// Security Assessment

08.11.2025 - 08.18.2025

Stellar Smart Account

Crossmint

HALBORN

Stellar Smart Account - Crossmint

Prepared by: HALBORN

Last Updated 09/04/2025

Date of Engagement: August 11th, 2025 - August 18th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
8	0	2	2	3	1

TABLE OF CONTENTS

1. Introduction
2. Assessment summary
3. Test approach and methodology
4. Risk methodology
5. Scope
6. Assessment summary & findings overview
7. Findings & Tech Details
   - 7.1 Plugin uninstall does not persist removal
   - 7.2 Admin count storage fails on subsequent updates
   - 7.3 Plugin on_auth invoked without failure isolation enables account-wide dos
   - 7.4 Missing webauthn challenge binding and payload usage on secp256r1 verification
   - 7.5 Missing policy on_revoke on signer revocation
   - 7.6 Policy callbacks not invoked on signer update
   - 7.7 Silent swallowing of external policy callback failures
   - 7.8 Missing event emissions for initialization and upgrades

1. Introduction

`Crossmint` engaged Halborn to conduct a security assessment of the **Stellar Smart Account** contracts, beginning on August 12th, 2025, and ending on August 19th, 2025. This security assessment was scoped to the smart contracts located in the **Stellar Smart Account GitHub** repository. Commit hashes and further details can be found in the *Scope* section of this report.

The **Stellar Smart Account** project is an advanced, fully on-chain account management system for the Stellar blockchain, implemented in Rust and compiled to WebAssembly. It replaces the standard Stellar signature-and-threshold model with a programmable, role-based authorization framework. Key features include multi-signature support (Ed25519, Secp256r1/WebAuthn), role separation (Admin/Standard), plugin extensibility, external policy enforcement (e.g., deny-list) and secure upgrade/migration capabilities.

2. Assessment Summary

This assessment spanned one week and involved a dedicated security team with deep expertise in Rust and smart contract security. The principal goal was to verify that the Stellar Smart Account implementation provided robust, resilient, and auditable mechanisms for multi-signature, role-based access, plugin extensibility, and secure upgradeability. The system's foundational features, including signer management, policy enforcement, and upgrade workflows, generally demonstrated mature engineering. However, several critical issues were identified requiring remediation prior to production deployment.

Notable findings include improper persistence of plugin removal, failure isolation in plugin callbacks, missing event emissions for key lifecycle actions, storage management errors that break admin updates, absent WebAuthn challenge binding, and unhandled TTL expiration for critical state. Immediate attention to these gaps will dramatically enhance operational reliability, security, and auditability.

All the findings have been addressed by the Crossmint team.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. Manual testing was emphasized to uncover flaws in logic, process, and contract interaction, while automated tools supported the detection of dependency vulnerabilities and unsafe coding patterns.

The following phases and associated tools were used during the assessment:

- Research into the architecture, purpose, and operational model of the Stellar Smart Account system.
- Manual code review and walk-through of all contracts.
- Verification of initialization flows and prevention of double-init or bypass conditions.
- Analysis of upgrade and migration flows.
- Review of cross-contract interactions and fail-closed behavior on errors.
- Scanning of Rust code for vulnerabilities and unsafe usage.
- Review and improvement of integration tests.
- Verification of integration test execution and addition of new ones where required.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: stellar-smart-account

(b) Assessed Commit ID: 3b79e1b

(c) Items in scope:

