HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: SG Forge
Date: 16 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 SG Forge
Approved By	Viktor Lavrenenko | SC Auditor at Hacken OÜ David Camps Novi | SC Audits Lead at Hacken OÜ Grzegorz Trawiński | SC Audits Approver at Hacken OÜ
Tags	ERC20 token; StableCoin;
Platform	EVM
Language	Solidity
Methodology	Link
Website	SG FORGE | Bridging the gap between Capital Markets and Digital Assets
Changelog	03.10.2023 - Initial Review 16.10.2023 - Second Review

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

Table of contents

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

Introduction

Hacken OÜ (Consultant) was contracted by SG Forge (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

SMART_COIN is an ERC20 stablecoin pegged to the Euro, with the following contracts:

- *SmartCoin* — ERC20 extended with:
  - UUPS upgrade mechanism
  - Operator Roles: *registrar*, *operations*, *technical*. These roles are introduced to manage the upgrade to new implementations and control transfers of tokens amongst these roles.
- *WhitelistUpgradeable* — the contract, which:
  - Manages the 3 roles: sets and holds their addresses, includes roles' modifiers.
  - Manages the whitelist: adds/removes addresses, and includes whitelist modifiers.
  - Manages new implementations: authorizes implementation, names new implementation operators, and allows operators to accept the new roles.
- *WhitelistDataLayout* - the contract contains the storage part of the WhitelistUpgradeable contract, and stores the data of the new implementation contract: its address and the addresses of new roles. Furthermore, it stores the whitelisted users.
- *SmartCoinDataLayout* - the contract, which contains the storage part of the SmartCoin contract.
- *EncodingUtils* - a library that has the functionality to compute the hash of transfer requests

Privileged roles

- Registrar operator:
  - Manages Whitelist of authorized users.
  - Validates/Rejects transfers to *registrar* and *operations* operators.
  - Names the operators for the new implementation.
  - Authorizes the upgrade to the next implementation.
  - Cannot be used as spender or destination of *transferFrom()*.
  - Can retrieve tokens from any address to itself.
  - Can mint and burn SmartCoin tokens.
- Operations operator:

    - Used when token owners want to sell the *SmartCoin* tokens to the issuer in exchange for cash
    - Cannot be used as spender or destination of *transferFrom()*.
    - Transfers to *operations* must be validated by the *registrar*.
- Technical operator:
    - Launches a previously authorized (by *registrar*) *implementation* upgrade.

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.
- Technical description is provided:
    - Technical specification is provided.
    - NatSpec is sufficient.

Code quality

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

- The development environment is configured.
- Deployment instructions are provided.

Test coverage

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

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

Security score

As a result of the audit, the code contains **1** high issue. The security score is **5** out of **10**.

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

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

Summary

According to the assessment, the Customer's smart contract has the following score: **6.5**. 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
3 Oct 2023	2	0	1	0
13 Oct 2023	0	0	1	0

Risks

- The SmartCoin contract is upgradeable, which means that the contract logic can be changed by the protocol owners at any time. The users should be aware of this fact when interacting with Upgradeable Contracts.
- The protocol uses a whitelisting method in order to control the users who can transfer the SmartCoin tokens. This means that users' tokens can become locked in their wallets if they are removed from the whitelist while holding tokens.
- The stablecoin supply is managed as explained in section 4 of the WhitePaper, although being a centralized protocol the users must trust the owners to keep manage peg EUR-EURCV and manage the collateral correctly.

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

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.	Not Relevant	
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.	Passed	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

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

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.	Failed	H01
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Failed	H01
Data Consistency	Smart contract data should be consistent all over the data flow.	Not Relevant	

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.	Passed	
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. 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

No critical severity issues were found.

■■■ High

H01. Highly Permissive Role Allows Access To Users Funds

Impact	High
Likelihood	Medium

The *Registrar* role possesses extensive control over users' funds. By using the *recall()* function, the *registrar* can retrieve tokens from SmartCoin holders.

```
/**
 * @dev Recalls a `amount` amount of tokens from `from` address
 * The tokens are transferred back to the registrar operator
 *
 * NB: This method is reserved to the registrar operator.
 */
function recall(address _from, uint256 _amount)
    external
    override
    onlyRegistrar
    onlyWhenBalanceAvailable(_from, _amount)
    returns (bool)
{
    super._transfer(_from, registrar, _amount);
    return true;
}
```

Roles should never be able to access users' funds without their permission. Additionally, overextended permissions can lead to substantial security risks, especially in the event of a key leak. Such a leak could precipitate dire consequences, potentially resulting in security breaches that undermine the integrity and trustworthiness of the entire system.

