting period
  - Emergency withdraw
	Contract achieves this functionality.

ozETH.sol
- Allows users to:
  - Transfer tokens
- Allows MINTER_ROLE to:
  - Mint/Burn tokens
	Contract achieves this functionality.

Code Quality

This audit scope involves the smart contracts of the OpenZK project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was required.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the OpenZK project smart contract code in the form of Github access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] RocketPoolVaultBase#_deposit - shares calculation based on incorrect total value

Description

The `_deposit` function calculates the `shares` value (the amount of vault tokens to be minted) based on the `total` value (the total amount of assets users deposit) and the contract allows users to deposit ETH or rETH as assets.

Even though the values of ETH and rETH are different, the `shares` value is calculated based on only the amount of assets to be deposited, regardless of which asset the user deposits.

Vulnerability Details

The `total` value is calculated by adding `assets` value (the amount of rETH) to `msg.value` value (the amount of ETH).

```
 function _deposit(
        uint256 assets,
        address receiver
) internal returns (uint256 shares) {
        uint256 total = assets + msg.value;
        shares = _previewAssetsToShares(total);

        // swap ETH to rETH
        _swapToUnderlying(msg.value);

        if (assets > 0) {
            // Transfer the assets to the vault
            IERC20(_underlying).transferFrom(msg.sender, _self, assets);
        }
```

```
        _mint(receiver, shares);
}
```

Even though the values of ETH and rETH are different, if users deposit the same amount of ETH and rETH, they will get the same amount of vault tokens.

Proof of Concept

An example of a test that simulates calculating the amounts of vault tokens to be minted when depositing the same amount of ETH and rETH.

```
it("The amounts of vault tokens are the same when depositing ETH and rETH", async () => {
      const { vault, john, reth } = await loadFixture(deployFixture);
      const depositAmount = hre.ethers.parseEther("1");
      const sharesMintedFromETH = await vault
        .connect(john)
        .deposit.staticCall(0, john.address, { value: depositAmount });
      await reth.connect(john).approve(vault.getAddress(), depositAmount);
      const sharesMintedFromRETH = await vault
        .connect(john)
        .deposit.staticCall(depositAmount, john.address);
      expect(sharesMintedFromETH).to.equal(sharesMintedFromRETH);
});
```

Impact

If the prices of ETH and rETH differ significantly, unfair results may occur.

Recommendation

Calculate the amount of vault tokens based on the amount of rETH swapped from ETH when users deposit ETH.

```
function _deposit(
        uint256 assets,
        address receiver
) internal returns (uint256 shares) {
        uint256 total = assets;
        // swap ETH to rETH
        total += _swapToUnderlying(msg.value);
```

```
        shares = _previewAssetsToShares(total);

        if (assets > 0) {
            // Transfer the assets to the vault
            IERC20(_underlying).transferFrom(msg.sender, _self, assets);
        }

        _mint(receiver, shares);
}
```

Status

Resolved

[H-02] RocketPoolVaultBase#_convertAssetsToShares - The function always returns 0

Description

The `_convertAssetsToShares` function is intended to return the amount of shares calculated from the amount of assets to be deposited.

However, the return variable (`shares`) is not updated in the function and its value is always 0.

Vulnerability Details

Since the `shares` value is not updated in the function, it always keeps 0 and the function always returns 0.

```
function _convertAssetsToShares(uint256 assets, address receiver) internal returns (uint256 shares) {
        uint256 total = _swapToUnderlying(msg.value);
        if (assets > 0) {
            IERC20(_underlying).transferFrom(msg.sender, receiver, assets);
            total += assets;
        }
        _mint(receiver, shares);
}
```

Impact

When this internal function is called from another external function, the depositor always gets 0 shares and loses his funds.

Recommendation

Consider removing the function if it's not called from any external functions.

Status

Resolved

Low

[L-01] RocketPoolVaultBase#_withdraw, _emergencyWithdraw - Deprecated Ether transfer function

Description

The above functions use the `.transfer()` function to send the native coins.

However, the Istanbul update made some changes to the EVM, which made the `.transfer()` function deprecated for the ETH transfer.

Recommendation

Use `.call()` function to send ether instead of `.transfer()` function and check the return value.

Status

Resolved

[L-02] LiquidityManager#setUnderlying, setOracle, RocketPoolVault#setVestingPeriod - Lack of emitting events

Description

The above functions are missing events when key storages are updated.

Recommendation

Add relevant events based on the variables to be updated.

Status

Resolved

Gas

[G-01] LiquidityManager - State variables that could be made immutable or constant

Description

The `SCALE` variable is assigned with a hardcoded value when the contract is deployed and it's not updated anymore.

The `_vesting_period` variable is only updated in the constructor.

Recommendation

Make the `SCALE` variable constant and the `_vesting_period` variable immutable.

Status

Resolved

[G-02] RocketPoolVaultBase#_setWeight - Could emit memory variables

Description

The `_setWeight` function emits the updated uniswap and balancer portion values at the end of the function.

However, the function reads the `uniswapPortion` and `balancerPortion` variables from storage to emit even if the `_uniswapPortion` and `_balancerPortion` memory variables exist.

Recommendation

Consider emitting the `_uniswapPortion` and `_balancerPortion` memory variables.

Status

Resolved

QA

[Q-01] RocketPoolVaultBase#_deposit, _convertAssetsToShares, _emergencyWithdraw - Lack of a check for the return value on the token transfer

Description

The above functions are missing a check for the return value on the `_underlying`'s transfer.

Recommendation

Add a check if the return value on the token transfer is true or use the `safeTransferFrom` function from Openzeppelin.

Status

Resolved

[Q-02] PrivateSale - Unused import

Description

The `PrivateSale` contract imports `IQuoterV2` but it's not used in the contract.

Recommendation

Remove the unused import.

Status

Resolved

[Q-03] IVestingManager - Unused events

Description

The `IVestingManager` interface declares `DepositUSD`, `Vested`, and `Withdraw` events but they are not used in the contract.

Recommendation

Remove the unused events.

Status

Resolved

[Q-04] LiquidityManager, RocketPoolVault, RocketPoolVaultBase, Token, UniswapOracle - Floating pragma

Description

The contracts use `pragma solidity ^0.8.27`.

This can allow the contracts to be deployed with the wrong pragma version which is different from the one the contracts were tested with.

Recommendation

Lock the pragma version.

Status

Resolved

[Q-05] RocketPoolVaultBase#_previewAssetsToShares, _previewSharesToAssets, _queueWithdraw - Console logs still exist

Description

The above functions still have console logs.

Having such logs in production could reduce the code quality.

Recommendation

Remove the hardhat console import and logs in the contract.

Status

Resolved

[Q-06] All contracts - Confusing readability by mixing uint and uint256

Description

The contracts are reducing the readability by using both `uint` and `uint256` keywords in the same contracts.

Recommendation

Unify it as either `uint` or `uint256`.

Status

Resolved

Centralisation

The OpenZK project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the OpenZK project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au