BlockApex

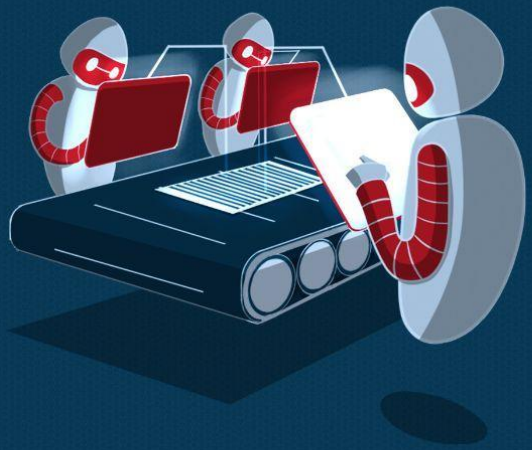
SMART CONTRACT SECURITY ANALYSIS REPORT

```
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";
    }
}
```



Powered by XORD

PREFACE

Objectives

The purpose of this document is 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 intellectual property of the client. It also contains information about potential risks and the processes involved in mitigating/exploiting the risks mentioned below. The usage of 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 discretion of the client.

Key understandings

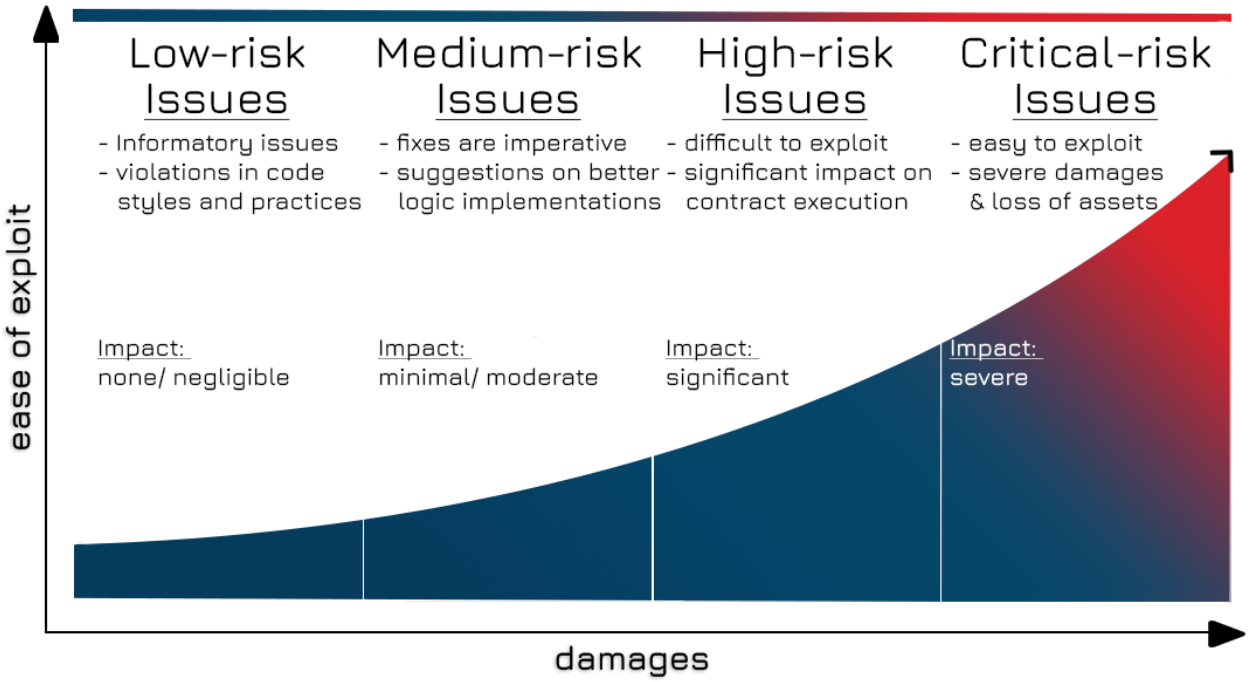


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by Unipilot (Client) for the purpose of conducting a Smart Contract Audit/ Code Review. This document presents the findings of our analysis which started on 14th Feb 2023

Name
Unipilot
Auditor
BlockApex
Platform
Polygon (L2) | 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 | fe0ee0bfa5f7cdea43575fb994cfe8aec6249bbc
White paper/ Documentation
https://unipilot.gitbook.io/unipilot/
Document log
Initial Audit Completed: Mar 6th, 2023
Final Audit Completed: Apr 7th, 2023

Scope

The git-repository shared was checked for common code violations along with 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

Unipilot is an automated liquidity manager designed to maximize ”in-range” intervals for capital through an optimized rebalancing mechanism of liquidity pools on Uniswap V3 and Algebra’s V3 implementation of the QuickSwap DEx. The Unipilot’s V2 also detects the volatile behavior of the QuickSwap pools in order to pull liquidity until the pool gets stable to save the pool from impermanent loss.