- contracts/smart-account-interfaces/src/policy.rs
- contracts/smart-account-interfaces/src/lib.rs
- contracts/smart-account-interfaces/src/plugin.rs
- contracts/initializable/src/lib.rs
- contracts/contract-factory/src/lib.rs
- contracts/storage/src/lib.rs
- contracts/smart-account/src/events.rs
- contracts/smart-account/src/interface.rs
- contracts/smart-account/src/constants.rs
- contracts/smart-account/src/auth/core/authorizer.rs
- contracts/smart-account/src/auth/core/mod.rs
- contracts/smart-account/src/auth/policies/mod.rs
- contracts/smart-account/src/auth/mod.rs
- contracts/smart-account/src/auth/signer.rs
- contracts/smart-account/src/auth/permissions.rs
- contracts/smart-account/src/auth/proof.rs
- contracts/smart-account/src/auth/signers/secp256r1.rs
- contracts/smart-account/src/auth/signers/mod.rs
- contracts/smart-account/src/auth/signers/ed25519.rs
- contracts/smart-account/src/auth/policy/time_based.rs
- contracts/smart-account/src/auth/policy/interface.rs
- contracts/smart-account/src/auth/policy/mod.rs
- contracts/smart-account/src/auth/policy/external.rs
- contracts/smart-account/src/error.rs
- contracts/smart-account/src/config.rs
- contracts/smart-account/src/lib.rs
- contracts/smart-account/src/account.rs
- contracts/smart-account/src/plugin.rs
- contracts/upgradeable/src/lib.rs

REMEDIATION COMMIT ID:

- 8e047f1
- 9ac2c03
- https://github.com/Crossmint/stellar-smart-account/pull/69
- https://github.com/Crossmint/stellar-smart-account/pull/75
- a3525f9
- https://github.com/Crossmint/stellar-smart-account/pull/73
- https://github.com/Crossmint/stellar-smart-account/pull/69/files
- a535753

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	2	3	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PLUGIN UNINSTALL DOES NOT PERSIST REMOVAL	HIGH	SOLVED - 08/17/2025
ADMIN COUNT STORAGE FAILS ON SUBSEQUENT UPDATES	HIGH	SOLVED - 08/17/2025
PLUGIN ON_AUTH INVOKED WITHOUT FAILURE ISOLATION ENABLES ACCOUNT-WIDE DOS	MEDIUM	SOLVED - 08/17/2025
MISSING WEBAUTHN CHALLENGE BINDING AND PAYLOAD USAGE ON SECP256R1 VERIFICATION	MEDIUM	SOLVED - 08/20/2025
MISSING POLICY ON_REVOKE ON SIGNER REVOCATION	LOW	SOLVED - 08/20/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
POLICY CALLBACKS NOT INVOKED ON SIGNER UPDATE	LOW	SOLVED - 08/20/2025
SILENT SWALLOWING OF EXTERNAL POLICY CALLBACK FAILURES	LOW	SOLVED - 08/20/2025
MISSING EVENT EMISSIONS FOR INITIALIZATION AND UPGRADES	INFORMATIONAL	SOLVED - 08/20/2025

7. FINDINGS & TECH DETAILS

7.1 PLUGIN UNINSTALL DOES NOT PERSIST REMOVAL

// HIGH

Description

The `uninstall_plugin` function removes plugins from the in-memory `existing_plugins` map but does not persist the updated plugin list to contract storage. This means that a supposedly uninstalled plugin can remain effectively active between contract invocations, still receiving `on_auth` callbacks and executing its logic.

This vulnerability could lead to several unforeseen issues, including:

- **Authorization bypass:** an uninstalled plugin could continue to authorize operations unintentionally.
- **Execution of malicious code:** an attacker could exploit this to maintain influence within the authorization process.
- **Inconsistent contract state:** external observers may believe the plugin is uninstalled, while it still affects execution.

