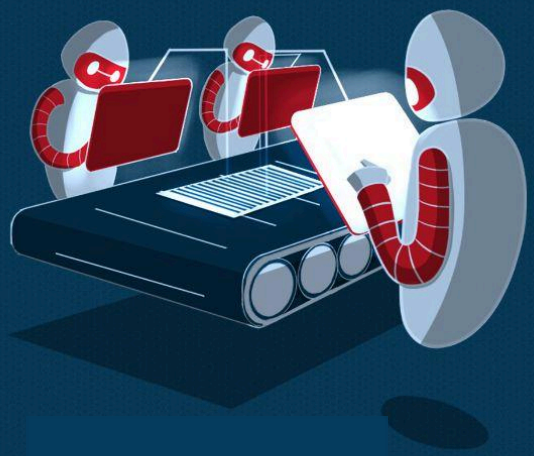
BlockÂpex

# SMART CONTRACT SECURITY ANALYSIS REPORT

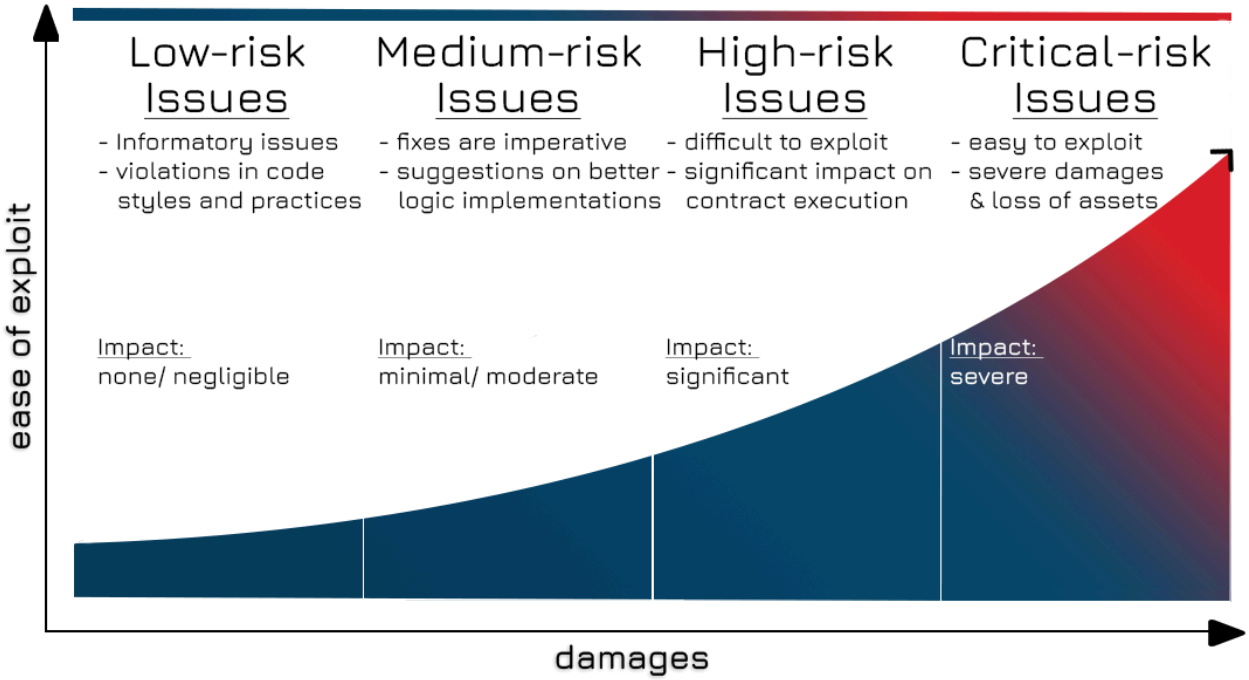
```
pragma solidity 0.7.0;
contract Contract {

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

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

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



# Objectives

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

# Key understandings



# TABLE OF CONTENTS

# INTRODUCTION

BlockApex (Auditor) was contracted by Stashed (Client) for the purpose of conducting a Smart Contract Code Security Review. This document presents the findings of our analysis which started on 18th Sep '2023

| Name |
| --- |
| Stashed |
| Auditor |
| Kaif Ahmed \| Muhammad Jariruddin |
| Platform |
| Ethereum \| EVM Compatible Blockchains \| 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 \| c146eb5fe50a9678ec63019a44c2d31ef64bd067<br>Final Commit \| d5ee20d41b7fb035232da47a6213260670cd8127 |
| White paper/ Documentation |
| - |
| Document log |
| *Initial Audit Completed: Sep 29 '2023* |
| *Interim Audit Completed: Oct 28 '2023* |
| *Final Audit Completed: 7th Nov '2023* |

# Scope

The shared git-repository/ codebase was checked for common code violations along with vulnerability-specific probing to detect major issues/vulnerabilities.

Some specific attack vectors and threat surfaces 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

The project is a sophisticated smart contract system, primarily built on Solidity, and it seems to be part of a decentralized finance (DeFi) ecosystem. It consists of several interconnected contracts, each serving a unique purpose and contributing to the overall functionality of the system. Below is a brief overview of each component and its role within the system.

## 1. StashAccount and StashAccountFactory

**StashAccount:** Represents a user account with functionalities to execute transactions, manage deposits, and perform account-related operations. It has mechanisms to interact with an EntryPoint and execute transactions either directly from the owner or through the EntryPoint.
**StashAccountFactory:** Responsible for creating StashAccount instances. It can calculate the counterfactual address of an account and deploy new accounts using the ERC1967Proxy pattern.

## 2. StashTokenPaymasterV2

- This contract is central to managing token-based payments within the ecosystem. It allows for the deposit and withdrawal of funds in both ERC20 tokens and the native cryptocurrency, ETH.
- A notable feature is the contract's ability to lock and unlock token deposits, ensuring users have a sufficient balance to cover transaction costs.
- The inclusion of a whitelisting function allows the system to support a curated list of tokens, with each token's transaction fees and decimal precision being adjustable by the owner.
