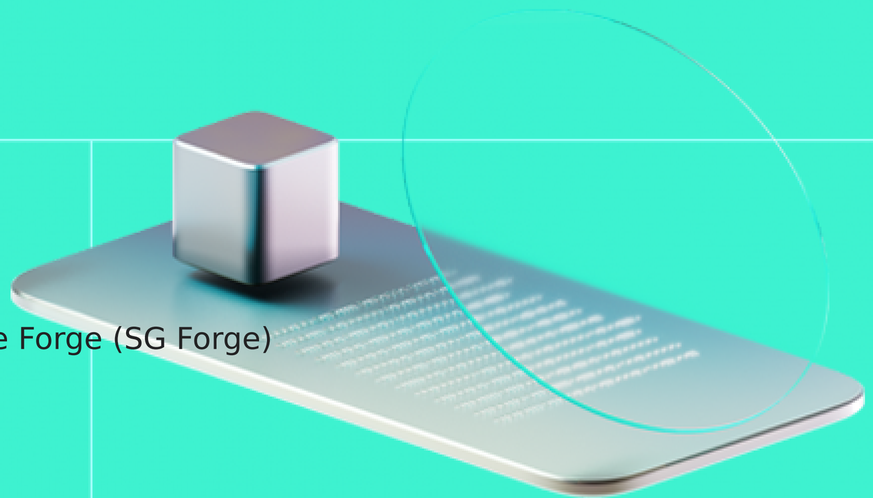
Smart Contract Code Review And Security Analysis Report



Customer: Societe Generale Forge (SG Forge)

Date: 04/09/2025

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.

SG Forge is a smart contract system built on the Soroban platform, a framework for the Stellar blockchain, designed to manage asset administration, account authorization, and governance through an integrated connection with the Stellar Asset Contract (SAC).

Document

Name	Smart Contract Code Review and Security Analysis Report for Societe Generale Forge (SG Forge)
Audited By	Panagiotis Konstantinidis, Kerem Solmaz
Approved By	Ataberk Yavuzer
Website	https://www.sgforge.com
Changelog	27/08/2025 - Preliminary Report
	04/09/2025 - Final Report
Platform	Soroban (Stellar)
Language	Rust
Tags	Incentives
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	Shared privately
Initial Commit	e8eed30068802e6dcaaf010c39409907171ed1ce
Remediation Commit	ee858043c7e6cf74291debf3372b00604f7f366f

Audit Summary

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

2	0	1	1
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	1
Low	1

Vulnerability	Severity	Status
F-2025-12378 - Lack of Authentication in authorize_trustline Allows Bypassing Admin Deauthorization	Medium	Accepted
F-2025-12431 - Asymmetric Logic in Freeze/Unfreeze Results in Unconditional Authorization	Low	Mitigated

Documentation quality

- Initial setup and build steps are provided in the README.md file.
- Missing formal functional and architectural documentation.
- Missing detailed description of system behaviour and logic flow.
- Inline comments are minimal and do not sufficiently explain the contract logic.
- Partial discrepancies exist between the actual code implementation and the provided README.md documentation.

Code quality

- Codebase uses Soroban SDK idiomatically and adheres to Rust best practices.
- Codebase follows a modular structure with separation of code files.
- Authorization checks and error handling are consistently applied.

Test coverage

Code coverage off the project is **N/A**.

- The test score is N/A since there no standard Soroban test coverage tool is available.
- Basic CLI tests were included and executed using Soroban test to verify core functionality.
- Meaningful integration and logic tests are written in Rust using `#[test]` functions and mock environments.

Table of Contents

System Overview **6**

Privileged Roles 6

Potential Risks **8**

Findings **9**

Vulnerability Details 9

Disclaimers 15

Appendix 1. Definitions **16**

Severities 16

Potential Risks 16

Appendix 2. Scope **17**

Appendix 3. Additional Valuables **18**

System Overview

SG Forge is a Stellar-based token management system leveraging Soroban smart contracts to provide administrative control over trustline authorizations, account restrictions, and minting operations.