Code Location

Code of `uninstall_plugin` function from **contracts/smart-account/src/account.rs** file:

```
196 fn uninstall_plugin(env: &Env, plugin: Address) -> Result<(), Error> {
197     Self::require_auth_if_initialized(env);
198
199     let storage = Storage::instance();
200     let mut existing_plugins = storage
201         .get::<Symbol, Map<Address, ()>>(env, &PLUGINS_KEY)
202         .unwrap();
203
204     if !existing_plugins.contains_key(plugin.clone()) {
205         return Err(Error::PluginNotFound);
206     }
207     existing_plugins.remove(plugin.clone());
208
209     // Counterwise to install, we don't want to fail if the plugin's on_uninstall fails,
210     // as it would prevent an admin from uninstalling a potentially-malicious plugin .
211     let res = SmartAccountPluginClient::new(env, &plugin)
212         .try_on_uninstall(&env.current_contract_address());
213     match res {
214         Ok(inner) => {
215             if inner.is_err() {
216                 env.events().publish(
217                     (TOPIC_PLUGIN, VERB_UNINSTALL_FAILED),
218                     PluginUninstallFailedEvent {
219                         plugin: plugin.clone(),
220                     },
221                 );
222             }
223         }
224         Err(_) => {
225             env.events().publish(
226
```

```
                    (TOPIC_PLUGIN, VERB_UNINSTALL_FAILED),
                    PluginUninstallFailedEvent {
                        plugin: plugin.clone(),
                    },
                );
            }
        }

        env.events().publish(
            (TOPIC_PLUGIN, VERB_UNINSTALLED),
            PluginUninstalledEvent { plugin },
        );

        Ok(())
    }
```

Proof of Concept

Scenario

Deploy a SmartAccount with a single admin signer and a dummy plugin whose `on_auth` increments a counter. Install the plugin, then uninstall it. Next, trigger an authorization (check_auth). If uninstall persisted correctly, the plugin should no longer be invoked and the counter should remain at 0. Instead, the counter increases, showing the plugin still receives `on_auth` after removal.

Test Code

```
#![cfg(test)]

extern crate std;

use soroban_sdk::{
    auth::Context, contract, contractimpl, symbol_short,
    testutils::{Address as _, BytesN as _},
    vec, Address, BytesN, Env, IntoVal, Symbol, Vec,
  };

use crate::{
    account::{SmartAccount, SmartAccountClient},
    auth::{
        permissions::SignerRole,
        proof::{SignatureProofs, SignerProof},
    },
    error::Error,
    tests::test_utils::{get_token_auth_context, setup, Ed25519TestSigner, TestSignerTrait as _},
};

// ----------------------------------------------------------------------------
// Dummy plugin contract that increments a counter on every on_auth
// ----------------------------------------------------------------------------

const COUNT: Symbol = symbol_short!("cnt");

#[contract]
pub struct DummyPlugin;

#[contractimpl]
impl DummyPlugin {
    pub fn on_install(_env: &Env, _source: Address) {}

    pub fn on_uninstall(_env: &Env, _source: Address) {}
```

```
    pub fn on_auth(env: &Env, _source: Address, _contexts: Vec<Context>) {
        let count: u32 = env
            .storage()
            .instance()
            .get(&COUNT)
            .unwrap_or(0);
        env.storage().instance().set(&COUNT, &(count + 1));
    }

    pub fn get_count(env: &Env) -> u32 {
        env.storage().instance().get(&COUNT).unwrap_or(0)
    }
}

// ---------------------------------------------------------------------------
// PoC: Uninstall does not persist removal, plugin still receives on_auth
// ---------------------------------------------------------------------------

#[test]
fn poc_uninstall_plugin_does_not_persist_removal() {
    let env = setup();
    env.mock_all_auths();

    // Deploy SmartAccount with one admin signer
    let admin = Ed25519TestSigner::generate(SignerRole::Admin);
    let smart_account_id = env.register(
        SmartAccount,
        (vec![&env, admin.into_signer(&env)], Vec::<Address>::new(&env)),
    );
    let smart_account = SmartAccountClient::new(&env, &smart_account_id);

    // Deploy dummy plugin
    let plugin_id = env.register(DummyPlugin, ());

    // Install plugin
    smart_account.install_plugin(&plugin_id);

    // Uninstall plugin (BUG: removal not persisted to storage)
    smart_account.uninstall_plugin(&plugin_id);

    // Trigger __check_auth to see if plugin.on_auth still runs
    let payload = BytesN::random(&env);
    let (admin_key, admin_proof) = admin.sign(&env, &payload);
    let auth_payloads = SignatureProofs(soroban_sdk::map![&env, (admin_key, admin_proof)]);

    // Should succeed auth; this will also call plugin.on_auth if plugin remains in storage
    env.try_invoke_contract_check_auth::<Error>(
        &smart_account_id,
        &payload,
        auth_payloads.into_val(&env),
        &vec![&env, get_token_auth_context(&env)],
    )
    .unwrap();

    // Read plugin counter; if uninstall had persisted, count would be 0
    let plugin_client = DummyPluginClient::new(&env, &plugin_id);
    let count_after = plugin_client.get_count();

    // PoC: demonstrates bug - plugin.on_auth was still invoked after uninstall
    assert_eq!(count_after, 1);
}
```

