SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Kei Finance
Date: 15 September, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for Kei Finance
Approved By	Arda Usman | Lead Solidity SC Auditor at Hacken OÜ
Tags	ERC20 token; Presale
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://kei.fi
Changelog	22.08.2023 - Initial Review 15.09.2023 - Second Review

Table of contents

Introduction 4
System Overview 4
Executive Summary 5
Risks 6
Checked Items 7
Findings 10
Critical 10
C01. Data Consistency 10
C02. Invalid Calculations 10
High 11
H01. Unsafe Casting 11
H02. Data Inconsistency 11
H03. Requirements Violation 12
Medium 12
M01. Unchecked Return Value 12
M02. Missing Zero Address Validation 13
M03. Incorrect Usage of Transfer 13
M04. Data Inconsistency 14
Low 14
L01. Inefficient Gas Pattern 14
L02. Style Guide Violation 15
L03. Floating Pragma 15
L04. Missing Zero Address Validation 16
Informational 16
I01. Redundant Declaration 16
I02. Unused Variable 16
I03. Unused Inheritance 17
I04. Replace "" With bytes(0) for Gas Optimization 17
I05. Replace Require Error Strings With Custom Errors for Gas Optimization 17
I06. Unoptimized Loop 17
Disclaimers 19
Appendix 1. Severity Definitions 20
Risk Levels 20
Impact Levels 21
Likelihood Levels 21
Informational 21
Appendix 2. Scope 22

Introduction

Hacken OÜ (Consultant) was contracted by Kei Finance (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Kei Finance is implementing a presale schedule with the following contracts:

- *Presale.sol* – is a presale contract that allows people to participate by purchasing tokens with PRESALE_ASSET token. Purchased tokens are distributed within the Presale contract during the *purchase* function call.
- *IPresale.sol* – interface of the Presale contract.
- IPresaleErrors.sol - interface that stores all custom errors.

Privileged roles

- The owner of the Presale contract can:
    - initialize the Presale contract
    - close the presale
    - set the withdrawTo address
    - transfer ownership

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements:
  - Project overview is detailed.
  - Business logic is provided.
  - Use cases are described and detailed.
  - All interactions are described.
- Technical description is provided:
  - Run instructions are provided.
  - Technical specification is provided.
  - NatSpec is sufficient.

Code quality

The total Code Quality score is **9** out of **10**.

- Deployment instructions description are provided.
- The code largely adheres to the Solidity Style Guide, with a few areas that could benefit from further refinement:
  - Function order is incorrect, private functions should go after internal ones.

Test coverage

Code coverage of the project is **100%** (branch coverage):

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage are present.
- Interactions by several users are tested.

Security score

As a result of the audit, the code contains **1** low severity issue. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.8**. The system users should acknowledge all the risks

summed up in the risks section of the report.

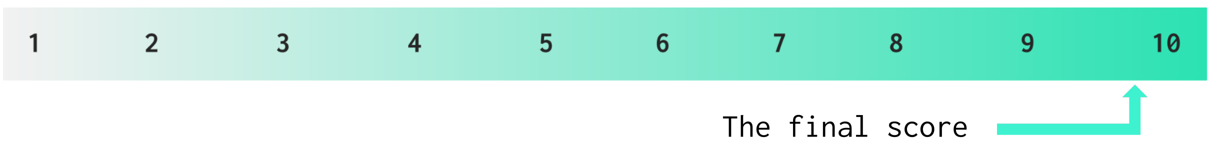

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
22 August 2023	4	4	3	2
15 September 2023	1	0	0	0

Risks

- The system has external calls in the *purchase()* function, which creates a reentrancy risk.
- The payment token contract is out of the scope of this review, indicating that it has not been assessed by Hacken. Consequently, assurance regarding the contract's safety or security cannot be provided.
- The owner can stop the Presale process at any time using the *close()* function. Thus, it creates a centralization risk.
- The contract employs an *initialize()* function intended for setting its initial states. While it uses a mechanism commonly associated with upgradable contracts, this specific contract is not upgradable. The initialize function, protected by the onlyOwner modifier, is crucial to be invoked post-deployment to set the contract in its intended operational state. Failure to call this function results in the contract remaining in an uninitialized state, which can cause unintended behaviors or inhibit users from interacting with it as expected. It is essential to verify that initialize was executed appropriately after deployment before any user interaction.

