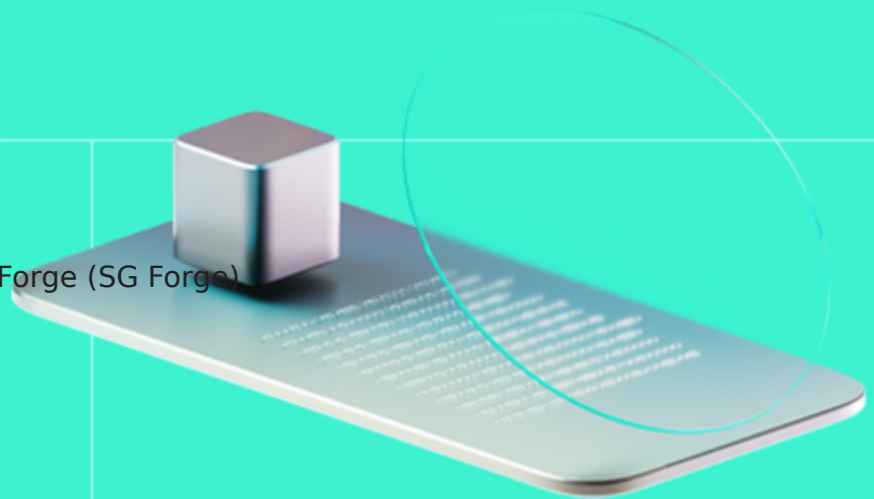




Smart Contract Code Review And Security Analysis Report

Customer: Societe Generale Forge (SG Forge)

Date: 30/10/2025



We express our gratitude to the Societe Generale Forge (SG Forge) team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

SmartCoin is a regulated, upgradeable stablecoin framework by SG-Forge, designed to tokenize euro-backed assets with strict compliance controls and off-chain redemption logic—merging institutional oversight with on-chain transparency.

Document

Name	Smart Contract Code Review and Security Analysis Report for Societe Generale Forge (SG Forge)
Audited By	Kornel Światłowski, Kaan Caglan
Approved By	Ataberk Yavuzer
Website	https://www.sgforge.com/
Changelog	22/10/2025 - Preliminary Report 30/10/2025 - Final Report
Platform	EVM
Language	Solidity
Tags	ERC20, Stablecoin
Methodology	https://hackenio.cc/sc_methodology

Review

Scope

File	smartcoin-ethereum-smartcoin_v4_20251007.zip
SHA3-256	7e42e1875bf791ca742912b8773147b8d8729a834d3d880c8cbee083a3e908d6
Commit Id	2b796273dce622976dd470032150e7a21a4e256e
Remediation	smartcoin-ethereum-smartcoin_v4_20251027_hacken_fix_1.zip
Remediation SHA3-256	428dfbfafc44f1f8f74809e1183a5b4852b0de217e49ec0083dc4bfe01e9de21
Remediation Commit Id	e8bc554bd28ffa4bef5765307655bcd094364115

Audit Summary

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

7	2	4	1
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	1
Medium	1
Low	2

Vulnerability	Severity	Status
F-2025-13637 - Bypass of Redemption Review via approve + transferFrom	High	Fixed
F-2025-13621 - Single Role Control Enables Arbitrary Token Burning From Frozen Addresses	Medium	Accepted
F-2025-13664 - Missing Modifier Enforcement in Approve Function Enables Unauthorized Approvals	Low	Fixed
F-2025-13671 - `wipeFrozenAddress()` Not Restricted by Pause State	Low	Mitigated
F-2025-13634 - Cache Variable Array Length In For Loop	Info	Accepted
F-2025-13636 - Repeated state reads in role-accept functions	Info	Accepted
F-2025-13638 - Unneeded initializations of uint256 and bool variable to 0/false	Info	Accepted

Documentation quality

- Functional requirements are present, but only at a high-level.
 - Roles description is provided.
 - Basic system description is provided.
- Technical description is robust.
 - Run instructions are provided.
 - Technical specification is provided.
 - NatSpec is sufficient.