- The paymaster handles conversions between token values and ETH, relying on off-chain price oracles, with the price being agreed upon by the user. This ensures transparency and user consent in transaction fee deductions.
- The contract introduces nonce management for individual users, enhancing security by preventing replay attacks and ensuring the legitimacy of transaction sponsorship.

- With its sophisticated signature validation system, the contract allows for secure and verifiable transactions, with the capacity to support self-sponsored transactions as well as third-party sponsorship.
- The system architecture ensures that the StashTokenPaymasterV2 is integral to the user experience, managing their deposits and facilitating their transactions with an emphasis on security and user autonomy.

## 3. StashVerifyingPaymaster

This contract is another type of paymaster that relies on an external service to decide whether to pay for the UserOp. It trusts an external signer to sign the transaction after performing off-chain verification. It validates the external signer's signature and manages sender nonces to prevent replay attacks.

# System Interaction and Flow

## User Account Creation and Management

Users interact with the StashAccount and StashAccountFactory for account creation and management, executing transactions, and managing deposits.

## Payment Management

The StashTokenPaymasterV2 and StashVerifyingPaymaster manage the payments and deposits for the user operations, ensuring the users have enough deposits and validating the signatures from external signers.

# Considerations and Assumptions:

- The contracts have mechanisms to ensure the validity of token addresses and prices, and the availability of sufficient deposits for user operations.
- The system seems to be designed with security in mind, implementing safeguards like ReentrancyGuard and ensuring proper access control with modifiers like onlyOwner.
- The contracts are well-structured and modular, allowing for a clear separation of concerns and responsibilities among different components of the system.

# Conclusion:

This system appears to be a part of a larger DeFi ecosystem, focusing on user account management, payment handling, and price information provision. It is built with robustness and security in mind, utilizing various mechanisms to ensure the integrity and reliability of user operations and transactions within the ecosystem.

# System Architecture

The system architecture is modular and decentralized, consisting of several smart contracts, each serving a distinct purpose within the ecosystem. Below is a detailed breakdown of the system architecture.

## 1. User Account Management

### StashAccount

- Represents individual user accounts.
- Handles execution of transactions and management of deposits.
- Interacts with an EntryPoint to execute transactions.

### StashAccountFactory

- Responsible for creating instances of StashAccount.