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed	
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	
Unchecked Call Return Value	The return value of a message call should be checked.	Passed	
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed	
SELFDESTRUCT Instruction	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Not Relevant	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Passed	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed	
Signature Unique Id	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Passed	
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Not Relevant	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	

Flashloan Attack	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used. Contracts shouldn't rely on values that can be changed in the same transaction.	Passed	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Passed	
Gas Limit and Loops	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed	
Style Guide Violation	Style guides and best practices should be followed.	Failed	L02
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed	
Secure Oracles Usage	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Passed	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Findings

■■■■ Critical

C01. Data Consistency

Impact	High
Likelihood	High

The current implementation gathers assets obtained from presale purchases and reserves user allocations for future distribution through a central mechanism that lies beyond the scope of the current project. The system encompasses two round types:

Liquidity round type: a user pays a specified amount referred to as X, and the *_c.userAllocation* is incremented by adding Y/2

Tokens round type: the *_c.userAllocation* is incremented by adding Y for the same paid X amount.

So, when the round type is Liquidity, the user's allocation is determined by splitting the due amount into two. Consequently, the *$userTokensAllocated* variable is updated by adding Y/2. There is no mechanism in place to distinguish whether a user allocation was generated using the Liquidity round type or otherwise. This leads to a scenario where users who acquired tokens during the Liquidity round experience a reduction of half their earnings.

Path: ./contracts/Presale.sol: *_purchase()*

Recommendation: Implement a distinct flag or identifier within the system to differentiate user allocations generated from the Liquidity round type, and prevent any reduction in earnings for users who staked tokens during that round.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed (Liquidity round type is removed from the platform) (Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

C02. Invalid Calculations

Impact	High
Likelihood	High

During purchasing with Ether, DAI and USDC the conversion from tknBits to tokens does not happen, which leads to incorrect number of USD.

Path: ./contracts/Presale.sol: *purchaseDAI()*, *purchaseUSDC()*, *purchase()*, *ethToUsd()*, *ethToTokens()*

Proof of Concept: Kei Finance Test Cases

Recommendation: Use the necessary precision number.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed (Purchasing with different stable coins feature is removed. Users can buy only with the PRESALE_ASSET token.) (Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

■■■ High

H01. Unsafe Casting

Impact	High
Likelihood	Medium

When receiving data from price oracles, the expected return is in int256, which can have negative values, as opposed to uint256, which only stores positive values. Price oracles often return a negative value to indicate that the data is not valid anymore.

The referred contracts perform an unsafe casting, transforming the oracle result in uint256 without checking if the value is negative. This leads to the same behavior of underflows, causing the int256 number to flip to the highest uint256 number.

Path: ./contracts/Presale.sol: *ethPrice()*

Recommendation: Put a statement that checks if the returned price is bigger than zero to prevent reverts during casting.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed (Purchasing with Ethereum is removed) (Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

H02. Data Inconsistency

Impact	High
Likelihood	Medium

Total USD allocation of a user to be saved for the transaction receipt is only recorded if the round type is ‘Liquidity’. That creates inconsistency and it may affect the distribution of the presale tokens if this value is going to be used.

Path: ./contracts/Presale.sol: *_purchase()*

Recommendation: Update the total USD allocation for both ‘Liquidity’ and ‘Tokens’ round types.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed (Liquidity round type is removed from the platform) (Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

H03. Requirements Violation

Impact	High
Likelihood	Medium

There is a mismatch between the documentation and the implementation.

The documentation says that the number of price levels will be 7 and the minimum/maximum contribution is set to 200$ and 50000$ respectively, the documentation also has a specific range for tokenPrice at each round, but the owner has the ability to change this data using the functions *initialize()*.

Another contradiction locates on line 282. Although it is stated that the minimum contribution is set at $200, with a maximum limit of $50,000 in the documentation, this is not validated in the require statement. Users can purchase even with 0 amount.