Result

After uninstalling the plugin, invoking authorization increments the plugin's counter to `1`, proving `on_auth` was executed. The expected behavior, if removal were persisted, is a counter value of 0.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

It is recommended to persist the updated plugin registry after removal by calling `storage.update::<Symbol, Map<Address, ()>>(env, &PLUGINS_KEY, &existing_plugins)` to ensure that the removal is persisted to storage and prevents any uninstalled plugin from continuing to participate in the contract's authorization flow.

Remediation Comment

SOLVED: The **Crossmint team** resolved this issue by updating the missed storage to include the missing data.

Remediation Hash

https://github.com/Crossmint/stellar-smart-account/commit/8e047f13efebbef9b272e9921a64aa47e47379ed

7.2 ADMIN COUNT STORAGE FAILS ON SUBSEQUENT UPDATES // HIGH

Description

The functions that adjust the admin count (`add_signer` and `update_signer`) use `Storage::store()` to write `ADMIN_COUNT_KEY` even when the key already exists. The `store()` operation returns `AlreadyExists` if the key is present, which prevents adding a second (or later) admin and can also break role transitions involving admin status. This can brick admin management by reverting calls that should simply increment or decrement the counter.

This manifests when:

- Adding a second admin in `add_signer`: the first admin sets the key to 1 via store, but the second admin attempts another store, which fails with **AlreadyExists**.
- Changing roles in `update_signer`: both the decrement (Admin→Standard) and increment (Standard→Admin) paths use store, which fails once the key exists.

This behavior can block legitimate administrative changes and brick the contract's ability to manage signer roles correctly.

Code Location

Code of `add_signer` function from contracts/smart-account/src/account.rs file:

```
94  fn add_signer(env: &Env, signer: Signer) -> Result<(), Error> {
95      Self::require_auth_if_initialized(env);
96      let key = signer.clone().into();
97      let storage = Storage::persistent();
98      storage.store::<SignerKey, Signer>(env, &key, &signer)?;
99
100     if let SignerRole::Standard(policies) = signer.role() {
101         for policy in policies {
102             policy.on_add(env)?;
103         }
104     }
105     if signer.role() == SignerRole::Admin {
106         let storage = Storage::persistent();
107         let count = storage
108             .get::<Symbol, u32>(env, &ADMIN_COUNT_KEY)
109             .unwrap_or(0);
110         storage.store::<Symbol, u32>(env, &ADMIN_COUNT_KEY, &(count + 1))?;
111     }
112     env.events()
113         .publish((TOPIC_SIGNER, VERB_ADDED), SignerAddedEvent::from(signer));
114
115     Ok(())
116 }
```

