BlockÂpex

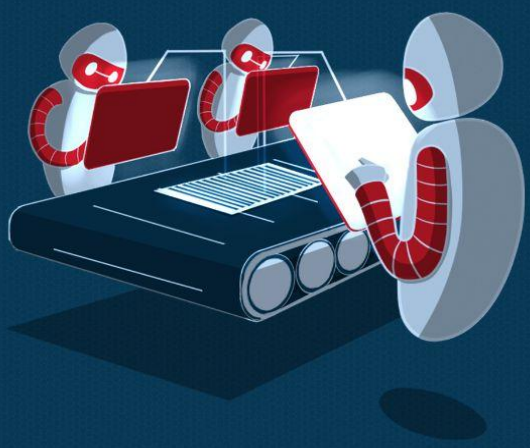
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



PREFACE

Objectives

This document aims to highlight any identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and the client's intellectual property. It also includes information on potential risks and the processes involved in mitigating/exploiting the risks mentioned below. The usage of the information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly to aid our growing blockchain community; at the client's discretion.

Key understandings

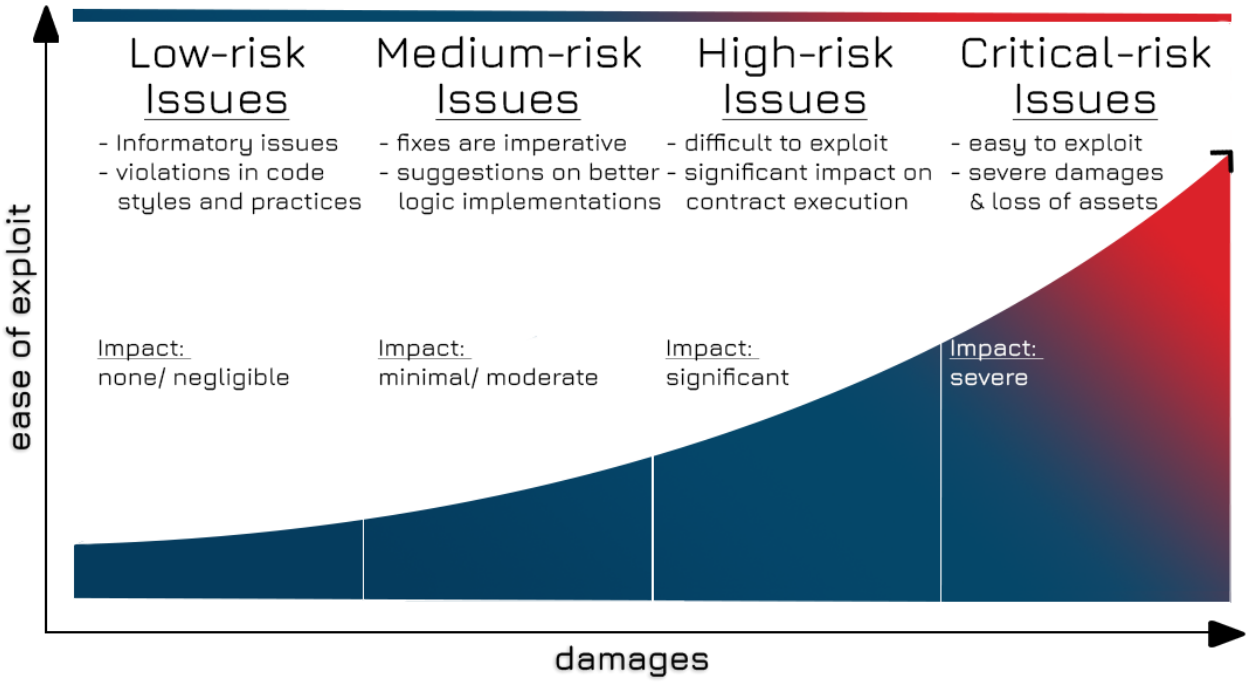


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by ZeroLiquid (Client) to conduct a Smart Contract Audit/ Code Review. This document presents the findings of our analysis, which started on 11th July '2023

Name
ZeroLiquid Protocol
Auditors
Kaif Ahmed | Muhammad Jarir Uddin
Platform
Ethereum and EVM Compatible Chains | Solidity
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review | Functional Testing | Computer-Aided Verification | Manual Review
Git repository/ Commit Hash
Private Repo | 041d229b92c97e96d21f5023d43b5dd55803f047
White paper/ Documentation
Protocol Overview - Gitbook
Document log
Initial Audit Completed: Aug 3rd '2023
Final Audit Completed: Aug 10th '2023