Path: ./contracts/smartCoin/SmartCoin.sol: *recall()*

Recommendation: The team should reconsider the protocol's design. It is recommended to delete the *recall()* functionality or add the extra allowance mechanism, which would allow users to give the necessary permission to the *registrar*. The *registrar* role should be managed via a multi-signature wallet.

Found in: f34ba59

Status: Acknowledged (Revised commit: dab4c9e)

Resolution: The SG Forge team decided not to follow the recommendations given the following explanation:

"The recall() function is a mandatory feature requested by the Compliance team and the internal policy of Societe Generale. Nonetheless, this recall feature is documented in the Terms & Conditions attached and provided to our investors. This feature could legally be triggered for special events as mentioned in paragraph 8.2 of the T&C ("Conversion following a Special Event")."

Additional information can be found in the project Terms and Conditions document.

Medium

No medium severity issues were found.

Low

L01. Missing Zero Address Validation

Impact	Low
Likelihood	Low

Additional checks against the *0x0* address should be included in the reported functions to avoid unexpected results.

Path: ./contracts/smartCoin/SmartCoin.sol: *constructor()*

Recommendation: It is recommended to add zero address checks.

Found in: f34ba59

Status: Fixed (Revised commit: dab4c9e)

Resolution: *onlyNotZeroAddress()* modifier has been created to protect the *constructor()* from *0x0* addresses.

L02. Multiple Roles Can Be Set For A Single Address

Impact	Low
Likelihood	Low

Missing address equality check in the *nameNewOperators()* function allows the *registrar* role to pass the same address for all system roles, which will create a negative impact on the system security and make it vulnerable to attacks.

Path:

./contracts/smartCoin/SmartCoin.sol: *nameNewOperators()*

Recommendation: It is recommended to enhance the system's security by implementing the check to ensure that the passed addresses for *operations* and *technical* roles are different from each other and from the *registrar*.

Found in: f34ba59

Status: Fixed (Revised commit: dab4c9e)

Resolution: *OnlyWhenOperatorsHaveDifferentAddress* modifier has been added to allow only different addresses for the *registrar*, *operations* and *technical* roles.

Informational

I01. Style Guide Violation: Order Of Layout

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.

Paths:
./contracts/smartCoin/SmartCoin.sol
./contracts/smartCoin/WhitelistUpgradeable.sol

Recommendation: It is recommended to change the order of layout to fit the Official Style Guide.

References: Solidity Style Guide

Found in: f34ba59

Status: Fixed (Revised commit: dab4c9e)

Resolution : The layout order has been updated to comply with the Solidity Style Guide.

I02. Variable Shadowing as Improper Coding Standard

The variable names *registrar*, *operations,* and *technical* in the *SmartCoin* *constructor* overshadow variables of the same name within the contract scope when inheriting from *WhitelistUpgradeable*.

- SmartCoin

```
constructor(address registrar, address operations, address technical)
WhitelistUpgradeable(registrar, operations, technical) {
        _disableInitializers();
    }
```

- WhitelistUpgradeable

```
    address public immutable registrar;
    address public immutable operations;
    address public immutable technical;
    constructor(address _registrar, address _operations, address _technical) {
        registrar = _registrar;
        operations = _operations;
        technical = _technical;
    }
```

Such a situation can lead to confusion and unintended consequences during code execution.

Path: ./contracts/smartCoin/SmartCoin.sol: *constructor()*.

Recommendation: It is recommended to rename the constructor arguments *registrar*, *operations,* and *technical* as *_registrar*, *_operations,* and *_technical*.

References: SWC-119

Found in: f34ba59

Status: Fixed (Revised commit: dab4c9e)

Resolution: The constructor arguments were updated to *_registrar*, *_operations,* and *_technical*.