Code of `update_signer` function from contracts/smart-account/src/account.rs file:

```rust
fn update_signer(env: &Env, signer: Signer) -> Result<(), Error> {
    Self::require_auth_if_initialized(env);
    let key = signer.clone().into();
    let storage = Storage::persistent();
    let old_signer = storage
        .get::<SignerKey, Signer>(env, &key)
        .ok_or(Error::SignerNotFound)?;
    if old_signer.role() == SignerRole::Admin && signer.role() != SignerRole::Admin {
        let count = storage
            .get::<Symbol, u32>(env, &ADMIN_COUNT_KEY)
            .unwrap_or(0);
        if count <= 1 {
            return Err(Error::CannotDowngradeLastAdmin);
        }
        storage.store::<Symbol, u32>(env, &ADMIN_COUNT_KEY, &(count - 1))?;
    } else if old_signer.role() != SignerRole::Admin && signer.role() == SignerRole::Admin {
        let count = storage
            .get::<Symbol, u32>(env, &ADMIN_COUNT_KEY)
            .unwrap_or(0);
        storage.store::<Symbol, u32>(env, &ADMIN_COUNT_KEY, &(count + 1))?;
    }
    storage.update::<SignerKey, Signer>(env, &key, &signer)?;
    env.events().publish(
        (TOPIC_SIGNER, VERB_UPDATED),
        SignerUpdatedEvent::from(signer),
    );

    Ok(())
}
```

Proof of Concept

Scenario

Deploy a SmartAccount with a single admin signer. Then attempt to add a second admin via `add_signer`. The expected behavior is that the admin count increments and the operation succeeds. The PoC demonstrates that the call fails because `ADMIN_COUNT_KEY` is updated using `Storage::store (insert-only)` when the key already exists, causing an `AlreadyExists` error.

Test Code

Test added into **contracts/smart_account/src/test/signer_management_test.rs** file

```rust
#[test]
fn poc_admin_count_store_fails_on_second_admin() {
    let env = setup();

    // Deploy with one admin
    let admin1 = Ed25519TestSigner::generate(SignerRole::Admin);
    let contract_id = env.register(
        SmartAccount,
        (vec![&env, admin1.into_signer(&env)], Vec::<Address>::new(&env)),
    );

    // Prepare a second admin signer
    let admin2 = Ed25519TestSigner::generate(SignerRole::Admin);
    let admin2_signer = admin2.into_signer(&env);

    // Call add_signer as contract (auth mocked) to simulate admin operation
    env.mock_all_auths();
    let result = env.as_contract(&contract_id, || {
```

```
        SmartAccount::add_signer(&env, admin2_signer)
    });

    // Expect failure due to Storage::store on existing ADMIN_COUNT_KEY (AlreadyExists)
    assert!(result.is_err());
}
```

Result

Adding the second admin returns an error (`AlreadyExists`) instead of succeeding. This confirms that using `Storage::store` on `ADMIN_COUNT_KEY` prevents multi-admin configurations.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (7.5)

Recommendation

It is recommended to use `Storage::update()` when `ADMIN_COUNT_KEY` may already exist. Alternatively, pre-create the key during initialization and always use `update()` for subsequent modifications. This ensures that admin count changes work in multi-admin configurations without causing `AlreadyExists` errors.

Remediation Comment

SOLVED: The **Crossmint team** has addressed this issue by implementing the recommended fixes.

Remediation Hash

https://github.com/Crossmint/stellar-smart-account/commit/9ac2c0380200ecb9418067c495d4570eaaac4386

7.3 PLUGIN ON_AUTH INVOKED WITHOUT FAILURE ISOLATION ENABLES ACCOUNT-WIDE DOS

// MEDIUM

Description

The `__check_auth` function initially calls `Authorizer::check` to verify authorization. It then proceeds to iterate over all installed plugins, invoking each plugin's `on_auth` **without any** a `try_*` exception handling wrapper.

If any plugin panics or causes a revert, the error propagates and terminates the execution of `__check_auth`, . This results in the failure of the entire transaction, even if an Admin signer has already been authorized. Since all account-dependent operations route through `__check_auth`, a faulty or malicious plugin could effectively prevent further administrative actions like uninstalling the plugin also pass through authorization and will revert as well. This can block critical management workflows.