Scope

The shared git-repository was checked for common code violations and vulnerability-specific probing to detect major issues/vulnerabilities. Some specific checks are as follows:

Code review		Functional review
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	Fungible token violations	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

ZeroLiquid is a decentralized protocol that offers non-liquidation, interest-free, self-repaying loans. The protocol allows users to utilize Liquid Staking Derivative (LSD) assets to issue loans against their collateral. Upon depositing LSD tokens, users receive a synthetic token, zETH, which can be traded to provide immediate liquidity. The unique feature of ZeroLiquid is that loans are self-repaying, using the staking rewards generated by the collateral. Users have the option to early unstake by repaying the remaining loan amount.

- Users can borrow against their LSD tokens, with the maximum loan amount being equivalent to 12 months of yield. The borrowed amount is received in the form of zETH, ZeroLiquid's synthetic ETH, which is pegged to the price of ETH. This zETH can be traded in liquidity pools created and incentivized by ZeroLiquid.
- The Diffuser plays a crucial role in maintaining the zETH/ETH peg. Whenever there's a divergence in the peg, arbitrageurs can acquire zETH at a discount and exchange it 1:1 for ETH on the ZeroLiquid protocol. The yield generated from the deposited LSD tokens (after deducting a 10% protocol fee) is directed to the Diffuser at regular intervals. This yield is then distributed proportionally to all zETH submitted for swap. When users claim or swap their ETH from the Diffuser pool, an equivalent amount of zETH is burned. This process ensures peg stability and reduces the overall debt burden of all users proportionally. To further enhance peg stability, ZeroLiquid is implementing LP incentives and integrating zETH with various DeFi protocols to ensure robust liquidity.

ZeroLiquid employs a multi-pronged approach to safeguard the zETH/ETH peg:

- Arbitrage: Users can exploit price differences between zETH and ETH, creating an arbitrage opportunity. When zETH's price is lower than ETH's, users can acquire zETH and convert it 1:1 to ETH using the Diffuser, thus maintaining the peg.
- Incentivized Liquidity: ZeroLiquid incentivizes users to provide liquidity to zETH/ETH pools. Initially, these incentives are in the form of $ZERO tokens, but will later be sustained by protocol revenue. This ensures that the peg remains stable even during large transactions.
- Protocol-owned Liquidity: ZeroLiquid owns the $ZERO/ETH liquidity on Uniswap v3. The ETH generated from LP fees in this pool is used to add zETH/ETH liquidity. This secures the peg and ensures that community-generated fees are reinvested into the platform for the community's benefit.

System Architecture

The ZeroLiquid protocol is built on a robust and secure architecture that enables seamless asset management and loan issuance. The protocol periodically harvests staking rewards, credited proportionally to users, reducing their outstanding debt. This unique feature allows the loans to be self-repaying, providing a hassle-free experience for the users.

The architecture comprises several key components, including the ZeroLiquid, ZeroLiquidToken, and Steamer smart contracts.

ZeroLiquid:

The ZeroLiquid contract is the core of the protocol, managing the deposit of LSD tokens, the issuance of loans, and the harvesting of staking rewards. It ensures the accurate tracking of assets and the proportional distribution of rewards to reduce the outstanding debt of users.

ZeroLiquidToken:

The ZeroLiquidToken contract creates and manages zETH, the protocol's synthetic ETH. It includes functionality for minting and burning tokens and managing the protocol's flash loan feature. The ZeroLiquidToken contract interacts with a number of other contracts and libraries to provide its functionality.

Steamer:

The Steamer contract is a crucial component of the ZeroLiquid Protocol, responsible for managing the protocol's native token. It includes functionality for minting and burning tokens and managing the protocol's flash loan feature.

These components work together to provide a robust and secure platform for creating and managing over-collateralized stablecoins. The architecture is designed to be modular and extensible, allowing for future upgrades and improvements. This ensures that the protocol can continue to evolve and adapt to the changing needs of the DeFi ecosystem.

In addition to these features, ZeroLiquid also allows users to unstake their collateral early by repaying the remaining loan amount. Users can do this by depositing either the LSD token they used as collateral or zETH. Furthermore, users can liquidate a portion or all of their collateral anytime, providing maximum flexibility and control over their assets.

