HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: 2cimple
Date: 20 Oct, 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 2cimple
Approved By	Kaan Caglan | Senior Solidity SC Auditor at Hacken OÜ
Tags	ERC20 token
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://www.2cimple.com/
Changelog	08.09.2023 - Initial Review 20.10.2023 - Second Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by 2cimple (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

2Cimple is a smart contract is a decentralized application (DApp):

- *DeedzCoin* — ERC-20 token that implements a locking mechanism implementing ERC1132 standard. Mints all total supply to a supplier role.
  It has the following attributes:
  - Name: DEEDZ COIN
  - Symbol: DEEDZ
  - Decimals: 18
  - Total supply: 5 * 10^26 tokens.

Privileged roles

- Owner: The owner can transfer the supplier Role using the `transferSupplierRole()` function.
- Supplier: The supplier can transfer and lock a specified amount of tokens for a specified reason and time as well as until a specific time using functions `transferWithLock()` and `transferWithLockActualTime()` respectively.

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 are provided:
  - Project overview is detailed.
  - All roles in the system are described.
  - Use cases are described and detailed.
  - For each contract all futures are described.
  - All interactions are described.
- The technical requirements are provided:
  - NatSpec is provided.
  - Technical specification is provided.
  - The description of the development environment is provided.

Code quality

The total Code Quality score is **7** out of **10**.

- Deployment instructions are provided.
- The Solidity Style Guide is not completely adhered to.
- Best practices are not followed: L01, I03.

Test coverage

Code coverage of the project is **97.83%** (branch coverage).

- Not all branches are covered with tests.

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: **9.3**. The system users should acknowledge all the risks summed up in the risks section of the report.

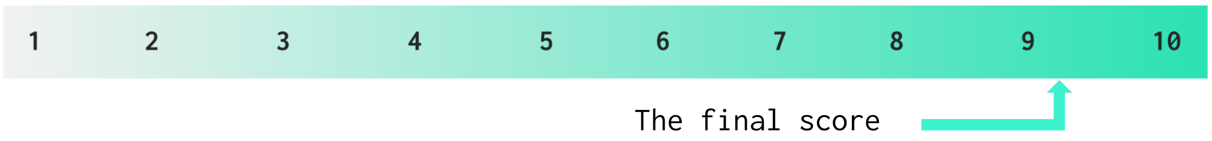

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
8 Sep 2023	2	0	1	1
20 Oct 2023	1	0	0	0

Risks

- All tokens are minted to a single address using a multisignature wallet. While this enhances security, the safeguarding of the supply still hinges on the secure storage of multiple keys. Additionally, it poses a risk of centralization, as the supplier has the potential to distribute tokens in an unfair manner.
- The whitepaper mentions that "Users who lock their tokens in the DeedzCoin smart contract for a specific period are eligible to receive bonus tokens." However, in the current version of the project, only the supplier has the capability to lock tokens. Additionally, there is no existing reward mechanism to distribute bonus tokens. This inconsistency can lead to misunderstandings and potential disputes with users who expect to earn bonus tokens upon locking.

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.	Not Relevant	
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.	Not Relevant	
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.	Passed	
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.	Not Relevant	
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	L01, I01, I03
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.	Not Relevant	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. The 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. Invalid Calculations

Impact	**High**
Likelihood	**High**

The `_transferSupplierRole()` function in the provided code snippet has an unintended balance assignment order, which results in a critical flaw. The balance of the *oldSupplier* is transferred to the *newSupplier*, but immediately afterward, the new balance of the *newSupplier* is overwritten by an arithmetic operation. Specifically, the function erroneously sets the *newSupplier* balance to *safeAdd(balanceOf(oldSupplier), balances[supplier()])* after having transferred the entire balance of *oldSupplier* to *newSupplier*. Given that the balance of the *oldSupplier* is set to 0 after the transfer, this effectively resets the balance of the newSupplier to 0, regardless of its previous state. This unintended behavior can lead to loss of funds stored in the contract.