However, under the project's **trusted-plugins** assumption, switching to a **non-reverting** invocation approach does **not** compromise authorization security. The enforcement of allow/deny decisions still depends solely on `Authorizer::check`. Plugins are responsible for side-effects/telemetry, not access control. Treating plugin failures as non-fatal **helps prevent unnecessary Denial of Service (DoS)** scenarios and **improves system observability**, rather than silently blocking normal flows when a trusted plugin misbehaves.

Code Location

Code of `_check_auth` function from contracts/smart-account/src/account.rs file:

```
292 fn __check_auth(
293     env: Env,
294     signature_payload: Hash<32>,
295     auth_payloads: SignatureProofs,
296     auth_contexts: Vec<Context>,
297 ) -> Result<(), Error> {
298     Authorizer::check(&env, signature_payload, &auth_payloads, &auth_contexts)?;
299
300     let storage = Storage::instance();
301     for (plugin, _) in storage
302         .get::<Symbol, Map<Address, ()>>(&env, &PLUGINS_KEY)
303         .unwrap()
304         .iter()
305     {
306         SmartAccountPluginClient::new(&env, &plugin)
307             .on_auth(&env.current_contract_address(), &auth_contexts);
308     }
309
310     Ok(())
311 }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

It is recommended to invoke plugins via `try_on_auth` and **never revert** on plugin errors; instead, **emit a structured event** for telemetry and continue:

- Call `try_on_auth(&account_addr, &auth_contexts)`.
- On error, publish e.g. `PluginAuthFailed { plugin, error, contexts }` and proceed without reverting.

This simple, robust pattern aligns with the trusted-plugins model: it **preserves security semantics** (authorization is still decided by `Authorizer::check`) while improving **availability** and **operational visibility**.

Remediation Comment

SOLVED: The **Crossmint team** resolved this issue by implementing the recommended fixes.

Remediation Hash

https://github.com/Crossmint/stellar-smart-account/pull/69

7.4 MISSING WEBAUTHN CHALLENGE BINDING AND PAYLOAD USAGE ON SECP256R1 VERIFICATION

// MEDIUM

Description

The current Secp256r1 `verify` function verifies signatures over `sha256(authenticator_data || sha256(client_data_json))` but ignores the `payload` parameter and does not check that `clientDataJSON.challenge` matches the base64url-encoded signature `payload` built by the host.

This lack of explicit binding presents a potential risk that a valid WebAuthn assertion from a different context could be accepted for the current transaction, weakening the intended integrity between the host-provided payload and the assertion.

Code Location

Code of `verify` function from **contracts/smart-account/src/auth/signers/secp256r1.rs** file:

```
20  impl SignatureVerifier for Secp256r1Signer {
21      fn verify(&self, env: &Env, _payload: &BytesN<32>, proof: &SignerProof) -> Result<(), Error>
22          match proof {
23              SignerProof::Secp256r1(signature) => {
24                  let Secp256r1Signature {
25                      mut authenticator_data,
26                      client_data_json,
27                      signature,
28                  } = signature.clone();
29
30                  authenticator_data
31                      .extend_from_array(&env.crypto().sha256(&client_data_json).to_array());
32
33                  // This will panic if the signature is invalid
34                  env.crypto().secp256r1_verify(
35                      &self.public_key,
36                      &env.crypto().sha256(&authenticator_data),
37                      &signature,
38                  );
39                  // Reaching this point means the signature is valid
40                  Ok(())
41              }
42              _ => Err(Error::InvalidProofType),
43          }
44      }
45  }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

It is recommended align with Stellar's official example for CustomAccountInterface, which constructs `expected_challenge = base64url(signature_payload)` and validates `clientDataJSON.challenge == expected_challenge` before accepting the assertion.

See: **CustomAccountInterface implementation tutorial.**

Remediation Comment

SOLVED: The **Crossmint team** has solved this issue by implementing the recommended fixes.