Methodology & Scope

The codebase was audited using a filtered audit technique. A pair of two (2) security researchers scanned the codebase in an iterative process for eighteen (18) days.

Starting with the recon phase, a basic understanding was developed, and the security researchers worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations, software, and security design patterns, code styles, best practices, and identifying false positives detected by automated analysis tools.

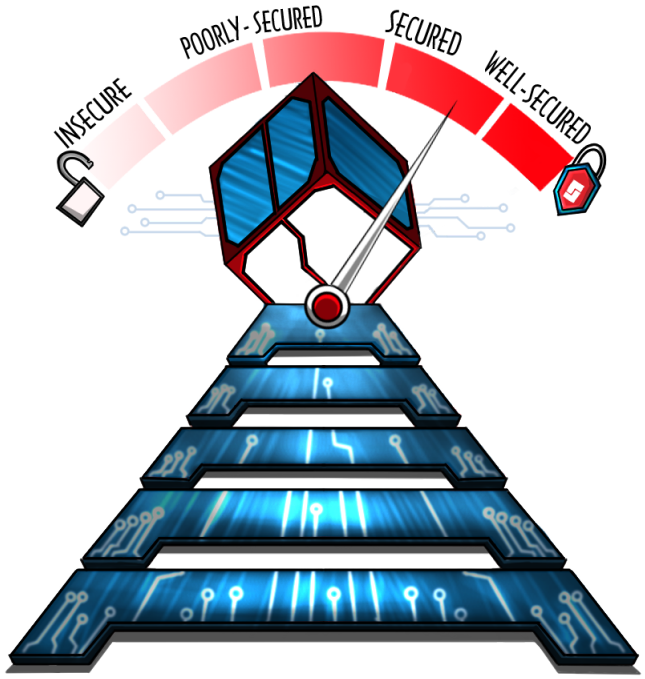
AUDIT REPORT

Executive Summary

Our team performed a technique called *Filtered Audit*, where two individuals separately audited the Zero Liquid Protocol.

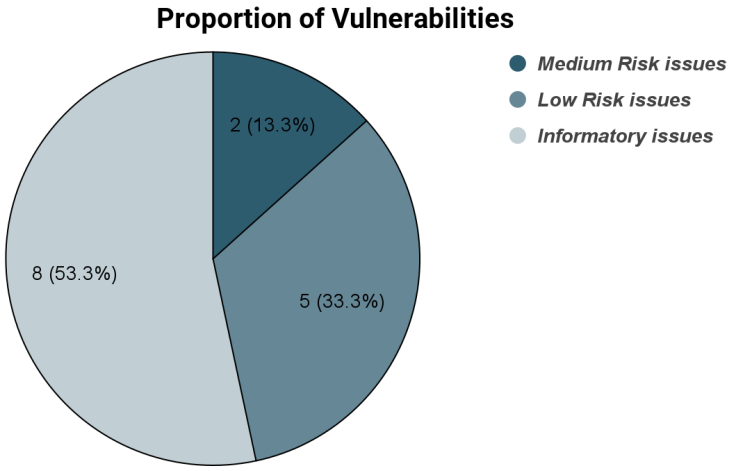
After a thorough and rigorous manual testing process involving line-by-line code review for bugs, an automated tool-based review was carried out using Slither for static analysis.

All the flags raised were manually reviewed and re-tested to identify the false positives.



Our team found:

#Issues	Severity Level
2	Medium-Risk issue(s)
5	Low-Risk issue(s)
8	Informatory issue(s)



Key Findings

#	Findings	Risk	Status
1	Unchecked Delegate Call and Return Values	Medium	Resolved
2	Missing `notPaused` Modifier in Several Functions	Medium	Resolved
3	Insufficient Input Validation in `setPendingAdmin`, `setSentinel`, and `setKeeper` Functions	Low	Resolved
4	Protocol Fee and flashMintFee Can Be Set to 99%	Low	Resolved
5	Insufficient Input Validations in `initialize` Functions	Low	Resolved
6	Potential Underflow in `conversionFactor` Calculation	Low	Resolved
7	flashLoan Function Does Not Check If amount Is Zero	Low	Resolved
8	Insufficient Testing of the Zero Liquid Protocol	Info	Acknowledged
9	Inefficient Mutexes and Insufficient Events	Info	Acknowledged
10	Insufficient Event Emission During Whitelist Update	Info	Acknowledged
11	Order of Layout Violated	Info	Acknowledged
12	Structs Should Be Moved to Interface for Increased Modularity	Info	Acknowledged
13	Typo in Variable Name	Info	Acknowledged
14	Inefficient Initialization of `isPaused` Variable	Info	Acknowledged
15	Missing Input Validation in setTransferAdapterAddress	Info	Acknowledged