Path: ./contracts/DeedzCoin.sol: `_transferSupplierRole()`;

POC Steps:

1. The owner deploys the contract, initializing Alice as the supplier. During contract deployment, Alice's balance is incremented by the totalSupply value. Simultaneously, a totalSupply amount of tokens are minted for Alice.
2. Later on, the owner decides to transfer the supplier role from Alice to Bob. To execute this, the owner calls the `transferSupplierRole()` function, providing Bob's address.
3. Within the `_transferSupplierRole()` function, the entire token balance of Alice is transferred to Bob. However, immediately after this, the function attempts to update Bob's balance in the balances mapping using Alice's balanceOf value. Since Alice's entire balance was already transferred to Bob, this results in Bob's balance being erroneously set to 0 in the mapping.

```
>>> deedzCoin.supplier()
'0x33A4622B82D4c04a53e170c638B944ce27cffce3'
```

```
>>> deedzCoin.supplier() == user1
True
>>> deedzCoin.balances(user1)
5000000000000000000000000000
>>> deedzCoin.transferSupplierRole(user2, {'from': owner})
Transaction sent:
0xc675ccd635f716bcdcb40108d407e39af0dd7303e263811d45d78e04be8ab8
65
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1
  DeedzCoin.transferSupplierRole confirmed   Block: 2   Gas
used: 39707 (0.33%)

<Transaction
'0xc675ccd635f716bcdcb40108d407e39af0dd7303e263811d45d78e04be8ab
865'>
>>> deedzCoin.balances(user1)
0
>>> deedzCoin.balances(user2)
0
```

Recommendation: To rectify this vulnerability, the sequence of operations should be reordered and simplified. First, the balance addition should occur, followed by the balance reset of the oldSupplier. This ensures that the newSupplier retains the combined balance. Here is a possible correction:

```
    function _transferSupplierRole(address newSupplier) internal
virtual {
        address oldSupplier = supplier();
        uint256 oldSupplierBalance = balanceOf(oldSupplier);

        setSupplier(newSupplier);
        emit SupplierRoleTransferred(oldSupplier, supplier());

        balances[newSupplier] = oldSupplierBalance;
        _transfer(oldSupplier, newSupplier, oldSupplierBalance);
        balances[oldSupplier] = 0;
    }
```

Found in: 9f57d71

Status: Fixed (Revised commit:4e3c236)

■■■ High

H01. Redundant *balances* Mapping Overlook

Impact	Medium
Likelihood	High

The DeedzCoin smart contract introduces a public mapping named *balances* to store the balance of tokens for each address. This is redundant as the contract inherits from the *ERC20* contract, which already provides a function named *balanceOf()* to retrieve an address's balance and maintains a private mapping *_balances* to store token balances for addresses.

Moreover, the contract does not override or update the functions (*transfer*, *transferFrom*) from the *ERC20*, which means, when those functions are called, only the *_balances* mapping from the *ERC20* will be updated, and not the *balances* mapping of the *DeedzCoin* contract. This could lead to major inconsistencies in the token balance data.

Path: ./contracts/DeedzCoin.sol: *balances()*, *transferSupplyRole()*;

Recommendation:

1. Remove Redundancy: Completely remove the balances mapping from the DeedzCoin contract. It is not only redundant but also potentially error-prone.
2. Use Inherited Functions and Variables: Directly use the *balanceOf()* function from the inherited ERC20 contract to fetch the balance for any address. If needed to perform internal operations, use the _balances mapping from the ERC20 contract. This ensures that operations are consistent with the standard functions provided by the ERC20 contract.
3. Review and Testing: It is crucial to have a thorough review of the smart contract. Every added piece of functionality needs to be reviewed in the context of the existing functions, especially when inheriting from well-established contracts. After any changes, conduct comprehensive testing, including edge cases, to ensure that the contract behaves as expected.

Found in: 9f57d71

Status: Fixed (Revised commit:4e3c236)

Medium

No medium severity issues were found.

Low

L01. Missing Zero Address Validation

Impact	Low
Likelihood	Medium

The smart contract does not validate for the zero address (0x0) when handling address parameters. This oversight could inadvertently trigger unintended external calls to the 0x0 address, which might lead to undesired behaviors or potential loss of funds.

Path: ./contracts/DeedzCoin.sol: *setSupplier();*

Recommendation: To safeguard against unintended interactions with the zero address, it is advised to integrate the following best practices:

1. Validation Checks: Implement validation checks at the start of functions or operations that involve address parameters. These checks should confirm that the address is not the zero address (0x0) before proceeding with further execution.

2. Reusable Modifier: Consider creating a reusable modifier such as isNotZeroAddress(address _address), which can be applied to functions to ensure that they are not passed or dealing with a zero address. This not only enhances code reusability but improves clarity.

3. Error Handling: If an address validation fails, ensure that the contract emits a clear and meaningful error message. This assists in debugging and alerts users to potential issues with their transactions.

4. Testing: After implementing the above changes, it is crucial to conduct comprehensive testing to ensure the smart contract behaves as expected and does not interact with the zero address.

By adhering to these recommendations, it is possible to reduce the risk associated with unintended external calls to the 0x0 address and enhance the robustness of smart contracts.

