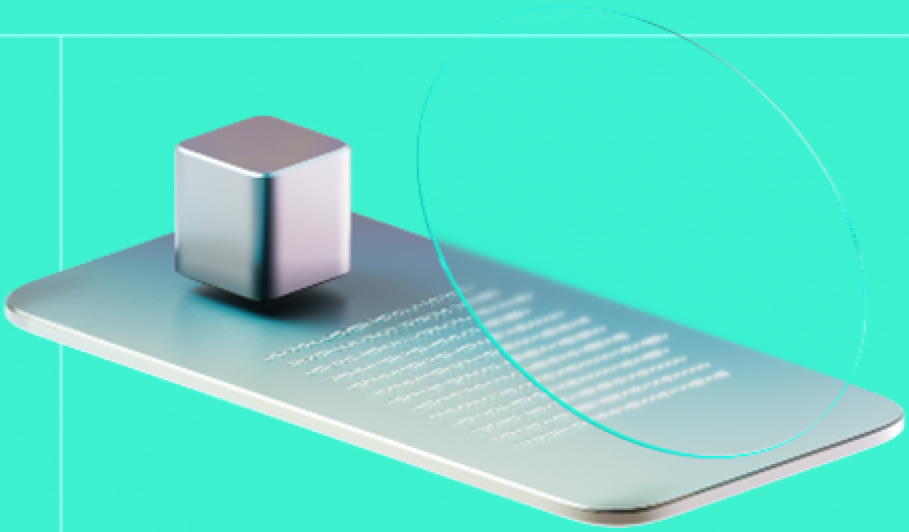
Smart Contract Code Review And Security Analysis Report



Customer: Commoncentz

Date: 16/01/2024

We express our gratitude to the Commoncentz team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Commoncentz is an ERC-20 Fiat Token project. It has regular ERC-20 token functionalities with blacklist feature. Owner can add malicious users to blacklist so they will not be able to transfer their tokens.

Platform: EVM

Language: Solidity

Tags: ERC-20,Fiat

Timeline: 05.01.2024 - 15.01.2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/commoncentz/commoncentz-contract
Commit	13d8809

Audit Summary

10/10	8/10	0%	6/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.2/10

The system users should acknowledge all the risks summed up in the risks section of the report

1	1	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	0
Medium	0
Low	1

Vulnerability	Status
F-2024-0361 - Non Disabled Implementation Contract	Fixed

Document

Name	Smart Contract Code Review and Security Analysis Report for Commoncentz
Audited By	Kaan Caglan, Seher Saylik
Approved By	Ataberk Yavuzer
Website	http://commoncentz.com/
Changelog	08/01/2024 - Preliminary Report
	16/01/2024 - Final Report

Table to Contents

System Overview

Commoncentz is a Fiat Token project with the following contracts.

FiatTokenV1 - Simple ERC-20 like token that has a blacklist feature.

FiatTokenV2 - Another token which has exactly same functionality with **FiatTokenV1.**

Privileged roles

- **Owner**: Has the ability to stop the marketplace and perform critical administrative functions.
- **MasterMinter**: Can configure and remove minters, as well as manage the minting allowance.
- **Blacklister**: Can blacklist or unblacklist addresses, controlling who can participate in the network.

Executive Summary

This report presents an in-depth analysis and scoring of the Customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **6** out of **10**.

- Functional requirements are not provided.
- Natspec is sufficient.
- Technical description is not provided.

Code quality

The total Code Quality score is **8** out of **10**.

- The code does not follow the Solidity best practices.

Test coverage

Code coverage of the project is **0%** (branch coverage).

- Coverage tool could not be run because there is no test case provided.

Security score

Upon auditing, the code was found to contain **0** critical, **0** high, **0** medium, and **1** low severity issue, leading to a security score of **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the Customer's smart contract yields an overall score of **9.2.** This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- The black lister role in the project has the right to **blacklist any user address and burn the tokens** they have.

Findings

Vulnerability Details

F-2024-0361 - Non Disabled Implementation Contract - Low

Description: The upgradeable contracts do not disable initializers in the constructor, as recommended by the imported **Initializable** contract. This means that anyone can call the initializer on the implementation contract to set the contract variables and assign the roles. To reduce the potential attack surface, `_disableInitializers` in the constructor needs to be called.

Assets:

- FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract]

Status: Fixed

Classification

Severity: Low

Recommendations

Recommendation: Build a constructor function in the upgradeable contracts that calls the `_disableInitializers()` function.

Remediation (Revised commit: 77bf6da): The Commoncentz team fixed this issue by adding `_disableInitializers();` function to the constructor.

External References:

- Openzeppelin

Observation Details

F-2024-0355 - Revert String Size Optimization - Info

Description: Shortening the revert strings to fit within 32 bytes will decrease deployment time and decrease runtime Gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional `mstore`, along with additional overhead to calculate memory offset, etc.