System Architecture

The protocol is built to support multiple dexes (decentralized exchanges) for liquidity management. Currently, it supports Uniswap v3 as well as **Quickswap's** liquidity now. In the future, the protocol will support other decentralized exchanges like Sushiswap (Trident). The architecture is designed to keep in mind future releases.

The protocol has 6 main smart contracts and their dependent libraries.

UnipilotActiveFactory.sol

The smart contract is the entry point in the protocol. It allows users to create a vault if it's not present in the protocol. Nevertheless, Active vaults can only be created by governance.

UnipilotPassiveFactory.sol

The smart contract is the entry point in the protocol. It allows users to create a vault if it's not present in the protocol. However passive vaults can be created by anyone.

UnipilotActiveVault.sol

Vault contract allows users to deposit, withdraw, readjustLiquidity and collect fees on liquidity. It mints LPs to its users representing their individual shares. It also has a pullLiquidity function if liquidity is needed to be pulled.

UnipilotPassiveVault.sol

PassiveVault contract allows users to deposit, withdraw, readjustLiquidity and collect fees on liquidity. It mints LPs to its users representing their individual shares.

UnipilotStrategy.sol

The smart contract to fetch and process ticks' data from Uniswap. It also decides the bandwidth of the ticks to supply liquidity.

UnipilotMigrator.sol

The smart contract aids to migrate users' liquidity from other Quickswap Liquidity Optimizer Protocols to Unipilot V2 Protocol.

Methodology & Scope

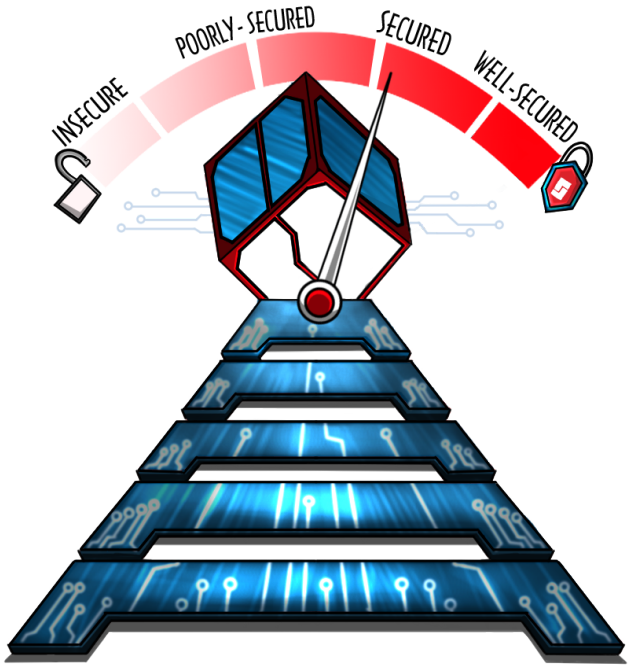
The codebase was audited using a filtered audit technique. A band of three (3) auditors scanned the codebase in an iterative process spanning over a time of three (3) weeks.

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

AUDIT REPORT

Executive Summary

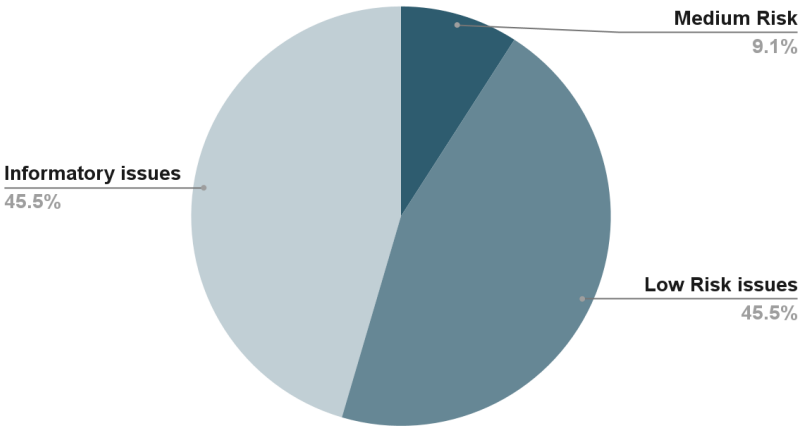
Our team performed the "Filtered Audit" technique, where the codebase is separately audited by individual security researchers.
Alongside a rigorous process of manual testing, an automated code review was carried out using the renowned tool Slither for static analysis and the framework Foundry for testing out edge cases and fuzzing invariants of the Program under test.
All the flags raised were manually reviewed and re-tested to identify and resolve any false positives.