Code quality

- The development environment is configured.
- Up-to-date Solidity version.
- NatSpec covers the code and is beneficial.

Test coverage

Code coverage of the project is **99.36%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is present.

Table of Contents

System Overview	6
Privileged Roles	6
Potential Risks	8
Findings	9
Vulnerability Details	9
Disclaimers	27
Appendix 1. Definitions	28
Severities	28
Potential Risks	28
Appendix 2. Scope	29
Appendix 3. Additional Valuables	30

System Overview

`SmartCoin` is an upgradeable ERC20 stablecoin using the UUPS proxy pattern. It implements mint/burn functionality controlled by the registrar operator and supports address freezing/blacklisting for compliance purposes. The contract includes a redemption system where transfers to designated redemption addresses are automatically routed through the registrar for validation. It also provides an emergency wipe function for frozen addresses and can be paused for circuit-breaker scenarios.

`AccessControlUpgradeable` implements a three-operator role system consisting of the Registrar (who controls minting, burning, freezing, pausing, and upgrade authorization), the Operations operator (a special address for token redemptions), and the Technical operator (who executes authorized upgrades). The contract manages the upgrade workflow with multi-party approval requirements and handles the redemption address registry.

`SmartCoinDataLayout` defines the storage structure for the `SmartCoin` contract. It contains transfer request structs and mappings for future features and includes a storage gap to enable upgrade-safe field additions without breaking the storage layout.

`AccessControlDataLayout` defines the storage structure for the `AccessControlUpgradeable` contract. It stores frozen addresses, pause state, operator roles, and redemption addresses, with a storage gap included for upgrade-safe field additions.

Privileged roles

The `SmartCoin` system implements a three-operator governance model with strict separation of duties:

- The `Registrar` is the primary administrative role with the following capabilities:
 - Token Supply Management
 - Mint new tokens to unfrozen addresses
 - Burn tokens from their own balance
 - Wipe (burn) all tokens from frozen addresses
 - Access Control & Compliance
 - Freeze addresses (blacklist users)
 - Unfreeze addresses (remove from blacklist)
 - Add addresses to the redemption address registry
 - Remove addresses from the redemption address registry
 - Contract State Management
 - Pause the contract (emergency circuit breaker)
 - Unpause the contract to resume normal operations
 - Upgrade Governance
 - Name new operators for the next implementation contract upgrade
 - Authorize specific implementation contracts for upgrade
 - Review and validate/reject transfers to the registrar or operations address
- The `Technical` role is responsible for executing contract upgrades:
 - Execute previously authorized upgrades using `upgradeTo`
 - Execute previously authorized upgrades with initialization data using `upgradeToAndCall`

- The **Operations** role does not have any privileges

Potential Risks

- **Flexibility and Risk in Contract Upgrades:** The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.
- **Absence of Upgrade Window Constraints:** The contract suite allows for immediate upgrades without a mandatory review or waiting period, increasing the risk of rapid deployment of malicious or flawed code, potentially compromising the system's integrity and user assets.
- **Single Entity Upgrade Authority:** The token ecosystem grants a single entity the authority to implement upgrades or changes. This centralization of power risks unilateral decisions that may not align with the community or stakeholders' interests, undermining trust and security.
- **Centralized Control of Minting Process:** The token contract's design allows for centralized control over the minting process, posing a risk of unauthorized token issuance, potentially diluting the token value and undermining trust in the project's economic governance.
- **Incomplete Enforcement of Transfer Restrictions:** The contract's logic for special addresses (registrar, operations, redemption) depends heavily on specific modifiers and manual configuration. If these controls are misapplied or misunderstood, users could bypass intended compliance checks or trigger unreviewed token movements, undermining regulatory integrity.
- **Reliance on Off-Chain Processes:** Critical redemption and fiat-settlement steps occur entirely off-chain. Any breakdown in the off-chain infrastructure, such as delays, compliance misconfigurations, or manual processing errors, could desynchronize token supply from real-world collateral, creating reconciliation and trust issues.
- **Potential Front-Running and Transfer Redirect:** The overridden `transfer()` function redirects transfers to the registrar address when the original recipient is designated as a redemption address. The registrar can dynamically add any address to the redemption mapping without delay. This design introduces a front-running risk, where the registrar could add a target address immediately before a transaction is executed, redirecting tokens to the registrar instead of the intended recipient. While this behavior is part of the intended functionality, it also amplifies the impact of potential registrar private key compromise or unauthorized access, which could result in the redirection of all token transfers and loss of user funds.