Detailed Overview

Medium-risk issues

ID	1
Title	Unchecked Delegate Call and Return Values
Path	`ZeroLiquid.sol`
Severity	Medium
Status	Resolved
Function Name	-

Description:
The `sweepRewardTokens` function uses `delegateCall` to call the `claim` function on the contract at the msg.sender address. However, the return value of the `delegatecall` is not checked, which could lead to unexpected behavior. The scenario arises due to two potential reasons as mentioned above;

- The function `sweepRewardTokens` can only be called by the `keeper` role, however, there is no contract found in the codebase shared that hints at what the keeper contract supposedly does. This leads to a state of indeterminacy where the functionality of `claim()` remains unclear and can affirm one of the two following assumptions;
    - The smart contract is assumed to be safe being outside the scope of the current audit.
    - The contract is unsafe and contains potential risks imposing undetermined threats.
- Return values of the delegate call, in each of the subcases above, are never checked, opening up windows of attack vectors related to such scenarios.

Code-Affected:

```
msg.sender.delegatecall(
    abi.encodeWithSignature("claim(address)", yieldToken)
);
```

Impact:
If the `claim` function fails due to some reasons which are unclear since the codebase is not shared, the `sweepRewardTokens` function will continue execution, potentially leading to incorrect state changes in the ZeroLiquid protocol smart contracts complex. This could lead to loss of funds or other unexpected behavior. Additionally, `claim()` can be used to reenter the smart contract in potential cases like the novel cross-functional reentrancy.

Recommendation:
It is recommended to confirm from the development team the following;

1. Ensure the contract address stored as a keeper role is always safe.
2. Check the return value of the `delegatecall` and revert if it is false. This will ensure that the `sweepRewardTokens` function fails if the `claim` function fails.

Developer Response:
This function is no longer needed and will be removed.

Auditor's Response:
Acknowledged with no follow-up required.

Additional Comments:
The linked reports emphasize on severity of this issue as high (or medium, at minimum) in similar contexts. Fortifying the claim that that the set of developer assumptions can be broken where either the smart contract in silo or the complete smart contract complex is led to an undetermined state - The remediations in all issues linked here focus on two aspects; 1) to document the intended functionality for successful target contract call even if no code executed AND 2) check for return values in any case. E.g. links 1 & 2 focus on unidentified state changes within the smart contracts, 3 & 4 focus on risks associated by such flow of execution, 5 & 6 emphasize on defining the severity of issue for impact and relevant likelihood - Link 1, Link 2, Link 3, Link 4, Link 5, Link 6

ID	2
Title	Missing **notPaused** Modifier in Several Functions
Path	`Streamer.sol`
Severity	Medium
Status	Resolved
Function Name	-

Description:

Several functions in the Steamer contract that should only be callable when the contract is not paused are missing the `notPaused` modifier. This includes the `deposit`, `withdraw`, `claim`, and `exchange` functions. These functions all modify the smart contract's state and should not be callable when the contract is paused. The contract will be entirely vulnerable in such cases, rendering the emergency mechanisms ineffectual.

Code-Affected:

```
function deposit(
    uint256 amount,
    address owner
) external override nonReentrant {
    ...
}

function withdraw(
    uint256 amount,
    address recipient
) external override nonReentrant {
    ...
}
```

```
function claim(
    uint256 amount,
    address recipient
) external override nonReentrant {
    ...
}

function exchange(
    uint256 amount
) external override nonReentrant onlyBuffer {
    ...
}
```

Impact:
If these functions are called while the contract is paused, they could lead to unexpected behavior or potential vulnerabilities. For example, if the `deposit` function is called while the contract is paused, this could lead to an imbalance between the total amount of tokens deposited and the total amount of tokens that can be withdrawn.

Recommendation:
Add the `notPaused` modifier to all functions that should only be callable when the contract is not paused. This will ensure that these functions behave as expected and prevent potential vulnerabilities.