Our team found:

No. of issues	Severity of the risk
0	Critical Risk issue(s)
0	High Risk issue(s)
1	Medium Risk issue(s)
5	Low Risk issue(s)
5	Informatory issue(s)

Proportion of Vulnerabilities



Key Findings

#	Findings	Risk	Status
1	Inconsistent specification and implementation	Medium	Fixed
2	Unsafe transfer of ownership	Low	Acknowledged
3	Insufficient input validation	Low	Fixed
4	Ineffectual limits on IndexFundPercentage	Low	Fixed
5	Missing Event Emission	Low	Acknowledged
6	Missing Error Messages in Modifiers	Low	Acknowledged
7	Unused Parameter	Informational	Pending
8	Unnecessary Variable Assignment	Informational	Acknowledged
9	Inconsistent variable naming	Informational	Fixed
10	Nonconformance with best practices	Informational	Acknowledged
11	Redundant requirement checks	Informational	Fixed
12	Centralization Risks	Unclassified	Resolved

Detailed Overview

Medium-risk issues

ID	1
Title	Inconsistent specification and implementation
Path	contracts/UnipilotActiveVault.sol
Function Name	pullLiquidity

Description:

The `pullLiquidity` function in the active vault smart contract can be executed only by an operator role due to the `onlyOperator` modifier. However, the comment above the function states that it can:

/// Only called by the governor or selected operators

Therefore, the current modifier is not accurate as it does not match the comment's description, potentially leading to confusion and unauthorized access to the function.

Code:

```
/// @dev Burns all the Unipilot position and HODL in the vault to prevent users from huge IL
/// Only called by the governor or selected operators
/// @dev Users can also deposit/withdraw during HODL period.
 function pullLiquidity() external onlyOperator {...}
```

Recommendation:

To ensure accurate access control to `pullLiquidity`, it is advised to modify the codebase as below:

1. Create a new modifier that checks whether the caller is the governance address or an operator.
    a. This new modifier can be named `onlyGovOrOperator`, for example, and used instead of the current `onlyOperator` modifier.
2. The comment above the function should also be updated to reflect the accurate description of the function's access control.

Low-risk issues

ID	2
Title	Unsafe transfer of ownership
Path	contracts/UnipilotActiveFactory.sol
Function Name	setGovernance

Description:
The `setGovernance` function in smart contracts does not have a safe transfer for ownership, implying that the governance can unilaterally change the address of the new governance without any restriction. This leaves the contract vulnerable to a potential attack vector where the governance can be taken over by a malicious actor, who can then manipulate the contract's functions, steal funds or cause other damage to the users.

Impact:
The lack of a safe ownership transfer on the `setGovernance` function can have severe consequences for the smart contract's overall security. If the governance is ever compromised, an adversary can exploit the vulnerability to manipulate the contract's functionalities and states, such as changing token prices, transferring funds to unauthorized addresses, and more. Such attacks can lead to financial losses for the contract and its users, damaging the trust in the project.

Code:

```
function setGovernance(address _newGovernance) external onlyGovernance {
    require(_newGovernance != address(0));
    emit GovernanceChanged(governance, _newGovernance);
    governance = _newGovernance; //@audit two step ownership transfer?
  }
```

Recommendation:
To mitigate this vulnerability, a two-step ownership transfer mechanism should be implemented on the `setGovernance` function. The first step could involve the current governance signing the transaction, while the second step could involve a community-led multi-sig that approves the change in governance address. This will provide an additional layer of security and ensure that the governance transfer is not arbitrary.

Developer Response:
Our governance is managed with a multi-sig wallet.

Auditor Response:
Resolved with no further action.

ID	3
Title	Insufficient input validation
Path	contracts/UnipilotActiveVault.sol contracts/UnipilotPassiveVault.sol
Function Name	deposit

Description

The `deposit` function in both contracts does not have a check for the zero address on the `recipient` parameter. The function is used to deposit tokens into the liquidity pool, and the `recipient` parameter is used to specify the address of the account that will receive the LP tokens. However, the lack of a zero address check on the `recipient` parameter can cause funds to be sent to an invalid address or a contract that is not intended to receive them.