Code Location

```
Path: ./contracts/v1/FiatTokenV1.sol:
    * FiatTokenV1.sol#78
    * FiatTokenV1.sol#82
    * FiatTokenV1.sol#86
    * FiatTokenV1.sol#105
    * FiatTokenV1.sol#127
    * FiatTokenV1.sol#128
    * FiatTokenV1.sol#131
    * FiatTokenV1.sol#148
    * FiatTokenV1.sol#228
    * FiatTokenV1.sol#229
    * FiatTokenV1.sol#253
    * FiatTokenV1.sol#289
    * FiatTokenV1.sol#290
    * FiatTokenV1.sol#291
    * FiatTokenV1.sol#341
    * FiatTokenV1.sol#342
    * FiatTokenV1.sol#353
    * FiatTokenV1.sol#362
    * FiatTokenV1.sol#374
    * FiatTokenV1.sol#386
    * FiatTokenV1.sol#420
```

Assets:

- FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract]

Status: Accepted

Recommendations

Recommendation: To optimize Gas usage in Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of Gas used during deployment and runtime when the revert condition is met.

F-2024-0356 - Custom Errors in Solidity for Gas Efficiency - Info

Description:

Starting from Solidity version 0.8.4, the language introduced a feature known as **Custom Errors**. These custom errors provide a way for developers to define more descriptive and semantically meaningful error conditions without relying on string messages. Prior to this version, developers often used the `require` statement with string error messages to handle specific conditions or validations. However, every unique string used as a revert reason consumes Gas, making transactions more expensive.

Custom errors, on the other hand, are identified by their name and the types of their parameters only, and they do not have the overhead of string storage. This means that when using custom errors instead of `require` statements with string messages, the Gas consumption can be significantly reduced, leading to more gas-efficient contracts.

Code Location

```
Path: ./contracts/v1/FiatTokenV1.sol
    * FiatTokenV1.sol#78
    * FiatTokenV1.sol#82
    * FiatTokenV1.sol#86
    * FiatTokenV1.sol#105
    * FiatTokenV1.sol#127
    * FiatTokenV1.sol#128
    * FiatTokenV1.sol#131
    * FiatTokenV1.sol#148
    * FiatTokenV1.sol#228
    * FiatTokenV1.sol#229
    * FiatTokenV1.sol#253
    * FiatTokenV1.sol#289
    * FiatTokenV1.sol#290
    * FiatTokenV1.sol#291
    * FiatTokenV1.sol#341
    * FiatTokenV1.sol#342
    * FiatTokenV1.sol#353
    * FiatTokenV1.sol#362
    * FiatTokenV1.sol#374
    * FiatTokenV1.sol#386
    * FiatTokenV1.sol#420
```

Assets:

- FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract]

Status: Accepted

Recommendations

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.

External References:

- Custom Errors

F-2024-0357 - Avoid Using State Variables Directly in `emit` for Gas Efficiency - Info

Description: In Solidity, emitting events is a common way to log contract activity and changes, especially for off-chain monitoring and interfacing. However, using state variables directly in `emit` statements can lead to increased Gas costs. Each access to a state variable incurs Gas due to storage reading operations. When these variables are used directly in `emit` statements, especially within functions that perform multiple operations, the cumulative Gas cost can become significant. Instead, caching state variables in memory and using these local copies in `emit` statements can optimize Gas usage.

Code Location

```
Path: ./contracts/v1/FiatTokenV1.sol

emit MasterMinterChanged(masterMinter);

emit BlacklisterChanged(blacklister);
```

Parameters `_newBlacklister` and `_newMasterMinter` can be used for event emitting instead of state variables.

Assets:

- FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract]

Status: Accepted

Recommendations

Recommendation: To optimize Gas efficiency, cache state variables in memory when they are used multiple times within a function, including in `emit` statements.

F-2024-0358 - Internal functions only called once can be inlined to save Gas - Info

Description: If an internal function is only used once, there is no need to modularize it, unless the function calling it would otherwise be too long and complex. Not inlining costs 20 to 40 Gas because of two extra JUMP instructions and additional stack operations needed for function calls.

```
function _approve(address owner, address spender, uint256 value) internal {
```

Internal function `_approve` is only called in function `approve`.

Assets:

- FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract]

Status: Accepted

Recommendations

Recommendation: Get rid of unnecessary internal `_approve` function.

F-2024-0359 - Owner Can Renounce Ownership - Info

Description:

The smart contract under inspection inherits from the `Ownable` library, which provides basic authorization control functions, simplifying the implementation of user permissions. While the contract allows for the transfer of ownership to a different address or account, it also retains the default `renounceOwnership` function from `Ownable`. Once the owner uses this function to renounce ownership, the contract becomes ownerless. Evidence in the transaction logs shows that, following the activation of the `renounceOwnership` function, any attempts to invoke functions requiring owner permissions fail, with the error message: `"Ownable: caller is not the owner."` This condition makes the contract's adjustable parameters immutable, potentially rendering the contract ineffective for any future administrative modifications that might be needed.