Found in: 9f57d71

Status: Reported (Revised commit:4e3c236).

L02. Insufficient Access Control on Token Management Functions

Impact	Low
Likelihood	Medium

The *DeedzCoin* smart contract contains several functions (*extendLock*, *increaseLockAmount*, *unlock*, and *lock*) that, according to the project's documentation, are meant to be restricted only to suppliers. However, in the actual contract implementation, these functions lack the *onlySupplier* modifier. This discrepancy can lead to potential misuse or misinterpretation, even if the direct risk posed by the current setup is low.

Specifically, while functions such as *extendLock()*, *increaseLockAmount()*, and *lock()* operate using *msg.sender* (and thus cannot directly be abused to manipulate another user's tokens), the *unlock* function can potentially be called by any actor for any address. While this does not pose a direct financial risk since the tokens are transferred to the specified address, it could potentially allow malicious actors to interfere in the token's intended flow and mechanics.

Path: ./contracts/DeedzCoin.sol: *extendLock(), increaseLockAmount(), unlock(), lock();*

Recommendation:

1) Consistency Between Implementation and Documentation: Address the discrepancy between the smart contract implementation and the documentation. It is crucial to ensure that the code aligns with the documented intent to prevent confusion and misuse.
2) Redefine Access Control: Decide on the appropriate level of access control for each function. If certain functions are truly intended to be used only by the supplier:
    a) Consider adding an `*address to*` parameter to these functions.
    b) Apply the onlySupplier modifier to ensure that only the supplier can execute them.
3) Reassess unlock Behavior: For the unlock function, if the ability to unlock tokens for any address remains the intended behavior, make this clear in the documentation. If not, restrict the ability as per the intended design.
4) Update and Clarify Documentation: After making any necessary code changes, update the documentation to match the contract's behavior accurately. This ensures that users and developers have a clear understanding of how the contract should work.

Found in: 9f57d71

Status: Fixed (Revised commit:4e3c236)

Informational

I01. Solidity Style Guide Violation

Contract readability and code quality are influenced significantly by adherence to established style guidelines. In Solidity programming, there exist certain norms for code arrangement and ordering. These guidelines help to maintain a consistent structure across different contracts, libraries, or interfaces, making it easier for developers and auditors to understand and interact with the code.

The suggested order of elements within each contract, library, or interface is as follows:

- Type declarations
- State variables
- Events
- Modifiers
- Functions

Functions should be ordered and grouped by their visibility as follows:

- Constructor
- Receive function (if exists)
- Fallback function (if exists)
- External functions
- Public functions
- Internal functions
- Private functions

Within each grouping, view and pure functions should be placed at the end.

Furthermore, following the Solidity naming convention and adding NatSpec annotations for all functions are strongly recommended. These measures aid in the comprehension of code and enhance overall code quality.

Path: ./contracts/DeedzCoin.sol

Recommendation: Consistent adherence to the official Solidity style guide is recommended. This enhances the readability and maintainability of the code, facilitating seamless interaction with the contracts. Providing comprehensive NatSpec annotations for functions and following Solidity's naming conventions further enrich the quality of the code.

Found in: 9f57d71

Status: Reported (Revised commit:4e3c236).

I02. Use Custom Errors

Using custom errors instead of revert strings can significantly reduce Gas costs, especially when deploying contracts. Prior to Solidity v0.8.4, revert strings were the only way to provide more information to users about why an operation failed. However, revert strings are expensive, and it is difficult to use dynamic information in them. Custom errors, on the other hand, were introduced in Solidity v0.8.4 and provide a Gas-efficient way to explain why an operation failed.

Path: ./contracts/DeedzCoin.sol

Recommendation: It is recommended to use custom errors instead of revert strings to reduce gas costs, especially during contract deployment. Custom errors can be defined using the error keyword and can include dynamic information.

Found in: 9f57d71

Status: Fixed (Revised commit:4e3c236)

I03. Redundant Check

The DeedzCoin smart contract has functions *transferWithLock()* and *transferWithLockActualTime()*, which have double checks on the number of already locked tokens. This pattern uses extra Gas and should be reduced.

Path: ./contracts/DeedzCoin.sol

Recommendation: Remove redundant checks that duplicate the validation efforts of other existing checks.

Found in: 9f57d71

Status: Reported (Revised commit:4e3c236)

I04. Unused Variable

The contract DeedzCoin has a mapping named allowed, which is used nowhere. It adds additional deployment costs and, hence, should be deleted.

Path: ./contracts/DeedzCoin.sol: *allowed();*

Recommendation: Remove redundant variables to save Gas on deployment and increase code quality.