- Calculates the counterfactual address of an account.
- Deploys new accounts using the ERC1967Proxy pattern.

## 2. Payment and Deposit Management

### StashTokenPaymasterV2

- This upgraded paymaster contract is a foundational component for the transactional operations of the system.
- It offers an advanced deposit management system, wherein users can deposit tokens or ETH, which are then secured and allocated for future transaction fees.
- The contract's withdrawal mechanism is safeguarded by the ReentrancyGuard to prevent potential vulnerabilities during fund transfers.
- A key enhancement in this version is the token whitelisting capability, which enables the system to filter and manage an approved list of tokens. This feature is critical for maintaining a secure and curated ecosystem of supported tokens.

- The contract employs a dual-layered security feature for transaction validation: nonce tracking for users and ECDSA signature verification. This ensures that each operation is unique and correctly authorized.
- Post-operation, the contract calculates the exact token fee, accounting for the actual gas used and agreed-upon token price, deducting the fee from the user's balance or compensating the paymaster accordingly.
- The robust token management system is designed for flexibility in token economics, allowing the owner to set and adjust fees and decimals, thereby maintaining the economic stability of the platform's operations.

### StashVerifyingPaymaster

- Relies on an external service to decide whether to pay for the UserOp.
- Validates the external signer's signature.
- Manages sender nonces to prevent replay attacks.

## Interaction Flow

### User Interaction

Users interact with StashAccount and StashAccountFactory for account creation, management, and execution of transactions.

### Payment Handling

StashTokenPaymasterV2 and StashVerifyingPaymaster manage payments and deposits, ensuring sufficient deposits and validating external signatures for user operations.

## Architectural Considerations

**Modularity:** The architecture is modular, with a clear separation of concerns among different components, allowing for easier maintenance and upgrades.

**Security**: The architecture implements various security mechanisms like ReentrancyGuard and access control modifiers to ensure the security of operations and transactions.

**Decentralization**: The architecture is decentralized, relying on smart contracts and blockchain technology to ensure transparency, security, and integrity of user operations and transactions.

**Scalability:** The modular and decentralized nature of the architecture allows for scalability, enabling the addition of new features and components as needed.

## Conclusion

The architecture is well-structured and robust, designed to handle user account management, payment, and deposit management, and provide accurate price information within a decentralized finance ecosystem. It is built with security, modularity, and scalability in mind, allowing for the development of a reliable and expandable DeFi platform.