Developer Response:
deposit, withdraw & claim functions do not need to check pause state, only exchange function does. Even if user deposits debt token while paused the unconverted debt tokens can still be withdrawn. Anyother specific function you can name which should check the pause state?

Auditor's Response:
Acknowledged. No other external functions perform critical updates to the smart contract's global state and hence there is no followup required in this regard.

Low-risk issues

ID	3
Title	**Insufficient Input Validation in setPendingAdmin, setSentinel, and setKeeper Functions**
Path	`ZeroLiquid.sol`
Severity	**Low**
Status	**Resolved**
Function Name	**-**

Description:

The `setPendingAdmin`, `setSentinel`, and `setKeeper` functions do not perform any checks on their parameters. This could allow the admin to set the `pending admin`, `sentinel`, or `keeper` to the zero address.

Code-Affected:

```
function setPendingAdmin(address value) external override {
    _onlyAdmin();
    pendingAdmin = value;
    emit PendingAdminUpdated(value);
}
function setSentinel(address sentinel, bool flag) external override {
    _onlyAdmin();
    sentinels[sentinel] = flag;
    emit SentinelSet(sentinel, flag);
}
function setKeeper(address keeper, bool flag) external override {
    _onlyAdmin();
    keepers[keeper] = flag;
    emit KeeperSet(keeper, flag);
```

```
}
```

Impact:
If the `setPendingAdmin`, `setSentinel`, or `setKeeper` functions are called with the zero address, this could cause the contract to be in an incorrect state. This could lead to unexpected behavior and potential loss of control over the contract.

Recommendation:
Add validations to the `setPendingAdmin`, `setSentinel`, and `setKeeper` functions to ensure that the address parameters are not the zero address.

Developer Response:
Setters only enable or disable any specific address using boolean flag, doesn't need any unnecessary validation or concerete meaning of "insufficient" should be provided otherwise. Are you talking about checking if the address was previously enabled or disabled before repeating the same action again?

Auditor's Response:
Acknowledged with no followup required.

ID	4
Title	Protocol Fee and flashMintFee Can Be Set to 99%
Path	`ZeroLiquid.sol`
Severity	Low
Status	Resolved
Function Name	`setProtocolFee, setFlashFee`

Description:

The `setProtocolFee` and the `setFlashFee` functions allow the admin to set the protocol fee to any value less than BPS (10,000 basis points, representing 100%). This means the admin could set the fee to 99.99%, which would be extremely high and could deter users from using the contract's functionality.

Code-Affected:

```
function setProtocolFee(uint256 value) external override {
    _onlyAdmin();
    _checkArgument(value <= BPS);
    protocolFee = value;
    emit ProtocolFeeUpdated(value);
}
function setFlashFee(uint256 newFee) external onlyAdmin {
    if (newFee >= BPS) {
        revert IllegalArgument();
    }
    flashMintFee = newFee;
    emit SetFlashMintFee(flashMintFee);
}
```

Impact:
If the protocol fee or the flash loan minting fee is set to a very high value, this could deter users from using the contract. This could reduce the utility and adoption of the contract.

Recommendation:
Consider implementing a reasonable upper limit (and a lower limit for the flashMintFee state variable) for the protocol fee. This could be a fixed value or a value that governance can update.

Developer Response:
These variables are purposefully dynamic so that they can be adjusted if needed, but the percentage is transparent on chain and not something hidden from users

Auditor's Response:
Acknowledged with no follow-up required.

Additional Comments:
This is a vague intention by design since a common user of DeFi believes in the specifications and figures drawn over the publicly accessible interfaces like a protocol's website or the technical documentation. A security first design ensures that in no case a functionality be abused by an adversary, It is therefore suggested that the developer team specify a range and enforce it in the smart contract code so that even in case of compromise the adversary can never enforce an illegal fee for users.

Final update:
Fixed for both functions.

ID	5
Title	Insufficient Input Validations in `initialize` Functions
Path	`Steamer.sol, ZeroLiquid.sol`
Severity	Low
Status	Resolved
Function Name	-

Description:

The `initialize` function in the Steamer and the ZeroLiquid smart contracts lack sufficient input validation. This function is responsible for setting up the initial state of the contracts, including setting up roles and initializing various state variables. However, it performs no checks on the input parameters such as `_syntheticToken`, `_underlyingToken`, and `_buffer` in the Steamer contract and in the ZeroLiquid contract this missing validation could allow the admin to set invalid values for the `protocolFee` parameter.