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

I03. Missing SPDX License Identifier

Trust in smart contracts can be better established if their source code is available. Since making source code available always touches on legal problems with regard to copyright, the Solidity compiler encourages the use of machine-readable *SPDX license identifiers*. Every source file should start with a comment indicating its *license*.

Path: ./contracts/*.sol

Recommendation: Implement *SPDX License Identifiers* at the beginning of the aforementioned files.

Found in: f34ba59

Status: Fixed (Revised commit: dab4c9e)

Resolution: The missing SPDX License Identifier has been added to the aforementioned files.

I04. Missing Initialization Calls To Parent Contracts

The function *initialize()* does not call the initialize functions of its parent contracts *__Whitelist_init()* and *__UUPSUpgradeable_init()*.

Solidity takes care of automatically invoking the constructors of all ancestors of a contract. However, when working with Upgradeable contracts it is necessary to manually call the initializers of all parent contracts.

In this case *__Whitelist_init()* and *__UUPSUpgradeable_init()* initializers contain empty implementation, thus, it has no impact on processing flow.

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

Recommendation: Call *__Whitelist_init()* and *__UUPSUpgradeable_init()* in the *initialize()* function of SmartCoin contract.

Found in: f34ba59

Status: Fixed (Revised commit: dab4c9e)

Resolution: The necessary function calls have been added to *initialize()*.

I05. State Variables In Upgradeable Contracts Should Be In Initializer

The contract *WhitelistUpgradeable* sets the state variables *registrar*, *operations* and *technical* in the *constructor*. Since those variables are *immutable*, they will be kept in the implementation bytecode.

However, when working with upgradeable contracts, it is recommended to work with *initializers* instead of *constructors*, because the

variables will then be stored in the proxy storage with the rest of the variables. This will maintain consistency amongst the whole storage, making it more robust to corner cases, unexpected interactions or invalid upgrades.

```
address public immutable registrar;
address public immutable operations;
address public immutable technical;

constructor(address _registrar, address _operations, address _technical) {
    registrar = _registrar;
    operations = _operations;
    technical = _technical;
}
```

Path: ./contracts/smartCoin/WhitelistUpgradeable.sol: constructor().

Recommendation: It is recommended to initialize the variables *registrar*, *operations* and *technical* via *initializer* instead of using a *constructor*.

References: OpenZeppelin Upgradeable Contracts FAQs.

Found in: f34ba59

Status: Mitigated (Revised commit: dab4c9e).

Resolution: The recommendation was not followed provided the given explanation from the client in compliance with OpenZeppelin plugin:

"The main goal for storing the operators in contract's bytecode is to lower the gas cost, as we perform checks on Operations and Registrar addresses in every transfer, approve, transferFrom, increaseAllowance, and decreaseAllowance.

For invalid upgrades, we would not be able to compromise the storage layout as it is verified by the Openzeppelin plugin @openzeppelin/hardhat-upgrades, and operators are checked by the contract itself."

I06. EngagedAmount Collection Processing Lacks Consistency

The mapping *_engagedAmount*, which is responsible for storing the engaged amounts of transactions executed to the *operations* or *registrar* roles, uses the *unchecked{}* keyword for Gas Optimization in *validateTransfer()* and *_rejectAmount()*.

```
function validateTransfer(bytes32 transferHash) external onlyRegistrar
returns (bool) {
        TransferRequest memory _transferRequest = _transfers[transferHash];
```

```
        if (_transferRequest.status == TransferStatus.Undefined) {
            revert TransferRequestNotFound();
        }

        if (_transferRequest.status != TransferStatus.Created) {
            revert InvalidTransferRequestStatus();
        }

        _transfers[transferHash].status = TransferStatus.Validated;

        unchecked {
            _engagedAmount[_transferRequest.from] -=
_transferRequest.value;
        }

        _safeTransfer(
            _transferRequest.from,
            _transferRequest.to,
            _transferRequest.value
        );

        emit TransferValidated(transferHash);
        return true;
    }
```

```
function _initiateTransferRequest(
        address _from,
        address _to,
        uint256 _value
    ) internal {
        unchecked {
            _engagedAmount[_from] += _value;
        }

        bytes32 transferHash = EncodingUtils.encodeRequest(
            _from,
            _to,
            _value,
            _requestCounter
        );

        _transfers[transferHash] = TransferRequest(
            _from,
            _to,
            _value,
```

```
            TransferStatus.Created
        );

        _requestCounter += 1;

        emit Transfer(_from, _to, 0);
        emit TransferRequested(transferHash, _from, _to, _value);
    }
```