Findings

Vulnerability Details

[F-2025-13637](#) - Bypass of Redemption Review via approve + transferFrom - High

Description:

The SmartCoin contract intends to enforce manual review of redemptions (cash-outs) by routing all transfers to the `operations` address through the `registrar` and emitting a `RedemptionStarted` event. However, this restriction only applies to the `transfer()` function.

Because `approve()` and `transferFrom()` lack equivalent checks, any user can indirectly transfer tokens to the `operations` address without triggering the review mechanism.

```
/**
 * @dev Sets a `value` amount of tokens as the allowance of `spender` over the
 * caller's tokens.
 * NB: both the owner(msg.sender) and the spender have to be not frozen
 * NB: Will fail if the spender is either the registrar operator or the operations operator
 */
function approve(
    address _spender,
    uint256 _value
)
    public
    override(ERC20Upgradeable, ISmartCoin)
    onlyWhenNotPaused
    onlyNotFrozen(_msgSender())
    onlyNotFrozen(_spender)
    returns (bool)
{
    super._approve(_msgSender(), _spender, _value);
    return true;
}

/**
 * @dev Same semantic as ERC20's transferFrom function(no validation needed)
 * NB: the owner(_from), the spender(msg.sender) and the destination have to be unfrozen
 * NB: Will fail if the destination is either the registrar operator or the operations operator
 */
function transferFrom(
```

```

        address _from,
        address _to,
        uint256 _value
    )

    public
    override(ERC20Upgradeable, ISmartCoin)
    onlyWhenNotPaused
    onlyNotFrozen(_msgSender())
    onlyNotFrozen(_from)
    onlyNotFrozen(_to)
    returns (bool)
    {
        return super.transferFrom(_from, _to, _value);
    }

```

Although the documentation explicitly states that approvals or transfers involving the registrar or operations addresses should fail, these restrictions are not enforced in the actual implementation.

Assets:

- contracts/smartCoin/SmartCoin.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]

Status:

Fixed

Classification

Impact Rate:	4/5
Likelihood Rate:	4/5
Exploitability:	Independent
Complexity:	Simple
Severity:	High

Recommendations

Remediation: To ensure that all redemptions follow the same compliance and manual review process, the logic in `approve()` and `transferFrom()` should be aligned with `transfer()`.

Add `forbiddenForRegistrar(_spender)` and `forbiddenForOperations(_spender)` in `approve()` to prevent approvals involving privileged entities.

Add `forbiddenForRegistrar(_to)` and `forbiddenForOperations(_to)` in `transferFrom()` to prevent direct token transfers to these addresses.