Found in: 9f57d71

Status: Fixed (Revised commit:4e3c236)

I05. Redundant Virtual Keyword

The contract DeedzCoin has virtual keywords in the function definition. These functions are not being overridden in audit scope and this functionality is not described in the documentation.

Path:./contracts/DeedzCoin.sol: *_transferSuppliedRole()*, *transferSupplierRole()*, *lock()*, *transferWithLock()*, *tokensLocked()*, *tokensLockedAtTime()*, *totalBalanceOf()*, *extendLock()*, *increaseLockAmount()*, *tokensUnlockable()*, *unlock()*, *getUnlockableToken();*

Recommendation: Remove redundant virtual keywords to save Gas on deployment and increase code quality.

Found in: 9f57d71

Status: Fixed (Revised commit:4e3c236) (The redundant virtual keywords have been deleted from the aforementioned functions).

I06. Redundant SafeMath Usage In Solidity

In Solidity versions prior to 0.8.0, integer overflow and underflow were major concerns, leading to potential vulnerabilities in smart contracts. To address this, the SafeMath library was widely used to ensure safe arithmetic operations. However, starting from Solidity version 0.8.0, overflow and underflow checks were integrated into the language itself, rendering the explicit usage of SafeMath for these checks redundant.

Path: ./contracts/DeedzCoin.sol: contract SafeMath {}

Recommendation: For smart contracts written in Solidity version 0.8.0 and above, developers should avoid using SafeMath for basic arithmetic operations. Relying on the built-in overflow and underflow checks will save Gas, optimize contract execution, and simplify code. If SafeMath functionalities are used for reasons other than overflow/underflow checks, consider alternatives or ensure that the usage is justified.

Found in: 9f57d71

Status: Fixed (Revised commit:4e3c236)

I07. Inefficient Event Emitting

The smart contract function `_transferSupplierRole` uses a state variable call for the new supplier when emitting the `SupplierRoleTransferred` event, instead of directly using the `newSupplier` parameter passed to the function. Using the parameter directly is more gas-efficient.

```
function _transferSupplierRole(address newSupplier) internal {
address oldSupplier = supplier();
setSupplier(newSupplier);
emit SupplierRoleTransferred(oldSupplier, supplier());
_transfer(oldSupplier, newSupplier, balanceOf(oldSupplier));
}
```

Path: ./contracts/DeedzCoin.sol

Recommendation: Use the original function parameter `newSupplier` instead of the state variable when emitting the event. The correct way to emit the event is:

emit SupplierRoleTransferred(oldSupplier, newSupplier);

Found in: 4e3c236

Status: New

I08. Code Duplication

The smart contract contains two functions `transferWithLock` and `transferWithLockActualTime` that are almost identical in logic and functionality. The only difference observed is the calculation of `validUntil` time. In `transferWithLock`, the time to lock is added to the current `block.timestamp`, whereas in `transferWithLockActualTime`, the absolute time for unlocking is directly provided.

Path: ./contracts/DeedzCoin.sol

Recommendation: To improve maintainability and reduce potential code errors, consider consolidating these two functions into one. An additional parameter or flag can be introduced to determine whether the provided time is absolute or relative, thus determining how `validUntil` should be calculated.

Found in: 4e3c236

Status: New

I09. Unnecessary Boolean Return

The smart contract contains four functions (`increaseLockAmount`, `transferWithLockActualTime`, `transferWithLock`, and `extendLock`) that all return a boolean value at the end of their execution. However, in all these functions, the only possible return value is `true`. In cases of failure or error, these functions use the `revert` mechanism, which means they do not proceed to the end and do not return any value. Hence, the returning of `true` is redundant, as users can deduce the success of the function by checking if the transaction has not been reverted.

Path: ./contracts/DeedzCoin.sol

Recommendation: Remove the boolean return type and the `return true;` statements from these functions. Users can determine the success of the functions by checking the transaction status, making the boolean return unnecessary.

Found in: 4e3c236

Status: New

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/experiencenano/DeedzCoin
Commit	9f57d71
Whitepaper	N/A
Requirements	README
Technical Requirements	README
Contracts	File: contracts/DeedzCoin.sol SHA3: 4eaa2284a2af129e793ffd763101e67a5be3d3822b2ad3a00fffe911810dcf40

Second review scope

Repository	https://github.com/experiencenano/DeedzCoin
Commit	4e3c236
Whitepaper	N/A
Requirements	README
Technical Requirements	README
Contracts	File: contracts/DeedzCoin.sol SHA3: 97e5d0c474a8cbb0e4d2c692ff841807c2553a4a2e9c286f559054ee