Code-Affected:

```
//Steamer.sol
function initialize(
    address _syntheticToken,
    address _underlyingToken,
    address _buffer
) external initializer {
    // no validations in place
    ...
}

//ZeroLiquid.sol
function initialize(
```

```
    InitializationParams memory params
) external initializer {
    _checkArgument(params.protocolFee <= BPS); //no ranges enforced
    // no validations in place
    debtToken = params.debtToken;
    admin = params.admin;
    steamer = params.steamer;
    minimumCollateralization = params.minimumCollateralization;
    protocolFee = params.protocolFee;
    protocolFeeReceiver = params.protocolFeeReceiver;
    ...
}
```

Impact:
Without proper input validation, the `initialize` function could be called with invalid or malicious parameters, leading to unexpected behavior or potential vulnerabilities. For example, if the `_syntheticToken` or `_underlyingToken` parameters are set to the zero address, this could lead to loss of funds or other unexpected behavior. Similarly, if the `_buffer` parameter is set to an address that does not implement the expected interface, this could lead to reverts or other unexpected behavior.

Recommendation:
Add appropriate input validations to the `initialize` function in both smart contracts. Ensure that the provided addresses are not zero and that the tokens have the expected properties (e.g., `decimals`). This will prevent the function from being called with invalid or malicious parameters.

Similarly, for the ZeroLiquid contract, add checks to the initialize function to ensure that all parameters are valid. For example, you could check that the `admin` and `steamer` addresses are not zero and that the `protocolFee` is within a reasonable range.

Developer Response:
Doesn't need any unnecessary validations or concrete meaning of "insufficient" should be provided otherwise.

Auditor's Response:
Acknowledged with no followup required.

Additional Comments:
A faulty initialization in both cases will lead to redeployment of the smart contract complex incurring additional gas cost and addresses recalculation.

ID	6
Title	Potential Underflow in `conversionFactor` Calculation
Path	`Streamer.sol`
Severity	Low
Status	Resolved
Function Name	-

Description:
The `conversionFactor` is calculated as `10 ** (debtTokenDecimals - underlyingTokenDecimals)`. If `underlyingTokenDecimals` is greater than `debtTokenDecimals`, this will result in a revert due to underflow.

Code-Affected:

```
conversionFactor = 10 ** (debtTokenDecimals - underlyingTokenDecimals);
```

Impact:
An underflow in the conversionFactor calculation during the initialization could lead to the failure of contract deployment. Since the initialize function is a crucial part of contract setup, an error during this stage would prevent the contract from being set up correctly. This would render the contract unusable, as the conversionFactor is a critical parameter used in various functions within the contract.

Recommendation:
Add a check in the initialize function to ensure that debtTokenDecimals is greater than or equal to underlyingTokenDecimals before calculating conversionFactor. This will prevent an underflow during the initialization and ensure that the contract is set up correctly. If the check fails, the function should revert with an appropriate error message, indicating that the initialization parameters are invalid.

Developer Response:
What is the potential range of numbers in which underflow occurs?

Auditor's Response:
For all cases; where, `debtTokenDecimals` < `underlyingTokenDecimals` the call to initialize will fail leading to failure in executing the deployment script. Since there exists no check to ensure the reason of failure, debugging will add to overall complexity.

Developer Response:
In our case, debtTokenDecimals (zETH) is 18 & underlyingTokenDecimals (WETH) is also 18 therefore the condition debtTokenDecimals < underlyingTokenDecimals will never be true, but still an edge case was identified therefore, we agree this is a low severity issue.

Auditor's Response:
Acknowledged with no follow-up required.

ID	7
Title	flashLoan Function Does Not Check If amount Is Zero
Path	ZeroLiquidToken.sol
Severity	Low
Status	Resolved
Function Name	-

Description:
The flashLoan function does not check if the amount parameter is zero. If the amount is zero, this would result in a no-operation and waste gas.