Resolution:

The Finding was fixed in the file hash

`428dfbfafc44f1f8f74809e1183a5b4852b0de217e49ec0083dc4bfe01e9de21` by changing the

`transferFrom` functionality and redirect to redemption address.

```
/**
 * @dev Same semantic as ERC20's transferFrom function(no validation needed)
 * NB: the owner(_from),the spender(msg.sender) and the destination have to be
unfrozen
 */
function transferFrom(
    address _from,
    address _to,
    uint256 _value
)
    public
    override(ERC20Upgradeable, ISmartCoin)
    onlyWhenNotPaused
    onlyNotFrozen(_msgSender())
    onlyNotFrozen(_from)
    onlyNotFrozen(_to)
    returns (bool)
{
    if (isRedemptionAddress[_to]) {
        super.transferFrom(_from, registrar, _value);
        emit RedemptionStarted(_from, _to, _value);
    } else {
        return super.transferFrom(_from, _to, _value);
    }

    return true;
}
```

Evidences

POC

Reproduce:

POC Steps:

- 1- Alice holds 10 tokens and wants to redeem them.
 - 2- Alice calls `approve(Bob, 10)` to grant Bob permission to transfer on her behalf.
 - 3- Bob calls `transferFrom(Alice, operations, 10)`.
 - 4- Tokens move directly from Alice → operations.
- The `registrar` never receives custody.

```
it.only('PoC: approve(operations) + transferFrom(..., operations, ...) bypass
es redemption review', async function () {
```

```

const tx = smartCoin
    .connect(signers.registrar)
    .addRedemptionAddresses([operationsAddress]);

await expect(
    smartCoin.connect(signers.investor1).approve(investor2Address, amount)
).to.emit(smartCoin, 'Approval').withArgs(investor1Address, investor2Address
, amount);

const registrarBefore = await smartCoin.balanceOf(registrarAddress);
const opsBefore = await smartCoin.balanceOf(operationsAddress);
const userBefore = await smartCoin.balanceOf(investor1Address);
console.log("Balances before transferFrom - Registrar:", registrarBefore.toString(), "Operations:", opsBefore.toString(), "User:", userBefore.toString());

// OPERATIONS pulls funds directly from investor1 to OPERATIONS via transfer
From
await smartCoin
    .connect(signers.investor2)
    .transferFrom(investor1Address, operationsAddress, amount);

// Vulnerable behavior:
// 1) Tokens landed directly in OPERATIONS (no registrar custody)
// 2) Registrar balance did NOT increase (so no "review" custody step took place)

const registrarAfter = await smartCoin.balanceOf(registrarAddress);
const opsAfter = await smartCoin.balanceOf(operationsAddress);
const userAfter = await smartCoin.balanceOf(investor1Address);

expect(registrarAfter).to.equal(registrarBefore, 'Registrar balance should not change in bypass');
expect(opsAfter).to.equal(opsBefore.add(amount), 'Operations should receive tokens directly');
expect(userAfter).to.equal(userBefore.sub(amount), 'User balance should decrease by amount');

const amount2 = 1;

await smartCoin.connect(signers.investor1).transfer(operationsAddress, amount2);

expect(await smartCoin.balanceOf(registrarAddress)).to.equal(registrarAfter.add(amount2));
expect(await smartCoin.balanceOf(operationsAddress)).to.equal(opsAfter); // unchanged by direct transfer
});

```

Results:

```
~12:41:21 kaancaglan@Mac $ npx hardhat test
No PRIVATE_KEY given, only the hardhat test network will be available

SmartCoin: approve
SmartCoin: Approve
Balances before transferFrom - Registrar: 0 Operations: 0 User: 10
✓ PoC: approve(operations) + transferFrom(..., operations, ...) bypasses red
emption review

1 passing (329ms)
```

[F-2025-13621](#) - Single Role Control Enables Arbitrary Token Burning From Frozen Addresses - Medium

Description:

The `SmartCoin` contract is implemented as an upgradeable ERC20 token. It defines the `wipeFrozenAddress()` function, which permanently removes (burns) the entire token balance of a specified address. This function the `_burn()` from `ERC20Upgradeable`, allowing token burning from any arbitrary address. Access to this function is restricted by the `onlyRegistrar` modifier, limiting its execution to the registrar operator role. Additionally, a modifier named `onlyFrozen` ensures that the provided address is marked as frozen before the burn operation proceeds.

```
/**
 * @dev Burns the balance of `_addr` address
 *
 * NB: This method is reserved to the registrar operator.
 */
function wipeFrozenAddress(address _addr) external override onlyRegistrar
    onlyFrozen(_addr) returns(bool) {
    super._burn(_addr, balanceOf(_addr));
    return true;
}
```