Recommendation:

To mitigate this vulnerability, a check for the zero address should be implemented on the `recipient` parameter. The check can be done by adding a `require` statement at the beginning of the function to ensure that the recipient parameter is not a zero address.

ID	4
Title	Ineffectual limits on indexFundPercentage
Path	contracts/UnipilotActiveFactory.sol
Function Name	setUnipilotDetails, constructor

Description:

There is no upper limit set on the percentage value being passed to this parameter, making it possible for someone to set an abnormally high value. Moreover, there is also no upper limit on the `indexFundPercentage` parameter in the `setUnipilotDetails` function, making it possible for anyone to set an arbitrarily high percentage value. Additionally, the constructor of the above mentioned contract takes an input parameter named percentage, which should be more appropriately named `indexFundPercentage` to avoid confusion.

Code:

```
function setUnipilotDetails(
    address _strategy,
    address _indexFund,
    uint8 _indexFundPercentage
  ) external onlyGovernance {
    strategy = _strategy;
    indexFund = _indexFund;
    indexFundPercentage = _indexFundPercentage;
  }
}
```

Impact

The lack of clarity in the variable name could lead to confusion and make it difficult for other developers to understand the code. Additionally, the absence of a cap on the percentage value could result in an excessively high percentage being set, leading to a

large portion of the pool assets being allocated to the index fund, which could be detrimental to the pool's performance.

Recommendation:
It is suggested that following changes be made to the code for efficient and optimized execution of the function:

1. Rename the percentage parameter in the constructor to `indexFundPercentage` to make it more clear.
2. Set an upper limit on the `indexFundPercentage` parameter to prevent an authorized user from setting an excessively high value.
3. A limit should also be defined for the `indexFundPercentage` parameter in the `setUnipilotDetails` function for the same purpose.
    a. This limit should be set at a reasonable percentage value to ensure the proper functioning and performance of the vault.

ID	5
Title	Missing Event Emission
Path	contracts/UnipilotActiveFactory.sol contracts/UnipilotPassiveFactory.sol
Function Name	setUnipilotDetails

Description:
In the passive and active factory smart contracts the function `setUnipilotDetails` does not emit a relevant event over the updates made to the `strategy`, `indexFund`, and `indexFundPercentage` variables. This can create difficulties in tracking the changes made to the contract, as there is no way to know when these sensitive parameters are updated.

Impact:
The absence of event emissions can make it difficult to monitor and track changes to the contract's state. As a result, there is a risk that unauthorized changes to sensitive parameters can go unnoticed, leading to serious security issues. Additionally, it can hinder the troubleshooting process and make it difficult to identify and address issues that may arise.

Recommendation:
It is recommended to add event emissions to the `setUnipilotDetails` function to improve transparency and facilitate monitoring of the contract's state changes. An event can be added that includes the old and new values of the variables being changed. This will enable the contract administrators and users to track changes and take necessary actions if any unexpected changes occur.

Developer Response:
Acknowledged. Contract size exceeding issue.

Auditor Response:
Resolved with no further action.

ID	6
Title	Missing Error Messages in Modifiers
Path	contracts/UnipilotActiveVault.sol contracts/UnipilotActiveFactory.sol contracts/UnipilotActiveFactory.sol
Function Name	onlyGovernance, onlyOperator

Description:

The modifiers `onlyGovernance` and `onlyOperator` are used in the contract, but they do not provide any error messages when the requirements in the modifier fail. The lack of error messages can make it difficult to understand why a transaction fails and can make debugging the contract more challenging for developers and all users interested in understanding the flow of the contract on a technical level.

Code:

```
modifier onlyGovernance() {
      (address governance, , , , ) = getProtocolDetails();
      require(msg.sender == governance);
      _;
   }
modifier onlyOperator() {
      require(_operatorApproved[msg.sender]);
      _;
   }
```

Recommendation:

Including informative error messages is a helpful practice in software development. Clear error messages can save time and help developers solve issues faster, which improves

the user experience. It's important to provide an error message that explains why a transaction failed and offers guidance on how to fix the issue.

Developer Response:
Acknowledged. Contract size exceeding issue.

Auditor Response:
Resolved with no further action.