Code-Affected:

```
function flashLoan(
    IERC3156FlashBorrower receiver,
    address token,
    uint256 amount,
    bytes calldata data
)
    external
    override
    nonReentrant
    returns (bool)
{
    if (token != address(this)) {
        revert IllegalArgument();
    }

    if (amount > maxFlashLoan(token)) {
        revert IllegalArgument();
    }
```

```
    uint256 fee = flashFee(token, amount);

    _mint(address(receiver), amount);

    if (receiver.onFlashLoan(msg.sender, token, amount, fee, data) !=
CALLBACK_SUCCESS) {
        revert IllegalState();
    }

    _burn(address(receiver), amount + fee); // Will throw error if not enough to burn

    return true;
}
```

Impact:
If the flashLoan function is called with the amount parameter set to zero, this would result in a no-op and waste gas.

Recommendation:
Add a check to the flashLoan function to ensure that the amount parameter is not zero.

Developer Response:
Transaction will revert if amount is zero.

Auditor's Response:
Acknowledged with no followup required.

Additional Comments:
Since the ERC20 standard allows a zero (0) mint, burn and transfers, the Zero Liquid Token contract does not have any additional constraints in place to support the claim that transaction will revert if amount is zero.

Informational issues

ID	**8**
Title	**Insufficient Testing of the Zero Liquid Protocol**
Path	**
Severity	**Informational**
Status	**Acknowledged**
Function Name	**-**

Description:
Throughout the codebase, the testing is insufficient to support the positive and negative test cases from the Zero Liquid protocol engineering team. The substandard testing of such a complex smart contract suite hints at a critical flaw in the Zero Liquid protocol as many cases remain unexplored.

Impact:
The lack of comprehensive testing for the ZeroLiquid protocol can have several potential impacts:

1. Undiscovered Bugs and Vulnerabilities: Without thorough testing, there may be bugs or security vulnerabilities in the code that remain undiscovered. These could potentially be exploited by malicious actors, leading to financial loss for users or even the failure of the protocol.
2. Unpredictable Behavior: Insufficient testing can lead to unpredictable behavior in edge cases or under unusual market conditions. This could result in unexpected losses for users or other undesirable outcomes.
3. Maintenance and Upgrade Challenges: Without comprehensive tests, it can be more difficult to maintain the protocol or to make upgrades in the future. This could slow down the development process and make it harder to respond to changes in the market or the wider DeFi ecosystem.

Recommendation:
It is highly recommended to have a sufficient suite of test cases executed over the production-ready smart contract code to achieve the satisfaction of all functionality working as expected. Additionally, the testing can ensure edge cases are handled elegantly by the protocol ensuring a higher level of security.

Additional Comments:
The Zero Liquid Protocol contains a complex codebase and architecture with a lesser modularity associated as per the industry standard. Hence, sufficient testing refers to the fact that an exhaustive list of test cases be necessarily executed over the production ready code to, at the very least, have the happy test cases work as expected. Not having a test suite creates a lesser level of trust for security researchers as third party reviewers of the protocol's code.

ID	9
Title	Inefficient Mutexes and Insufficient Events
Path	`ZeroLiquid.sol`
Severity	Informational
Status	Acknowledged
Function Name	setTransferAdapterAddress

Description:

The `setTransferAdapterAddress` function in the `ZeroLiquid` contract uses a `lock` modifier commonly employed to prevent reentrancy attacks. However, given this function's behavior, the reentrancy risk seems minimal. Overusing the lock modifier can increase gas costs for the function calls. Additionally, the function does not emit an event after changing the state variable transferAdapter, a recommended practice for transparency and traceability.

Code-Affected:

```
function setTransferAdapterAddress(
        address transferAdapterAddress
    ) external override lock {
        _onlyAdmin();
        transferAdapter = transferAdapterAddress;
    }
```

Recommendation:

1. Remove the lock Modifier: Given that no ether is being sent or received in this function, and no external calls to untrusted contracts after a state change, the lock modifier might be unnecessary. Consider removing it to reduce gas costs.

2. Emit an Event for State Changes: It's a good practice to emit events for significant state changes in a contract. Consider adding an event like TransferAdapterChanged and emit it when the transferAdapter address is updated.

Fixed-Code:

```
event TransferAdapterChanged(address newAddress);

function setTransferAdapterAddress(
    address transferAdapterAddress
) external override {
    _onlyAdmin();
    transferAdapter = transferAdapterAddress;
    emit TransferAdapterChanged(transferAdapterAddress);
}
```

ID	10
Title	Insufficient Event Emission During Whitelist Update
Path	`ZeroLiquidToken.sol`
Severity	Informational
Status	Acknowledged
Function Name	-

Description:
The `setWhitelist` function updates the whitelist of addresses that are allowed to mint new tokens, but it does not emit an event when the whitelist is updated. This makes it harder to track changes to the whitelist.