# Methodology & Scope

The codebase went through a security review using a filtered code review technique. A pair of two (2) security researchers scanned the codebase in an iterative process for a time spanning two (2) weeks.

1. Started with the reconnaissance phase, a basic understanding was developed.
2. The security researchers worked on developing presumptions for the production-ready codebase and the relevant documentation/ white paper provided by the client protocol.
3. The security audit moved up to the manual code reviews with the motive of finding logical flaws in the codebase.
4. Further complemented with code optimizations, software, and security design patterns implementation, code styles, best practices, and identifying false positives that were detected by automated analysis tools.

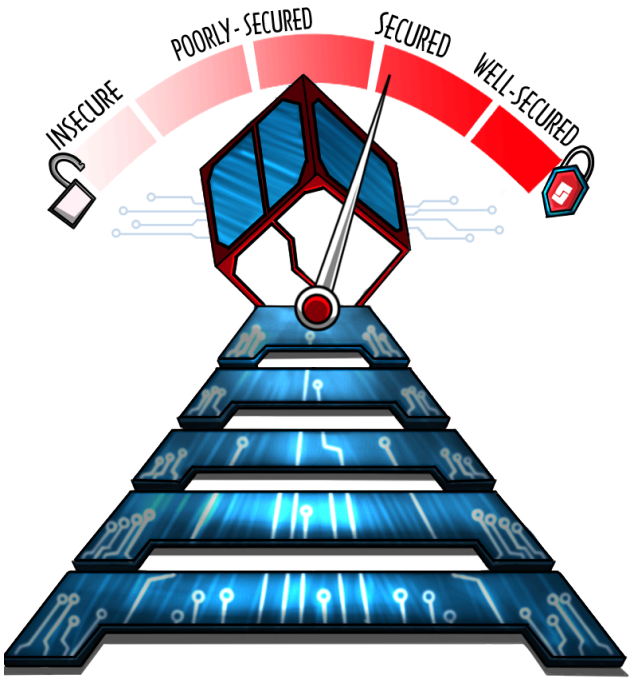
# SECURITY REVIEW REPORT

## Executive Summary

Our team performed a technique called "Filtered Security Review", where the Stashed codebase was separately reviewed by two individuals.

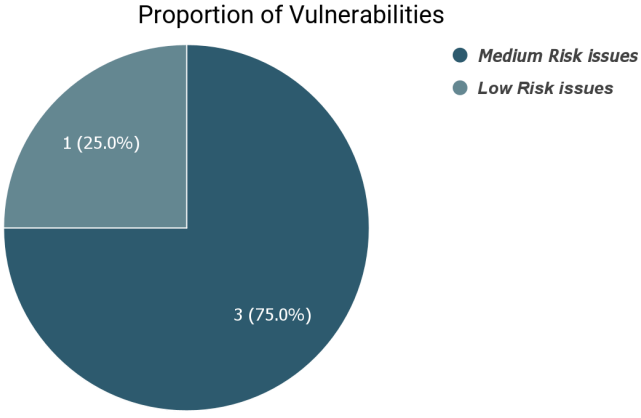
After a thorough and rigorous process involving manual code review, automated testing was carried out using; Slither for static analysis and Foundry for fuzzing invariants.

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




**Our team found:**

| Issues | Severity Level | Open | Resolved | Acked |
|---|---|---|---|---|
| - | Critical | - | - | - |
| - | High | - | - | - |
| 3 | Medium | 0 | 2 | 1 |
| 1 | Low | 0 | 1 | 0 |
| - | Informatory | - | - | - |
| - | Undetermined | - | - | - |



# Key Findings

| # | Findings | Risk | Status |
|---|---|---|---|
| 1. | Potentially Inexplicable Transaction Fees | Medium | Fixed |
| 2. | Account Abstraction Assumption Violated for EIP-1559 | Medium | Acked |
| 3. | Reentrancy due to Bad Design for External Calls | Medium | Fixed |
| 4. | Inappropriate Input Validations | Low | Fixed |

# Findings

## Detailed Overview

### Medium-risk issues

| ID | 1 |
|---|---|
| Title | Potentially Inexplicable Transaction Fees |
| Path | src/StashTokenPaymasterV2.sol |
| Function Name | whitelistToken( ) |

**Description:** We have identified a potential issue in the `StashTokenPaymasterV2` contract where the smart contract owner has the ability to set transaction `fees` up to `100%` of the transaction cost. This is concerning as it allows the owner to double the cost that users must pay while interacting with the smart contract.

**Location:**

The issue is located in the `whitelistToken` function where the `fee` value for a token can be set, and in the `_validatePaymasterUserOp` function where the transaction cost is calculated with the fees added.

**Code Reference:**