The `setFrozen()` function, which updates the `isFrozen` flag for an address, is also restricted by the same `onlyRegistrar` modifier. As a result, both freezing an address and performing the burn operation can be executed by the same privileged role. This design consolidates excessive power within a single role, creating a risk of unauthorized or accidental token destruction.

```
/**
 * @dev Freeze a list of addresses
 * NB : only the registrar operator can call this method
 */
function freeze(address[] calldata _addrs) external onlyRegistrar
    returns(bool) {
    for (uint256 i = 0; i < _addrs.length; i++) {
        if (isFrozen[_addrs[i]]) revert AddressAlreadyFrozen(_addrs[i]);
        isFrozen[_addrs[i]] = true;
    }
    emit AddressesFrozen(_addrs);
    return true;
}
```

Such unrestricted control introduces centralization risk and undermines the integrity of token management. The registrar operator could, either

maliciously or through error, freeze and burn tokens from any holder address without oversight, resulting in permanent and unrecoverable asset loss.

Assets:

- contracts/smartCoin/SmartCoin.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]

Status:

Accepted

Classification

Impact Rate: 5/5

Likelihood Rate: 3/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Medium

Recommendations**Remediation:**

It is recommended to remove the `wipeFrozenAddress()` function from the codebase. Alternatively, to mitigate the issue, separate role-based access control should be implemented to ensure that freezing and burning operations are handled by distinct privileged roles. Additionally, comprehensive documentation should be provided to describe the intended use cases and operational conditions for the function.

Resolution:

This Finding is accepted by the SG Forge team.

[F-2025-13664](#) - Missing Modifier Enforcement in Approve Function Enables Unauthorized Approvals - Low

Description:

According to the NatSpec description of the overridden `approve()` function in the `SmartCoin` contract, the function is expected to revert if the provided `_spender` argument corresponds to an operator or operations operator address. This restriction is intended to prevent these privileged roles from being granted additional spending permissions through standard ERC20 allowance mechanisms.

```
/**
 * @dev Sets a `value` amount of tokens as the allowance of `spender` over the
 * caller's tokens.
 * NB: both the owner(msg.sender) and the spender have to be not frozen
 * NB: Will fail if the spender is either the registrar operator or the operations operator
 */
function approve(
    address _spender,
    uint256 _value
)
    public
    override(ERC20Upgradeable, ISmartCoin)
    onlyWhenNotPaused
    onlyNotFrozen(_msgSender())
    onlyNotFrozen(_spender)
    returns (bool)
{
    super._approve(_msgSender(), _spender, _value);
    return true;
}
```

However, unlike the `increaseAllowance()` function—which correctly enforces this restriction through the `forbiddenForRegistrar()` and `forbiddenForOperations()` modifiers—the `approve()` function lacks these modifiers. This omission contradicts the documented behavior and creates functional inconsistency between the two allowance-modifying functions.

```
/**
 * @dev Atomically increases the allowance granted to `spender` by the caller.
 *
 * This is an alternative to {approve} that can be used as a mitigation for
 * problems described in {IERC20-approve}.
 *
 * Emits an {Approval} event indicating the updated allowance.
 */
```

```

*
* Requirements:
*
* - `spender` cannot be the zero address.
* - `spender` and msg.sender have to be unfrozen
* - `spender` cannot be either the registrar operator or the operations operator
or
*/
function increaseAllowance(
    address _spender,
    uint256 _addedValue
)
public
override(ERC20Upgradeable, ISmartCoin)
onlyWhenNotPaused
onlyNotFrozen(_spender)
onlyNotFrozen(_msgSender())
forbiddenForRegistrar(_spender)
forbiddenForOperations(_spender)
returns (bool)
{
    return super.increaseAllowance(_spender, _addedValue);
}

```

As a result, operator and operations operator addresses can be approved as spenders through the `approve()` function, potentially allowing unintended interactions or privilege escalations that bypass the intended security model. This inconsistency may also cause confusion for external integrators or audits relying on NatSpec documentation for behavioral expectations.