Code-Affected:

```
function setWhitelist(address minter, bool state) external onlyAdmin {
    whitelisted[minter] = state;
}
```

Impact:
Without an event being emitted when the whitelist is updated, it is harder to track changes to the whitelist. This could make it more difficult for users to understand who is allowed to mint new tokens, and it could make it harder to audit the contract.

Recommendation:
Consider emitting an event in the `setWhitelist` function when the whitelist is updated. This could include the address that was added or removed, and the new state of the whitelist.

ID	11
Title	Order of Layout Violated
Path	`ZeroLiquid.sol`
Severity	Informational
Status	Acknowledged
Function Name	-

Description:
The contract's layout order is violated from the Solidity docs. External functions that implement some functionality should be placed on top so that bytecode saves the code for interactive functions as soon as possible, incurring lesser gas costs to storage when it tries to fetch the code of the function.

Code-Affected:
Various parts of the contract.

Impact:
The layout order in the contract does not affect its functionality, but it could increase the gas cost for storage and execution.

Recommendation:
Consider reordering the functions in the contract according to the Solidity docs. Place external functions that implement some functionality on top.

ID	12
Title	Structs Should Be Moved to Interface for Increased Modularity
Path	`Streamer.sol`
Severity	Informational
Status	Acknowledged
Function Name	-

Description:
Several structs are defined in the Steamer contract that could be moved to the ISteamer interface. This would increase the modularity, reusability, and readability of the code.

Code-Affected:

```
struct Account {...}
struct UpdateAccountParams {...}
struct ExchangeCache {...}
```

Impact:
By defining these structs in the Steamer contract rather than the ISteamer interface, the code is less modular and reusable. Other contracts that interact with the Steamer contract might need to define their versions of these structs, leading to code duplication and potential inconsistencies.

Recommendation: Move the definition of these structs to the ISteamer interface. This will make the code more modular and reusable and ensure that other contracts that interact with the Steamer contract can use the same definitions of these structs.

ID	13
Title	Typo in Variable Name
Path	`Streamer.sol`
Severity	Informational
Status	Acknowledged
Function Name	-

Description:
There is a typo in the variable name normaizedAmount in the exchange function. This could lead to confusion for developers reading or maintaining the code.

Code-Affected:

```
uint256 normaizedAmount = _normalizeUnderlyingTokensToDebt(amount);
```

Recommendation:
Correct the typo in the variable name normalized Amount to normalizedAmount. This will improve the readability of the code.

ID	14
Title	Inefficient Initialization of `isPaused` Variable
Path	`Streamer.sol`
Severity	Informational
Status	Acknowledged
Function Name	-

Description:
The isPaused variable is initialized to false in the initialize function, which is unnecessary as boolean variables in Solidity are false by default.

Code-Affected:

```
isPaused = false;
```

Impact:
This unnecessary initialization of isPaused to false results in additional gas costs during contract deployment. While the impact is low, it is a waste of gas and could be easily avoided.

Recommendation:
Remove the unnecessary initialization of isPaused to false in the initialize function. This will reduce the gas cost of contract deployment.

ID	15
Title	Missing Input Validation in setTransferAdapterAddress Function
Path	ZeroLiquid.sol
Severity	Informational
Status	Acknowledged
Function Name	-

Description:
The setTransferAdapterAddress function does not perform any checks on its parameter. This could allow the admin to set the transfer adapter address to zero.

Code-Affected:

```
function setTransferAdapterAddress(
    address transferAdapterAddress
) external override lock {
    _onlyAdmin();
    transferAdapter = transferAdapterAddress;
}
```

Impact:
If the setTransferAdapterAddress function is called with the zero address, this could cause the contract to be in an incorrect state. This could lead to unexpected behavior and potential loss of control over the contract. Additionally, it will incur gas costs to reset the contract with the correct values to call the relevant functions again.

Recommendation:
Add a check to the setTransferAdapterAddress function to ensure that the address parameter is not the zero address.

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices to date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered and should not be interpreted as an influence on the potential economics of the token, its sale, or any other aspect of the project.

Crypto assets/tokens are the results of emerging blockchain technology in the domain of decentralized finance, and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service, or another asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond Solidity that could present security risks.

This audit cannot be considered 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 recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.