The project is developed using the Loam SDK and consists of key modules organized under `lib.rs`, `admin.rs`, `sac.rs`, `errors.rs` and `utils.rs`. These modules implement the administrative logic (`admin.rs`), Stellar Asset Contract (SAC) interaction wrappers (`sac.rs`), reusable helper functions (`utils.rs`), error definitions (`errors.rs`), and the main contract entrypoint (`lib.rs`) for contract dispatch.

Core Functionality

The system's administrative layer provides the following operations:

- `authorize_trustline` / `deauthorize_trustline`: Grant or revoke authorization for individual accounts to interact with the token.
- `batch_pause` / `batch_unpause`: Admin revoke or restore authorization for multiple accounts in bulk.
- `add_banned_accounts` / `remove_banned_accounts`: Allows the admin to maintain a ban list of restricted accounts.
- `pause` / `unpause`: Admin toggle the operational state of the contract.
- `mint_to_account`: Allows mint new tokens directly to authorized accounts.
- `freeze_accounts` / `unfreeze_accounts`: Allows the admin to ban and authorize accounts by managing both banned and authorized flags.
- `clawback`: Admin forcefully burns tokens from any account.

Privileged roles

The admin account holds exclusive authority over the system and is the only role permitted to invoke administrative instructions. This account has the ability to:

- Authorize or deauthorize trustlines, determining which accounts can interact with the token.
- Pause and unpause the contract, enabling or disabling sensitive operations.
- Add or remove banned accounts from the contract's internal ban list.
- Mint tokens directly to authorized accounts.
- Freeze or unfreeze accounts in bulk, updating both SAC authorization and ban status.
- Execute clawbacks, forcefully burning tokens from any account.

The implications of these privileges are significant, as the admin controls the token's authorization model, supply expansion, and the enforcement of bans or freezes. Any misuse or compromise of the admin key could result in arbitrary minting, unauthorized clawbacks, censorship of users, or full suspension of operations through pausing. Therefore, it is essential

that the admin key is securely managed, ideally with multisig or hardware-backed key management practices to mitigate centralized risk.

Potential Risks

- **Scope Definition and Security Guarantees:** This audit only covers the shared files (`lib.rs`, `sac.rs`, `utils.rs`, `admin.rs`, and `errors.rs`) that form the core logic of the SAC token management system. Any logic outside of this scope, such as frontend applications, deployment scripts, or external integrations, is not covered and may introduce unforeseen vulnerabilities.
- **System Reliance on External Contracts:** The contract acts as a wrapper around Soroban's Stellar Asset Contract (SAC). Its behavior depends on the correct and secure implementation of the underlying SAC system, which is not part of this audit. Any flaws in the SAC could directly impact this contract's correctness and reliability.
- **Centralized Minting to a Single Address:** Token minting is fully controlled by the admin, who can arbitrarily mint to any account. If the admin key is compromised or misused, this creates a risk of excessive or malicious token issuance.
- **Centralized Control of Minting and Burning:** The admin exclusively governs minting and clawback (forced burns). This centralization poses a significant threat to the token economy if governance or key management practices are weak.
- **Coarse-grained Authorization Model:** The `require_admin()` check grants the admin wide-reaching control, including minting, banning/unbanning, freezing/unfreezing, and pausing/unpausing. If the admin key is compromised, attackers could execute all critical operations without restriction.
- **Insufficient Multi-signature Controls for Critical Functions:** All sensitive operations (e.g., minting, clawback, freeze/unfreeze, pause/unpause) can be performed by a single key holder. Lack of multisig creates a single point of failure.
- **Absence of Time-lock Mechanisms:** There is no enforced delay for critical operations such as minting, clawback, or pausing. This increases the risk of immediate malicious changes without oversight or recourse.
- **Single Point of Failure:** The contract depends entirely on one centralized admin role. Misuse, loss, or compromise of the admin key could compromise the integrity of the contract and user balances.
- **Administrative Key Control Risks:** Critical functions such as account authorization, minting, freezing, and clawback depend only on the admin key. Without strong key management, this presents a significant risk vector for exploitation.