Informatory issues and Optimizations

ID	7
Title	Unused Parameter
Path	contracts/UnipilotActiveFactory.sol

Description:
The code contains an unused variable `swapPercentage` which is not set or used anywhere in the contract. This could potentially lead to confusion for other developers and may cause issues in the future if the variable is expected to have a certain value but does not.

Recommendation:
Removing the unused `swapPercentage` variable from the contract is recommended. This will help to avoid any confusion for future developers.

ID	8
Title	Unnecessary Variable Assignment
Path	contracts/UnipilotActiveVault.sol contracts/UnipilotActiveFactory.sol contracts/UnipilotPassiveVault.sol contracts/UnipilotPassiveFactory.sol

Description:

The contract assigns a value to the WETH contract address through the constructor argument _WETH. However, since the WETH address is a well-known constant and is unlikely to change, there is no need to assign it dynamically through the constructor. This introduces unnecessary overhead.

Recommendation:

To address this issue, it is recommended to hardcode the WETH contract address directly into the contract code and make it an immutable state variable. This removes the need for an unnecessary constructor argument.

Developer Response:

The contract suite will be deployed on multiple chains that's why WETH address is dynamic with input.

Auditor Response:

Resolved with no further action.

ID	9
Title	Inconsistent variable naming
Path	contracts/UnipilotActiveVault.sol contracts/UnipilotPassiveVault.sol
Function Name	deposit

Description:
In the `deposit` function, the variable sender is used to hold the value of `_msgSender()`, which refers to the address of the function caller. However, when calling the `TransferHelper.safeTransferETH` function to transfer `ethers` to the recipient, the `msg.sender` is used instead of sender. This can cause confusion and increase the risk of errors during auditing and maintenance.

Recommendation:
To avoid confusion and errors, it is recommended to use the same variable throughout the function to refer to the same address. In this case, it would be better to use the local variable `sender` instead of `msg.sender`.

Developer Response:
Acknowledged. Contract size exceeding issue.

Auditor Response:

Resolved with no further action.

ID	10
Title	Nonconformance with best practices
Path	contracts/UnipilotActiveVault.sol

Description:

In the given smart contract, the state variables `token0`, `token1`, and `unipilotFactory` are declared as private. However, it is observed that these state variables are never updated in the contract. Therefore, it is recommended to declare them as private immutable. Immutable state variables can be initialized only once and can never be changed again, which makes the contract more secure and efficient.

Recommendation:

It is recommended to declare the state variables `token0`, `token1`, and `unipilotFactory` as private immutable instead of private only. By doing so, it will ensure that these variables are only set once and cannot be modified later, reducing the potential attack vectors and making the contract more secure.

Developer Response:

Acknowledged. Contract size exceeding issue.

Auditor Response:

Resolved with no further action.

ID	11
Title	Redundant requirement check
Path	contracts/UnipilotStrategy.sol
Function Name	setBaseTicks

Description:

The function `setBaseTicks` contains three requirement checks to validate the inputs. The first and second requirement checks validate the lengths of `_pools` against `_baseMultiplier` and `_strategyType` respectively. However, the third requirement check also checks the length of `_pools` and `_baseMultiplier`. It is unnecessary to place the third check as the two already validate the length of all three arrays as a transitive property.

Code:

```
function setBaseTicks(
      address[] memory _pools,
      uint16[] memory _strategyType,
      int24[] memory _baseMultiplier
    ) external onlyGovernance {
      require(_pools.length == _baseMultiplier.length);
      require(_pools.length == _strategyType.length);
      require(_baseMultiplier.length == _strategyType.length);

      for (uint256 i = 0; i < _pools.length; i++) {
        activePoolStrategy[_pools[i]][_strategyType[i]] = _baseMultiplier[
          i
        ];
      }
    }
```

Recommendation:

The unnecessary requirement check should be removed to make the code more efficient. Therefore, the third `require` statement can be removed from the code.

Centralization Risks:

ID	12
Title	Overly Centralized Method of Operation
Path	contracts/UnipilotActiveVault.sol, contracts/UnipilotPassiveVault.sol
Function Name	pullLiquidity

Description:

As communicated in the previous audit reports for the Layer 2 contract suite to be deployed on Polygon and Arbitrum for Uniswap V3, the centralization factors exist in the current codebase as well, specifically in the `pullLiquidity` method.

Developer Response:

We are not using the `pullLiquidity` method in the smart contract, if we plan to execute it in the future we will take steps to involve our community to be part of this decision.

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 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 to make any decisions to buy or sell any token, product, service, or other 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.