However, the mapping is not wrapped in *unchecked{}* in *rejectTransfer()*. As a result, it creates a code inconsistency:

```
function rejectTransfer(bytes32 transferHash) external onlyRegistrar
returns (bool) {
        TransferRequest memory transferRequest = _transfers[transferHash];

        if (transferRequest.status == TransferStatus.Undefined) {
            revert TransferRequestNotFound();
        }

        if (transferRequest.status != TransferStatus.Created) {
            revert InvalidTransferRequestStatus();
        }

        _engagedAmount[transferRequest.from] -= transferRequest.value;

        _transfers[transferHash].status = TransferStatus.Rejected;

        emit TransferRejected(transferHash);

        return true;
    }
```

Path: ./contracts/SmartCoin.sol: rejectTransfer().

Recommendation: Consider using the keyword *unchecked{}* in *rejectTransfer()* for Gas Optimization and consistency along the code.

Found in: f34ba59

Status: Fixed (Revised commit: dab4c9e)

Resolution: unchecked{} was added into rejectTransfer() for consistency.

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

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.

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

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, do not affect security score but can affect code quality score.

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

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.

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

Appendix 2. Scope

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

Initial review scope

Repository	https://drive.google.com/file/d/1tSRyv3FxujjpSZl6DkhmS9vIKbmuZbrT/view?usp=sharing
Commit	f34ba59
Whitepaper	Not provided
Requirements	Link
Technical Requirements	README.md
Contracts	File: EncodingUtils.sol SHA3: 1da492afaf87f8fa077a0058c1317f01bf05f2f2e88a646f97fa3535c201063e File: ISmartCoin.sol SHA3: fd99d650908914900e3e57b7d96873c5b6cc9e346706bc5b133a6d1e0b95529f File: IWhitelist.sol SHA3: 5e71a2aa9aeed3ebb2c7d5235f862c59b09b7dcd40532e87b3a5f0c1fc27f0c2 File: SmartCoin.sol SHA3: 08ccc0f51edbefbdae0e9d36eb64496c46669d5e74d22ca2b17eeea7d5d28989 File: WhitelistDataLayout.sol SHA3: 57e4095847575ec6b84d45776669109838f11c0757553503b20cbb4f0eb349cc File: SmartCoinDataLayout.sol SHA3: 37ce8cbea18fd3ee62aa8b93911236b37b96f685ae83de8275e35e5dc57ef53d File: WhitelistUpgradeable.sol SHA3: ec62ecffd2214a11263efd0c7e525a7ef01c62f40a5b2e9a7dfd6ce4ece91ea1

Second review scope

Repository	Zip file
Commit	dab4c9e
Whitepaper	Whitepaper
Requirements	Link
Technical Requirements	README.md
Contracts	File: contracts/libraries/EncodingUtils.sol SHA3: 730be9ee27a42f1803b43ae013e38483c0f5ce2caf7b998caadeacf5fc484066 File: contracts/smartCoin/ISmartCoin.sol

	SHA3: 94d769fde091fd6b05ac87762739627c68e8a4f50817ab2376170afe11081a3b File: contracts/smartCoin/IWhitelist.sol SHA3: 5e93023a686c333e5f64385f8f598a23642e21526080f4a3e05373e39b03563f File: contracts/smartCoin/SmartCoin.sol SHA3: 88c18e32728d4c903354c22b343906e17aeec45bc2743f5d005af65746c2cc16 File: contracts/smartCoin/SmartCoinDataLayout.sol SHA3: 0dfc3a8ef4b30dad175fb2413ed7d410cd569a45c4ac181e001c7831ec4beabc File: contracts/smartCoin/WhitelistDataLayout.sol SHA3: 7541c176fa0443cb0dcd33ce4c59710d885864d376194db8bf88c0635feeff4e File: contracts/smartCoin/WhitelistUpgradeable.sol SHA3: d3f5ac7fb487681ebc3ed03372a8d54f0294ae32c71632c80c9792dfcb31fb58