Assets:

- FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract]

Status:

Accepted

Recommendations

Recommendation:

To mitigate this vulnerability:

- Override the `renounceOwnership` function to revert transactions: By overriding this function to simply revert any transaction, it will become impossible for the contract owner to unintentionally (or intentionally) render the contract ownerless and thus immutable.

F-2024-0360 - Public Functions That Should Be External - Info

Description: Functions that are meant to be exclusively invoked from external sources should be designated as `external` rather than `public`. This is essential to enhance both the gas efficiency and the overall security of the contract.

```
function destroyBlackFunds(
        address _from
    ) public whenNotPaused onlyBlacklister {
```

Public function `destroyBlackFunds` is never called in the contract.

Assets:

- FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract]

Status: Accepted

Recommendations

Recommendation: Transition the relevant functions, which are exclusively utilized by external entities, from their current `public` visibility setting to the `external` visibility setting.

F-2024-0362 - Use Ownable2Step library instead of Ownable - Info

Description:	Ownable2Step and Ownable2StepUpgradeable prevent the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner permissions actively accept via a contract call of its own.
Assets:	• FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract]
Status:	Accepted

Recommendations

Recommendation:	Consider using `Ownable2Step` or `Ownable2StepUpgradeable` instead of `Ownable` or `OwnableUpgradeable` from OpenZeppelin Contracts to enhance the security of contract ownership management. These contracts prevent the accidental transfer of ownership to an address that cannot handle it, such as due to a typo, by requiring the recipient of owner permissions to actively accept ownership via a contract call. This two-step ownership transfer process adds an additional layer of security to contract's ownership management.

F-2024-0363 - Style Guide Violation - Info

Description:

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:

1. Type declarations
2. State variables
3. Events
4. Modifiers
5. Functions

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

1. Constructor
2. Receive function (if exists)
3. Fallback function (if exists)
4. External functions
5. Public functions
6. Internal functions
7. 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.

On `FiatTokenV1` modifiers are not defined before than functions. Function orderings should follow the given order. Currently, there are public functions before external functions, modifiers after functions, etc.

Assets:

- FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract]

Status: Accepted

Recommendations

Recommendation:	Consistent adherence to the official Solidity style guide is recommended. This enhances 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.

F-2024-0375 - Solidity version 0.8.20 might not work on all chains due to `PUSH0` - Info

Description:	Solidity version 0.8.20 employs the recently introduced **PUSH0** opcode in the Shanghai EVM, this opcode might not be universally supported across all blockchain networks and Layer 2 solutions. It is advisable to use an earlier version of Solidity to ensure compatibility.
Assets:	• FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract] • FiatTokenV2.sol [https://github.com/commoncentz/commoncentz-contract]
Status:	Accepted

Recommendations

Recommendation:	To ensure compatibility with a wide range of blockchain networks and Layer 2 solutions, consider using an earlier version of Solidity that does not rely on the **PUSH0** opcode introduced in Solidity version 0.8.20. Using a more widely supported Solidity version can help avoid potential compatibility issues and ensure the smooth deployment and execution of smart contracts.

F-2024-0377 - Unpacked Variables Consuming Gas - Info

Description: In Ethereum and similar blockchain platforms, the cost of storage is one of the most significant factors affecting transaction costs. Each storage variable occupies a separate slot, and slots are charged individually. This can result in increased gas costs when contracts use a suboptimal arrangement of storage variables.

The issue at hand is related to the order in which storage variables are declared within a contract. If a variable we are trying to pack exceeds the 32-byte limit of the current slot, it gets stored in a new one.

However, each storage variable in Solidity, regardless of its size, consumes one 32-byte storage slot, except for structs and arrays. By packing variables together, it is possible to optimize the storage layout and reduce the number of slots required, thereby minimizing storage costs.

Assets:

- FiatTokenV1.sol [https://github.com/commoncentz/commoncentz-contract]

Status: Accepted

Recommendations

Recommendation: To optimize storage and reduce Gas costs, rearrange the storage variables in a way that makes the most of each 32-byte storage slot. For example, the order of the variables in the contract:

```
string public name; (32 bytes)
string public symbol; (32bytes)
uint8 public decimals;(1 bytes)
string public currency; (32 bytes)
address public blacklister; (20 bytes)
address public masterMinter; (20 bytes)
uint256 internal totalSupply_; (32 bytes)
```

Storing it like in order as follows will save 1 storage :

```
string public name; (32 bytes)
string public symbol; (32bytes)
string public currency; (32 bytes)
uint8 public decimals;(1 bytes)
address public blacklister; (20 bytes)
address public masterMinter; (20 bytes)
uint256 internal totalSupply_; (32 bytes)
```

For detailed info: [Official Solidity Docs: Layout in Storage]()

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 **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
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 will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

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

Scope Details

Repository	https://github.com/commoncentz/commoncentz-contract
Commit	13d8809272e8f2e71552eb74b007716a5389cb77
Whitepaper	Not Provided
Requirements	Not Provided
Technical Requirements	Not Provided

Contracts in Scope

./contracts/v1/FiatTokenV1.sol

./contracts/v2/FiatTokenV2.sol