Assets:

- contracts/smartCoin/SmartCoin.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]

Status:

Fixed

Classification

Impact Rate:	2/5
Likelihood Rate:	3/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation:

It is recommended to apply the same access control modifiers (`forbiddenForRegistrar()` and `forbiddenForOperations()`) to the `approve()` function to align its behavior with the documented design and maintain consistency with the `increaseAllowance()` implementation.

Resolution:

The Finding was fixed in file hash `428dfbaf44f1f8f74809e1183a5b4852b0de217e49ec0083dc4bfe01e9de21`. The requirement from the `approve()` NatSpec comment was removed when the spender is either the registrar operator or the operations operator.

[F-2025-13671](#) - `wipeFrozenAddress()` Not Restricted by Pause State

- Low

Description: The `wipeFrozenAddress()` function allows the registrar to burn the entire balance of any frozen address:

```
function wipeFrozenAddress(address _addr)
    external
    override
    onlyRegistrar
    onlyFrozen(_addr)
    returns (bool)
{
    super._burn(_addr, balanceOf(_addr));
    return true;
}
```

Unlike other critical token-flow functions (e.g., `mint()` and `burn()`), this function **does not include the** `onlyWhenNotPaused` **modifier**.

As a result, the registrar can still execute token burns even when the contract is paused — a state typically meant to halt all balance-affecting operations during emergencies or investigations.

Assets:

- contracts/smartCoin/SmartCoin.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]

Status: Mitigated

Classification

Impact Rate:	5/5
Likelihood Rate:	1/5
Exploitability:	Dependent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: Add the `onlyWhenNotPaused` modifier to `wipeFrozenAddress()` to ensure consistency with other state-changing functions such as `mint()` and `burn()`.

Resolution:

The SG Forge team stated that this behavior is intentional. For regulatory reasons, they must retain the ability to wipe assets of sanctioned addresses at any time, even when the token is paused.

[F-2025-13634](#) - Cache Variable Array Length In For Loop - Info

Description:

Failing to cache the array length when iterating through arrays in Solidity can have significant performance and gas cost implications. In Solidity, array lengths can change during execution due to external calls or storage modifications. When the array length is not cached before entering a loop, it is recomputed with each iteration, leading to unnecessary gas consumption.

```
for (uint256 i = 0; i < _addrs.length; i++) {  
  
for (uint256 i = 0; i < _addrs.length; i++) {  
  
for (uint i = 0; i < addrs.length; i++) {  
  
for (uint i = 0; i < addrs.length; i++) {  
  
for (uint256 i = 0; i < addrs.length; i++) {  
  
for (uint256 i = 0; i < addrs.length; i++) {
```

Assets:

- contracts/smartCoin/AccessControlUpgradeable.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]

Status:

Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

To enhance performance and reduce gas costs, cache the array length before entering a for loop in Solidity. This approach prevents repeated computation of the array length and mitigates the risk of reentrancy attacks due to array length changes during loop execution.

Resolution:

This Finding is accepted by the SG Forge team.