Findings

Vulnerability Details

F-2025-12378 - Lack of Authentication in authorize_trustline Allows Bypassing Admin Deauthorization - Medium

Description:

The contract implements the `authorize_trustline` function that enables accounts to authorize themselves for interaction with the Stellar Asset Contract (SAC). This function is publicly accessible and does not enforce any signature or authentication constraint tied to the caller. As a result, it can be invoked by any user to authorize any account, including accounts that have previously been deauthorized by an administrator.

```
    fn authorize_trustline(&mut self, account: Address) -> Result<(), Error>
{
        if self.paused() {
            return Err(Error::ContractPaused);
        }
        if Self::operator() == account {
            return Err(Error::CannotAuthorizeOperator);
        }
        if env().current_contract_address() == account {
            return Err(Error::CannotAuthorizeAdminContract);
        }
        if self.b.has(account.clone()) {
            return Err(Error::AccountBanned);
        }
        if self.sac().authorized(&account) {
            return Err(Error::AccountAlreadyAuthorized);
        }
        self.sac().set_authorized(&account, true)
    }
```

The `deauthorize_trustline` function is protected by the `require_admin()` check and is intended to allow the admin to remove trustline permissions from specific accounts.

```
    fn deauthorize_trustline(&mut self, account: Address) -> Result<(), Error
> {
        require_admin();
```

```
        self.sac().set_authorized(&account, false)
    }
```

However, due to the open access contract of `authorize_trustline` function, an account that has been deauthorized can immediately reauthorize itself by simply calling the function again. This behavior undermines the purpose of administrative revocations and breaks the expected control guarantees around trustline management. Although the logic may be designed to support decentralized onboarding and ease of access, it introduces a crucial gap in authority enforcement, particularly when dealing with unauthorized accounts.

As a result, this oversight allows previously deauthorized users to regain authorization without admin consent, which may lead to abuse scenarios such as transferring, or redeeming tokens without proper authorization. In environments where strong administrative control over trustline permissions is expected, this weakens the program's access model and could result in unintended economic consequences.

Assets:

- admin.rs [smartcoin-stellar_20250721.tar.gz]

Status: Accepted

Classification

Impact Rate: 3/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: To preserve the authority of administrative actions, the program should introduce an additional check in `authorize_trustline` to prevent reauthorization of accounts that were explicitly deauthorized. One possible approach is to maintain a mapping of such accounts (e.g., a persistent `deauthorized` list) and require administrative clearance before they can be reauthorized. This would ensure that trustline access cannot be regained unilaterally by the account itself and must go through the intended administrative process.

Resolution: **Accepted**. The client is aware of the finding and stated:

> We will accept this because the use case does not really exist.
> The only accounts that should not have an authorized trustline are the frozen/banned accounts.
> And these accounts won't be able to regain access by calling the authorize_trustline function.
> In addition, this function has no authentication on purpose so that client can authorize their trustline themselves, or we can do it for them if they have difficulties making a smart contract call.

Evidences

Any account can authorize any other account's trustline without consent.

Reproduce:

To reproduce:

1. Move `authorization_bypass_test.rs` under `contracts/sac_admin/src/`
2. Make sure your working directory is `contracts/sac_admin/`
3. Run command `cargo test -p sac-admin-contract --lib -- authorization_bypass --test-threads=1 --nocapture`