```
function whitelistToken(
        address token,
        uint8 fee,
        uint8 decimals
    ) public onlyOwner {
        require(token != address(0), "STP: Zero Address");
```

```
        require(!isWhitelisted[token], "STP: Token Already exists.");
        tokenData[token].fees = fee;
        tokenData[token].decimals = decimals;
        isWhitelisted[token] = true;
    }
```

Impact:

- **High Tx Cost**: If the contract owner sets the fees to 100%, the total cost deducted from the user's balance for a transaction will be twice the estimated transaction cost. This could result in excessive charges for users, potentially leading to a loss of trust and a reduction in the contract's usage.
- **Lack of Transparency**: The documentation does not cover the aspects of a `fee` overhead hence users are unaware of the additional fee being added, leading to unexpected costs and potential trust issues.
- **User Deterrence**: Unjustified or high fees, coupled with lack of transparency, could deter users from interacting with the contract.

Proposed Recommendations:

- **Use Fee Cap**: Implement a `maximumFeePercentage` that is deemed acceptable by community standards and ensure this is enforced in the contract logic.
- **Improve Transparency**: Clearly document the `fee` structure and provide user-friendly functions allowing users to calculate the total cost before executing transactions.

```
function calculateTotalCost(uint256 gasCost, address token) public
view returns (uint256 totalCost) {
    uint256 fee = (tokenData[token].fee * gasCost) / 10000; //
Assuming fee is in basis points
    return gasCost + fee;
}
```

- **Justify and Optimize Fee Structure**: Ensure the `fee` is reasonable, provides value to the users, and is used to enhance the sustainability or security of the contract. Provide a rationale for the `fee` in the documentation and consider user

feedback to optimize the fee structure. Transparency in fee structure is vital, and users should be clearly informed about any fees that will be applied to their transactions.

| ID | 2 |
| --- | --- |
| Title | Account Abstraction Assumption Violated for EIP-1559 |
| Path | src/StashTokenPaymasterV2.sol |
| Function Name | addDepositFor() |

**Description:** In the Ethereum Blockchain, with the introduction of EIP-1559, transaction pricing underwent a significant change. Before EIP-1559, the transaction fee was solely based on a `gasPrice` set by the transaction sender. After EIP-1559, a new mechanism was introduced where the fee has two components: `baseFeePerGas` and `maxFeePerGas`. The `baseFeePerGas` is burned and `maxFeePerGas` is the maximum gas price a sender is willing to pay.

If you're using `userOp.gasPrice()`, the old gas pricing mechanism might not be compatible with post-EIP-1559 mechanisms. If you're in a post-EIP-1559 context, then it is highly suggested to switch to `userOp.maxFeePerGas`. Using the incorrect method in a post-EIP-1559 environment could lead to unexpected behavior and potential issues.

**Recommendations:**

- It is highly suggested to change from `userOp.gasPrice()` to `userOp.maxFeePerGas`, we are anticipating a scenario where the `gasPrice()` function, as provided by you, could lead to invoking `BASEFEE` (since it includes block.basefee in its calculation for post-EIP-1559 transactions). Using `userOp.maxFeePerGas` directly would bypass any such calculations and therefore avoid the risk of inadvertently using the banned opcode.
- The decision to use `userOp.gasPrice()` or `userOp.maxFeePerGas` directly should be correlated to the network's capabilities where the smart contract is designed to be deployed. If the network does not support EIP-1559, avoid using any logic that could invoke `BASEFEE`, which ultimately includes avoiding `userOp.gasPrice()`.

**Developer Response:**

To be handled via SDK.

| ID | 3 |
| --- | --- |
| Title | Reentrancy due to Bad Design for External Calls |
| Path | src/StashTokenPaymasterV2.sol |
| Function Name | addDeposit() |

Description:

The `StashTokenPaymasterV2` smart contract in the codebase contains an open attack surface where the function has a violating design practice for unsafe external calls. The `addDeposit` function allows a user to deposit their ERC20 tokens in order to be later withdrawn or used as gas by the parameterized account. As seen in the code referred to below, the function takes in a `token` argument of type address and calls the `safeTransferFrom` function on it to move funds from the `msg.sender` to the smart contract's own address, finally updating its internal states for the `balances` mapping.

Impact:

The checks-effects-interactions pattern is a core solidity design pattern for smart contracts on EVM. Violating such a pattern leads to cases of reentrancy such as classic, cross-functional etc. Since the contract inherits from the OpenZeppelin's ReentrancyGuard library it is evident that the current implementation design goes against the reentrancy protection mechanisms. This pattern has proven to lead to cases of users losing their funds where hackers and malicious actors exploit the smart contracts code logic for their benefit.

Code Reference:

```
    function addDeposit(address token, address account, uint256 amount)
external payable {
        // Sender must have approval for the paymaster
        require(isWhitelisted[token], "STP: Token not supported");
        if (token != nativeAddress) {
            require(amount > 0, "STP: Cannot transfer 0 amount.");
```

```
            IERC20(token).safeTransferFrom(msg.sender, address(this),
amount);
            balances[token][account] += amount;
        } else {
            require(msg.value > 0, "STP: Cannot transfer 0 amount.");
            balances[token][account] += msg.value;
        }
    }
```

Proposed Recommendations:

It is highly suggested that the function is corrected by design, where the implementation reflects the design choices appropriately such that the system's overall security is at the forefront. To cater to this issue, the order of conditional constraints, state updates, and external calls should follow the checks-effects-interactions design pattern.

Also, a minor optimization to cater here is that the `safeTransferFrom` is designed to be used to approve funds moving between via some `spender` on behalf of the fund's `owner`. However, in the current implementation for the `msg.sender` itself, the function `safeTransferFrom` is used which is unorthodox in any case and should be corrected as follows.

```
    function addDeposit(address token, address account, uint256 amount)
external payable {
        // Sender must have approval for the paymaster
        require(isWhitelisted[token], "STP: Token not supported");
        if (token != nativeAddress) {
            require(amount > 0, "STP: Cannot transfer 0 amount.");
            balances[token][account] += amount;
            IERC20(token).safeTransfer(address(this), amount);
        } else {
            require(msg.value > 0, "STP: Cannot transfer 0 amount.");
            balances[token][account] += msg.value;
        }
    }
```

## Low-risk issues

| | |
|---|---|
| ID | 4 |
| Title | Inappropriate Input Validations |
| Path | src/StashTokenPaymasterV2.sol |
| Function Name | addDeposit(), withdraw(), unlistToken(), _validatePaymasterUserOp() |

**Description:**

In the `StashTokenPaymasterV2` smart contract, the above-mentioned functions do not properly place input validations for incoming function arguments. The functions commonly accept the address arguments and modify the internal storage of the smart contract therefore storing unintended or malintended values against the local states can be consequential in some cases.

**Impact:**

Failing to place safety input validations has caused loss to protocols and ultimately users in most of the cases in the web3 ecosystem. For instance, not checking for zero addresses while storing an account value is detrimental as it acts like a straight sink for funds where funds sent to zero addresses, although accidentally or by mistake are unrecoverable. Additionally, missing input validations, an apparently simple violation can create critical severity vulnerability in the smart contract code if combined with another low-risk issue of the same category, that is, bad coding practices.

Alternatively, adding checks at unnecessary instances can incur additional gas costs to the user. So a robust design that is highly tailored and crafted not leaving any windows of inconsistencies in the codebase.

**Code Reference:**

```
function addDeposit(address token, address account, uint256 amount)
```

```
external payable {...}

function withdraw(address token, address target, uint256 amount) public
nonReentrant {...}

function unlistToken(address token) public onlyOwner {...}

function _validatePaymasterUserOp(UserOperation calldata userOp, bytes32
userOpHash, uint256 maxCost) internal view override returns (bytes memory
context, uint256 validationData) {...}
```

### Proposed Recommendations:

It is recommended that the functions have proper checks for zero address in place, specifically as follows:

1. Place a zero address check for the function argument `account` in `addDeposit()` and the variable named `target` in `withdraw()`.
2. Remove the zero address check for the `token` argument from the `unlistToken()` to enforce a consistent pattern of design so that any redundant operation is removed from the code.
3. Place a short-circuit check for `signature.length` in `_validatePaymasterUserOp()` to be `64` as well as the currently validated length of `65` only.

# DISCLAIMER

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities 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 (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Crypto assets/ tokens are the results of the 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 programmable code and only the programmable code, we note, as being within the scope of our review within this report. The smart contract programming 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 the scope of the programming language's compiler that could present security risks.

This security review 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 BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of smart contracts.