[F-2025-13636](#) - Repeated state reads in role-accept functions - Info

Description:

In the three role-accept functions, the same state variable is read multiple times in the same context (comparison → mapping write → event). Caching the value in a local variable removes redundant loads and slightly reduces gas per call.

```
function acceptRegistrarRole() external {
    if (newRegistrar != msg.sender) revert UnauthorizedRegistrar();
    hasAcceptedRole[newRegistrar] = true;
    emit AcceptedRegistrarRole(newRegistrar);
}

function acceptOperationsRole() external {
    if (newOperations != msg.sender) revert UnauthorizedOperations();
    hasAcceptedRole[newOperations] = true;
    emit AcceptedOperationsRole(newOperations);
}

function acceptTechnicalRole() external {
    if (newTechnical != msg.sender) revert UnauthorizedTechnical();
    hasAcceptedRole[newTechnical] = true;
    emit AcceptedTechnicalRole(newTechnical);
}
```

Assets:

- contracts/smartCoin/AccessControlUpgradeable.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]

Status:

Accepted

Classification

Impact Rate:	1/5
Likelihood Rate:	1/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation:

Cache the state variable in a local and reuse it for the require/revert check, mapping write, and event emission.

Resolution:

This Finding is accepted by the SG Forge team.

[F-2025-13638](#) - Unneeded initializations of uint256 and bool variable to 0/false - Info

Description:

In Solidity, it is common practice to initialize variables with default values when declaring them. However, initializing `uint256` variables to `0` and `bool` variables to `false` when they are not subsequently used in the code can lead to unnecessary gas consumption and code clutter. This issue points out instances where such initializations are present but serve no functional purpose.

```
for (uint256 i = 0; i < _addrs.length; i++) {  
  
for (uint256 i = 0; i < _addrs.length; i++) {  
  
uint16 count = 0;  
  
for (uint i = 0; i < addrs.length; i++) {  
  
uint16 count = 0;  
  
for (uint i = 0; i < addrs.length; i++) {  
  
for (uint256 i = 0; i < addrs.length; i++) {  
  
for (uint256 i = 0; i < addrs.length; i++) {
```

Assets:

- contracts/smartCoin/AccessControlUpgradeable.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]

Status:

Accepted

Classification

Impact Rate:	1/5
Likelihood Rate:	1/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation:

It is recommended not to initialize integer variables to `0` to and boolean variables to `false` to save some Gas.

Resolution:

This Finding is accepted by the SG Forge team.

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.

As part of Hacken's ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.

The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at <https://hacken.io/audits/>.

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. Definitions

Severities

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](https://github.com/hacken/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.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

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

Scope Details	
File	smartcoin-ethereum-smartcoin_v4_20251007.zip
SHA3-256	7e42e1875bf791ca742912b8773147b8d8729a834d3d880c8cbee083a3e908d6
Commit Id	2b796273dce622976dd470032150e7a21a4e256e
Remediation	smartcoin-ethereum-smartcoin_v4_20251027_hacken_fix_1.zip
Remediation SHA3-256	428dfbfafc44f1f8f74809e1183a5b4852b0de217e49ec0083dc4bfe01e9de21
Remediation Commit Id	e8bc554bd28ffa4bef5765307655bcd094364115
Whitepaper	N/A
Requirements	README.md
Technical Requirements	README.md

Asset	Type
contracts/smartCoin/AccessControlDataLayout.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]	Smart Contract
contracts/smartCoin/AccessControlUpgradeable.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]	Smart Contract
contracts/smartCoin/SmartCoin.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]	Smart Contract
contracts/smartCoin/SmartCoinDataLayout.sol [smartcoin-ethereum-smartcoin_v4_20251007.zip]	Smart Contract

Appendix 3. Additional Valuables

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.

Frameworks and Methodologies

This security assessment was conducted in alignment with recognised penetration testing standards, methodologies and guidelines, including the [NIST SP 800-115 - Technical Guide to Information Security Testing and Assessment](#), and the [Penetration Testing Execution Standard \(PTES\)](#). These assets provide a structured foundation for planning, executing, and documenting technical evaluations such as vulnerability assessments, exploitation activities, and security code reviews. Hacken's internal penetration testing methodology extends these principles to Web2 and Web3 environments to ensure consistency, repeatability, and verifiable outcomes.