Results:

```
running 3 tests
test authorization_bypass_focused_test::critical_vulnerability_any_account_can_authorize_any_other ... ✓ PASS: Bob starts unauthorized
✓ PASS: Authorization succeeded without Bob's consent
✓ PASS: Bob is now authorized without permission
✓ PASS: Bob can receive tokens without consent
✓ PASS: Bob now has unwanted tokens (balance: 100)
ok
test authorization_bypass_focused_test::vulnerability_persists_in_multiple_victim_scenario ... Writing test snapshot file for test "authorization_bypass_focused_test::vulnerability_persists_in_multiple_victim_scenario" to "test_snapshots/authorization_bypass_focused_test/vulnerability_persists_in_multiple_victim_scenario.1.json".
✓ PASS: Victim Contract(CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAMDR4) starts unauthorized
✓ PASS: Victim Contract(CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAOLZM) starts unauthorized
✓ PASS: Victim Contract(CAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAARQG5) starts unauthorized
✓ PASS: Attacker successfully authorized victim without consent
✓ PASS: Victim is now authorized without permission
```

```
✓ PASS: Attacker successfully authorized victim without consent
✓ PASS: Victim is now authorized without permission
✓ PASS: Attacker successfully authorized victim without consent
✓ PASS: Victim is now authorized without permission
ok
test authorization_bypass_focused_test::why_customer_mitigation_was_insuffici
ent ... Writing test snapshot file for test "authorization_bypass_focused_tes
t::why_customer_mitigation_was_insufficient" to "test_snapshots/authorization
_bypass_focused_test/why_customer_mitigation_was_insufficient.1.json".
✓ PASS: Victim starts unauthorized
✓ PASS: Attack succeeds - victim authorized without consent
✓ PASS: Victim is now authorized in vulnerability window
✓ PASS: Admin banned the victim (reactive response)
✓ PASS: Victim authorization revoked after freeze
✓ PASS: Subsequent attacks correctly blocked (victim is banned)
ok
test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 13 filtered out;
finished in 0.07s
```

Files: authorization_bypass_test.rs

F-2025-12431 - Asymmetric Logic in Freeze/Unfreeze Results in Unconditional Authorization - Low

Description:

The contract implements freeze and unfreeze functionality to allow administrators to temporarily revoke and restore account authorization. The `freeze_account()` function removes authorization from accounts and adds them to the banned list, while `unfreeze_account()` is intended to reverse this process for previously frozen accounts.

The `unfreeze_accounts()` function contains asymmetric logic that unconditionally authorizes any account provided in the input vector, regardless of whether the account was previously frozen or authorized. This allows administrators to inadvertently grant authorization to accounts that should not have access to the system.

```
fn freeze_accounts(&mut self, accounts: Vec<Address>) -> Result<(), Error> {
    require_admin();
    for account in accounts.iter() {
        if self.sac().authorized(&account) {                 // conditional r
evoke
            self.sac().set_authorized(&account, false)?;
        }
        self.ban_account(account);
    }
    Ok(())
}

fn unfreeze_accounts(&mut self, accounts: Vec<Address>) -> Result<(), Error>
{
    require_admin();
    for account in accounts.iter() {
        self.sac().set_authorized(&account, true)?;          // unconditional
grant
        self.unban_account(account);
    }
    Ok(())
}
```

The `freeze_accounts()` function only processes accounts that are currently authorized, ensuring it only affects accounts with existing permissions. In contrast, `unfreeze_accounts()` unconditionally authorizes all provided accounts without verifying their previous state. This

asymmetric behavior allows unauthorized accounts to gain authorization by being included in an unfreeze operation.

Assets:

- admin.rs [smartcoin-stellar_20250721.tar.gz]

Status: Mitigated

Classification

Impact Rate: 3/5

Likelihood Rate: 3/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Implement symmetric validation logic in `unfreeze_accounts()` to ensure only previously frozen accounts can be unfrozen:

```
fn unfreeze_accounts(&mut self, accounts: Vec<Address>) -> Result<(), Error>
{
    require_admin();
    for account in accounts.iter() {
        if !self.sac().authorized(&account) {
            self.sac().set_authorized(&account, true)?;
        }
        self.unban_account(account);
    }
    Ok(())
}
```

Consider maintaining a separate storage mapping for frozen account state to explicitly track which accounts were frozen and can be legitimately unfrozen.

Resolution: **Mitigated** at commit `ee85804`. The freeze and unfreeze logic was changed to unconditional authorization and deauthorization calls, which simplifies the flow but does not enforce prior state checks.