Path: ./contracts/Presale.sol: *initialize()*

Recommendation: Consider adding the data boundaries.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Mitigated (Setter functions of related variables are removed from the platform. There are no restrictions on maintaining the price at $7 and allowing contributions only between $200 and $50,000. However, the owner is now able to declare the following variables - withdrawTo, minDepositAmount, maxUserAllocation, startDate, price, and allocation - only once after the contract deployment.)
(Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

Medium

M01. Unchecked Return Value

Impact	High
Likelihood	Low

The function *ethPrice()* fetches the asset price from a Chainlink aggregator using the *latestRoundData()* function.

However, there are no checks on *timeStamp* or roundId return values, resulting in stale prices. The oracle wrapper calls out to a

Chainlink oracle receiving *latestRoundData*(). The returned updatedAt timestamp is not checked or considered at all.

Path: ./contracts/Presale.sol: *ethPrice()*

Recommendation: Consider adding checks on the return data with proper revert messages if the price is stale or the round is incomplete for example:

```
(uint80 roundID, int256 price, , uint256 timestamp, uint80
answeredInRound) = AggregatorV3Interface(ORACLE).latestRoundData();

require(timestamp >= block.timestamp-3600, "...");
require(answeredInRound >= roundID, "Stale price");
require(price > 0, "...");
```

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed (Purchasing with Ethereum is removed) (Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

M02. Missing Zero Address Validation

Impact	High
Likelihood	Low

All purchase functions save the allocations for the given address parameters. *address account* parameter is not checked whether it is zero address or not.

This may lead to unwanted 0x0 calls and user may lose their allocations and assets.

Path: ./contracts/Presale.sol: *purchase()*, *purchaseUSDC()*, *purchaseDAI()*

Recommendation: Implement zero address checks.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed
(Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

M03. Incorrect Usage of Transfer

Impact	Medium
Likelihood	High

The Presale contract has a *_send()* function, which uses a *transfer()* method. It only has 2300 Gas (and throws errors). The execution will fail if the sender will be a Contract and will have additional code in the *receive/fallback* functions, which requires extra Gas. This can lead to reduce interaction with the system.

Path: ./contracts/Presale.sol: *_send()*

Recommendation: Replace *transfer()* with *call()* and implement the return value check.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed (Purchasing with Ethereum is removed)
(Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

M04. Data Inconsistency

Impact	Medium
Likelihood	Medium

While USDC and DAI are intended to maintain a 1:1 value ratio with the US Dollar (USD), it is important to note that the actual value of 1 USDC or 1 DAI can occasionally differ from 1 USD. Price fluctuations can cause different values of USDC/DAI than 1 USD.

Path: ./contracts/Presale.sol: *purchaseUSDC()*, *purchaseDAI()*

Recommendation: Do not assume that 1 USD is equal to the 1 USDC/DAI and use oracle to bring USDC/DAI price.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed (Purchasing with DAI/USDC was removed)
(Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

Low

L01. Inefficient Gas Pattern

Impact	Low
Likelihood	Medium

There is no check implemented to validate if the user has enough funds to purchase presale tokens.

Implementing zero balance check will prevent spending unnecessary Gas for users when they accidentally start a transaction with 0 amount of tokens.

Path: ./contracts/Presale.sol: *purchase()*, *purchaseDAI()*, *purchaseUSDC()*

Recommendation: Implement checks to help users to spend unnecassary Gas.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Mitigated (Also implement a statement that checks if the user’s payment token balance (IERC20(payment) is equal or greater than the parameter assetAmount.) (Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

L02. Style Guide Violation

Impact	Low
Likelihood	Low

Official style guide is violated. Especially pay attention to line length, order of layout and variable naming.

Path: ./contracts/Presale.sol

Recommendation: Follow the official Solidity guidelines. https://docs.soliditylang.org/en/v0.8.13/style-guide.html

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Reported (Max Line length is fixed. Naming inconvention and function ordering violation are still present) (Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

L03. Floating Pragma

Impact	Low
Likelihood	Low

The project uses floating pragma ^0.8.9.

Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Path: ./contracts/IPresale.sol

Recommendation: Consider locking the pragma version whenever possible and consider deploying the code with any of the following versions: 0.8.19, 0.8.20.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Mitigated (IPresale contract is inherited only by the Presale contract, which has not a floating pragma. Therefore, all the contracts in the scope will be compiled with the 0.8.19 Solidity version.) (Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

L04. Missing Zero Address Validation

Impact	Medium
Likelihood	Low

Allocation functions save the allocations for the given address parameters. *address account* parameter is not checked whether it is zero address or not.

This may lead to unwanted 0x0 calls and owner can allocate zero addresses.

Path: ./contracts/Presale.sol: *allocate(), constructor()*

Recommendation: Implement zero address checks.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed
(The Customer removed the 'allocate' function.) (Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

Informational

I01. Redundant Declaration

On line 316, *_c.remainingUSD* variable is already updated in the previous if statement(line 308).

Path: ./contracts/Presale.sol: *_purchase()*

Recommendation: Remove the redundant declaration on line 316.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed

(Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

I02. Unused Variable

The variable *USD_PRECISION* is declared, but its value is never used. Redundant declarations spend unnecessary Gas and decrease code readability.

Path: ./contracts/Presale.sol

Recommendation: Remove the unused variable.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed

(Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

I03. Unused Inheritance

The Presale contract inherits the ReentrancyGuard, but there is no way to use its functionalities.

Path: ./contracts/Presale.sol

Recommendation: Remove or implement the unused inheritance.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed
(Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

I04. Replace "" With *bytes(0)* for Gas Optimization

There is an empty string, which was passed to the *purchase()* function via the *receive()* function. It can be substituted with the *new bytes(0)* for Gas optimization.

Path: ./contracts/Presale.sol

Recommendation: Consider the Gas optimization for the aforementioned code.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed
(Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

I05. Replace Require Error Strings With Custom Errors for Gas Optimization

The functions *_setConfig()*, *_purchase()* use error string messages for reverting behavior. They can be changed with the custom errors for Gas optimization.

Path: ./contracts/Presale.sol: *_setConfig(), _purchase()*

Recommendation: Consider optimizing the code using the custom errors.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Fixed
(Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

I06. Unoptimized Loop

Since Solidity 0.8.0, underflow/overflow checks are automatic, they can be disabled for Gas optimization.

Path: ./contracts/Presale.sol: *_setRounds()*

Recommendation: Consider loop optimization via the unchecked{} keyword.

Found in: 0ea9e6ff7aa5aad8966a0d59420300b262108f13

Status: Mitigated (The function *_setRounds()* is now can be called only once, the emphasis on Gas optimization diminishes)
(Revised commit: ea0fb67db4f129e91a85541fbd48a3003f19205f)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in 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).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

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.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

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 vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	Critical	High	Medium
Medium Likelihood	High	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/Kei-Finance/presale-contract
Commit	0ea9e6ff7aa5aad8966a0d59420300b262108f13
Whitepaper	Link
Requirements	Link
Technical Requirements	Natspec
Contracts	File: contracts/Presale.sol SHA3: 27524a44a480bdff51659a546977da2b4d1945f4e5d1085ce6a5c47e871d223f File: contracts/IPresale.sol SHA3: dbd5315348298875ed8d0edb21dea226c10c97bd1d1894076f40fb65fdc9e581

Second review scope

Repository	https://github.com/Kei-Finance/presale-contract
Commit	ea0fb67db4f129e91a85541fbd48a3003f19205f
Whitepaper	Link
Requirements	Link
Technical Requirements	Natspec
Contracts	File: contracts/interfaces/IPresaleErrors.sol SHA3: 4d45cc9b672ed57036c3b84eef610c1cc1b79a02f8bd319354648e8b16ade089 File: contracts/interfaces/IPresale.sol SHA3: 1b6d013523e0d184b4f6d0ae54849bb731956e9ef34c4a0ca6b3279d55ab125f File: contracts/Presale.sol SHA3: 83137a6178e007b6c4cf796857b65c7d040819dedfb63c6d89a43ac67def26db