Remediation Hash

https://github.com/Crossmint/stellar-smart-account/pull/75

7.5 MISSING POLICY ON_REVOKE ON SIGNER REVOCATION

// LOW

Description

The `revoke_signer` function deletes a Standard signer without invoking `on_revoke` for the signer's associated policies. Although removed policies will no longer be evaluated during authorization, the external policy contracts may retain stale state (e.g., references, counters, residual deny entries), causing storage bloat and operational inconsistencies.

Code Location

Code of `revoke_signer` function from `contracts/smart_account/src/account.rs` file:

```rust
fn revoke_signer(env: &Env, signer_key: SignerKey) -> Result<(), Error> {
    Self::require_auth_if_initialized(env);

    let storage = Storage::persistent();

    let signer_to_revoke = storage
        .get::<SignerKey, Signer>(env, &signer_key)
        .ok_or(Error::SignerNotFound)?;

    if signer_to_revoke.role() == SignerRole::Admin {
        return Err(Error::CannotRevokeAdminSigner);
    }
    storage.delete::<SignerKey>(env, &signer_key)?;
    env.events().publish(
        (TOPIC_SIGNER, VERB_REVOKED),
        SignerRevokedEvent::from(signer_to_revoke),
    );
    Ok(())
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

It is recommended to iterate the policies attached to the signer being revoked and invoke `on_revoke` on each policy **before** deleting the signer from storage. Use “try” variants (non-reverting) and emit telemetry events on failures, ensuring revocation cannot be blocked by a misbehaving policy while still providing an audit trail.

Remediation Comment

SOLVED: The **Crossmint team** solved this issue by implementing the recommended fixes.

Remediation Hash

https://github.com/Crossmint/stellar-smart-account/commit/a3525f9ef57a8c10d6866435eb403d918e5f6f0b

7.6 POLICY CALLBACKS NOT INVOKED ON SIGNER UPDATE

// LOW

Description

The `update_signer` function updates a signer's role and admin counter, then persists the new Signer, but it does not invoke policy lifecycle callbacks aligned with role transitions. Specifically:

- **Admin → Standard:** policies become active, but `on_add` is not called.
- **Standard → Admin:** policies are deactivated, but `on_revoke` is not called.

If `update_signer` ever mutates the policy set itself, it also does not call `on_add` for newly added policies nor `on_revoke` for removed ones. This can leave external policy contracts with stale or missing state (e.g., quotas, locks, counters, allow/deny lists), breaking inter-contract invariants even though the core authorizer will stop consulting removed policies.

Code Location

Code of `update_signer` function from **contracts/smart-account/src/account.rs** file:

```
118 fn update_signer(env: &Env, signer: Signer) -> Result<(), Error> {
119     Self::require_auth_if_initialized(env);
120     let key = signer.clone().into();
121     let storage = Storage::persistent();
122     let old_signer = storage
123         .get::<SignerKey, Signer>(env, &key)
124         .ok_or(Error::SignerNotFound)?;
125     if old_signer.role() == SignerRole::Admin && signer.role() != SignerRole::Admin {
126         let count = storage
127             .get::<Symbol, u32>(env, &ADMIN_COUNT_KEY)
128             .unwrap_or(0);
129         if count <= 1 {
130             return Err(Error::CannotDowngradeLastAdmin);
131         }
132         storage.store::<Symbol, u32>(env, &ADMIN_COUNT_KEY, &(count - 1))?;
133     } else if old_signer.role() != SignerRole::Admin && signer.role() == SignerRole::Admin {
134         let count = storage
135             .get::<Symbol, u32>(env, &ADMIN_COUNT_KEY)
136             .unwrap_or(0);
137         storage.store::<Symbol, u32>(env, &ADMIN_COUNT_KEY, &(count + 1))?;
138     }
139     storage.update::<SignerKey, Signer>(env, &key, &signer)?;
140     env.events().publish(
141         (TOPIC_SIGNER, VERB_UPDATED),
142         SignerUpdatedEvent::from(signer),
143     );
144
145     Ok(())
146 }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

It is recommended to detect the role transition inside `update_signer` and invoke the corresponding callbacks **before** persisting the new signer:

- On **Admin → Standard**, call `on_add` for the signer's policy set.
- On **Standard → Admin**, call `on_revoke` for the signer's policy set.