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

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	
Repository	Shared privately
Initial Commit	e8eed30068802e6dcaaf010c39409907171ed1ce
Remediation Commit	ee858043c7e6cf74291debf3372b00604f7f366f
Whitepaper	N/A
Requirements	N/A
Technical Requirements	README.md

Asset	Type
admin.rs [smartcoin-stellar_20250721.tar.gz]	Smart Contract
error.rs [smartcoin-stellar_20250721.tar.gz]	Smart Contract
lib.rs [smartcoin-stellar_20250721.tar.gz]	Smart Contract
sac.rs [smartcoin-stellar_20250721.tar.gz]	Smart Contract
util.rs [smartcoin-stellar_20250721.tar.gz]	Smart Contract

Appendix 3. Additional Valuables

Verification of System Invariants

During the audit of SAC Admin Contract, Hacken followed its methodology by performing fuzz-testing on the project's main functions. cargofuzz, a tool used for fuzz-testing, was employed to check how the protocol behaves under various inputs.. Due to the complex and dynamic interactions within the protocol, unexpected edge cases might arise. Therefore, it was important to use fuzz-testing to ensure that several system invariants hold true in all situations.

Custom fuzz scripts were implemented using `libfuzzer` and Soroban's in-memory testing environment. These fuzzers exercised full end-to-end flows of the admin contract where the setup built a stateful, sequence-driven fuzz harness that stressed the SAC Admin contract's core behaviors - init, authorize trustline / deauthorize trustline, single and batch ban/unban, pause/unpause, freeze/unfreeze, and mint/clawback -under randomized operation scripts. Each script executed actions in different orders and repetitions, asserting the expected outcomes after every step. This approach was designed to surface bugs that appear under complex combinations (e.g., pause → ban → unfreeze → mint → deauth). Across the file **12** fuzzing tests were performed to validate correctness and safety under unexpected inputs.

Fuzzing testing (file: fuzz_testing_sac_adming.rs)	Test Result	Run Time
First init must authorize the designated operator in SAC	Passed	10 Min
Subsequent init calls must not succeed (once-only)	Passed	10 Min
Contract must never authorize its own address	Passed	10 Min
Paused state must block new authorizations	Passed	10 Min
Banned accounts must never be authorized	Passed	10 Min
Re-authorizing already authorized accounts must fail	Passed	10 Min
Single ban/unban operations must update state consistently	Passed	10 Min
Batch ban/unban operations must update all targeted accounts consistently	Passed	10 Min
Freeze must enforce ban + deauthorization	Passed	10 Min
Unfreeze must enforce unban + authorization	Passed	10 Min
Mint must only affect target balance and adjust supply correctly	Passed	10 Min
Clawback must only affect target balance and adjust supply correctly	Passed	10 Min

Additionally, complementary fuzzing scripts were implemented to stress more granular aspects of the admin contract's logic. These included targeted fuzzers for batch operations (4 tests), amount edge cases (5 tests) and clawback operations (5 tests). In total, **14** additional fuzzing tests were verified over millions of runs, complementing the stateful end-to-end harness with focused stress on arithmetic correctness and batch handling.

Fuzzing testing	Test Result	Run Count
Batch Operations Testing (file: fuzz_batch_operations.rs)		
Batch operations must not exceed contract limits (50 accounts max)	Passed	4M+
Batch address generation must not cause arithmetic overflow	Passed	4M+
Vec operations during batch processing must remain consistent	Passed	4M+
Large batch operations must not cause memory issues or panics	Passed	4M+
Amount Edge Cases Testing (file: fuzz_amount_edge_cases.rs)		
Arithmetic operations must handle i128MIN and i128MAX safely	Passed	1M+
Balance calculations must never underflow or overflow	Passed	1M+
Batch amount distribution must preserve value conservation	Passed	1M+
Transfer operations must maintain total value consistency	Passed	1M+
Division operations must satisfy mathematical identities	Passed	1M+
Clawback Operations Testing (file: fuzz_clawback.rs)		
Clawback amount validation must handle negative and edge case inputs safely	Passed	3M+
Balance calculations must prevent negative balances in all scenarios	Passed	3M+
Sequential clawback operations must maintain balance consistency	Passed	3M+
Boundary value testing must not cause arithmetic overflow	Passed	3M+
Address generation in clawback context must not cause memory issues	Passed	3M+

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.