Remediation Comment

SOLVED: The **Crossmint team** has resolved the issue by refactoring the code, resulting in improved readability.

Remediation Hash

https://github.com/Crossmint/stellar-smart-account/pull/73

7.7 SILENT SWALLOWING OF EXTERNAL POLICY CALLBACK FAILURES

// LOW

Description

The external policy lifecycle callbacks are invoked via non-reverting `try_*` methods, but their `Result` is ignored.

When `try_on_add` or `try_on_revoke` fails, the error is discarded and the function returns `Ok(())` unconditionally. This design provides no on-chain signal (event/flag/counter) for operators and integrators, reducing observability and potentially leaving external policy contracts in inconsistent state (e.g., unreleased locks, stale counters, dangling deny-list entries) without any audit trail. While keeping callbacks non-blocking is desirable to avoid DoS, silently discarding failures hinders incident response and post-mortem debugging.

Code Location

```
25  impl PolicyCallback for ExternalPolicy {
26      fn on_add(&self, env: &Env) -> Result<(), Error> {
27          let policy_client = SmartAccountPolicyClient::new(env, &self.policy_address);
28          let _ = policy_client.try_on_add(&env.current_contract_address());
29          Ok(())
30      }
31
32      fn on_revoke(&self, env: &Env) -> Result<(), Error> {
33          let policy_client = SmartAccountPolicyClient::new(env, &self.policy_address);
34          let _ = policy_client.try_on_revoke(&env.current_contract_address());
35          Ok(())
36      }
37  }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

It is recommended to handle the `try_*` results explicitly to preserve non-blocking behavior **with observability**:

- Emit a structured event (e.g., `PolicyCallbackFailed`) including `{policy_address, verb("on_add"/"on_revoke"), signer_id/role, error_code}` whenever a callback fails.
- Optionally maintain per-policy failure counters and expose a view (`get_policy_health`) for monitoring; consider a configurable quarantine after *N* consecutive failures, emitting `PolicyQuarantined`.
- Keep the main operation non-reverting (to avoid DoS), but make failures visible and queryable.

Remediation Comment

SOLVED: The **Crossmint team** resolved this issue by implementing the recommended fixes.

Client Note:
This PR was for Find-009, but it also implements those mentioned events!

Remediation Hash

https://github.com/Crossmint/stellar-smart-account/pull/69/files

7.8 MISSING EVENT EMISSIONS FOR INITIALIZATION AND UPGRADES

// INFORMATIONAL

Description

The functions responsible for `initialization` and `code upgrades` do not emit any events.

This prevents reliable, on-chain observability and audit trails for two high-impact lifecycle actions: when a contract is initialized and when its Wasm code is upgraded. Without these signals, external monitors and compliance tooling cannot deterministically attribute who/when/what occurred, complicating incident response and forensics.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to emit structured events for both actions:

- On `initialization`: an `Initialized` event containing source (contract address), initiator (if applicable), and timestamp.
- On `upgrade`: an `UpgradePerformed` event containing initiator, old/new Wasm identifiers (hashes), and timestamp, avoiding sensitive data. Emit immediately after `update_current_contract_wasm` (and also at completion if using a two-step migrate flow).

Remediation Comment

SOLVED: The **Crossmint team** successfully addressed this issue by adding the missing events.

Remediation Hash

https://github.com/Crossmint/stellar-smart-account/commit/a535753c99780e0b94df3e710eddbcc8697b972e

---