MO Issuance V2 - SVM - Managed Smart Contract Assessment

MO

HALBORN

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ABOUT THIS REPORT

ASSESSMENT TYPE
Assurance Assessment

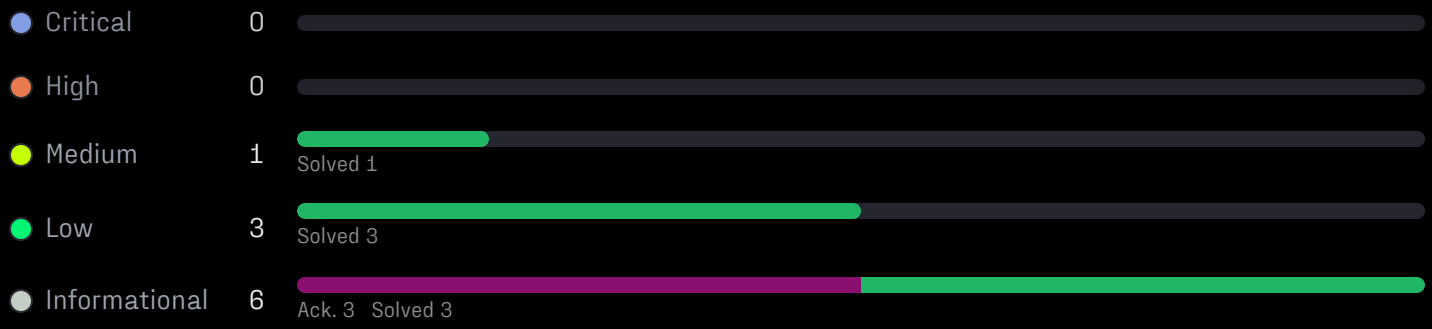
DATE OF ENGAGEMENT
Oct 1, 2026 - Oct 14, 2026

SOURCE CODE
solana-m-extensions
/issuance-v2-3ac60b772e4985141bc627e7f11b034

ASSESSED COMMIT ID
https://github.com/m0-platform/solana-m-extensions/pull/73

SEVERITY BREAKDOWN

10	7	3	0
All Findings	Solved	Acknowledged	Risk Accepted



Introduction

The `M0 team` engaged `Halborn` to conduct a security assessment of their **issuance-v2 Solana program** beginning on October 2 2026, and ending on October 14 2026. The security assessment was scoped to the Solana Programs provided in issuance-v2 GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The `M0 Issuer Gateway` is the Solana implementation of `M0 v2`'s stablecoin issuance infrastructure: an upgradeable program deployed once per issuer and operated through a single admin key, with no `M0` authority inside the issuer's deployment. It gives each issuer a self-contained platform to launch, govern, and operate a Token-2022 stablecoin, separating token logic, policy configuration, and compliance enforcement into layers that can evolve independently while the issuer retains ultimate control through its upgrade authority.

Assessment Summary

`Halborn` was provided 9 days for the engagement and assigned one full-time security engineer to review the security of the Solana Programs in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.
The purpose of the assessment is to:

- Identify potential security issues within the Solana Programs.
- Ensure that smart contract functionality operates as intended.

In summary, `Halborn` identified some improvements to reduce the likelihood and impact of risks, most of which were addressed by the `M0 team`. The main ones were the following:

- `Reconcile principal_of_supply against the actual mint.supply on every rewards-reading path to avoid yield drift on adopted mints lacking the PermissionedBurn extension.`
- `Ensure adopted mints are not paused.`
- `Ensure that the IssuerConfig admin proves private key control.`
- `Pin the gateway program ID at compile time.`

Test Approach and Methodology

Halborn performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.
The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Scanning dependencies for known vulnerabilities (`cargo audit`).
- Local runtime testing (`litesvm`)

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)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	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)	0
	Low (C:L)	0.25
	Medium (C:M)	0.5
	High (C:H)	0.75
	Critical (C:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical (A:C)	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium (Y:M)	0.5
	High (Y:H)	0.75

IMPACT METRIC (m_I)	METRIC VALUE	NUMERICAL VALUE
	Critical (Y:C)	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)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	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.

Critical	High	Medium	Low	Informational

9 - 10	7 - 8.9	4.5 - 6.9	2 - 4.4	0 - 1.9

Scope

REPOSITORY

(a) Repository: solana-m-extensions

(b) Assessed Commit ID: https://github.com/m0-platform/solana-m-extensions/pull/73

(c) Items in scope:

- programs/m_ext/src 4 files
  - instructions 2 files
    - handoff.rs Rust
    - mod.rs Rust
  - errors.rs Rust
  - lib.rs Rust

FILE

(a) Submitted File: /issuance-v2-3ac60b772e4985141bc627e7f11b034fc94d190e/svm/crates/state/src/roles/mod.rs

(b) Items in scope:

- issuance-v2-3ac60b772e4985141bc627e7f11b034fc94d190e/svm 71 files
  - crates/state/src 11 files
    - roles 3 files
      - mod.rs Rust
      - role_grant.rs Rust
      - role.rs Rust
    - constants.rs Rust
    - issuer_config.rs Rust
    - lib.rs Rust
    - mint_proposal.rs Rust
    - stablecoin.rs Rust
    - thaw_policy.rs Rust
    - token_acl.rs Rust
    - transmute.rs Rust
  - programs/issuer-gateway/src 60 files
    - actions 47 files
      - admin 6 files
        - accept_admin.rs Rust
        - initialize_issuer_config.rs Rust

- mod.rs Rust
- pause_gateway.rs Rust
- propose_admin.rs Rust
- resume_gateway.rs Rust

compliance 6 files
- force_transfer.rs Rust
- freeze_account.rs Rust
- mod.rs Rust
- pause.rs Rust
- resume.rs Rust
- thaw_account.rs Rust

confidential 3 files
- approve_confidential_account.rs Rust
- mod.rs Rust
- update_confidential_transfer_mint.rs Rust

issuance 7 files
- burn.rs Rust
- cancel_mint.rs Rust
- execute_mint.rs Rust
- mod.rs Rust
- propose_mint.rs Rust
- reset_breaker.rs Rust
- trip_breaker.rs Rust

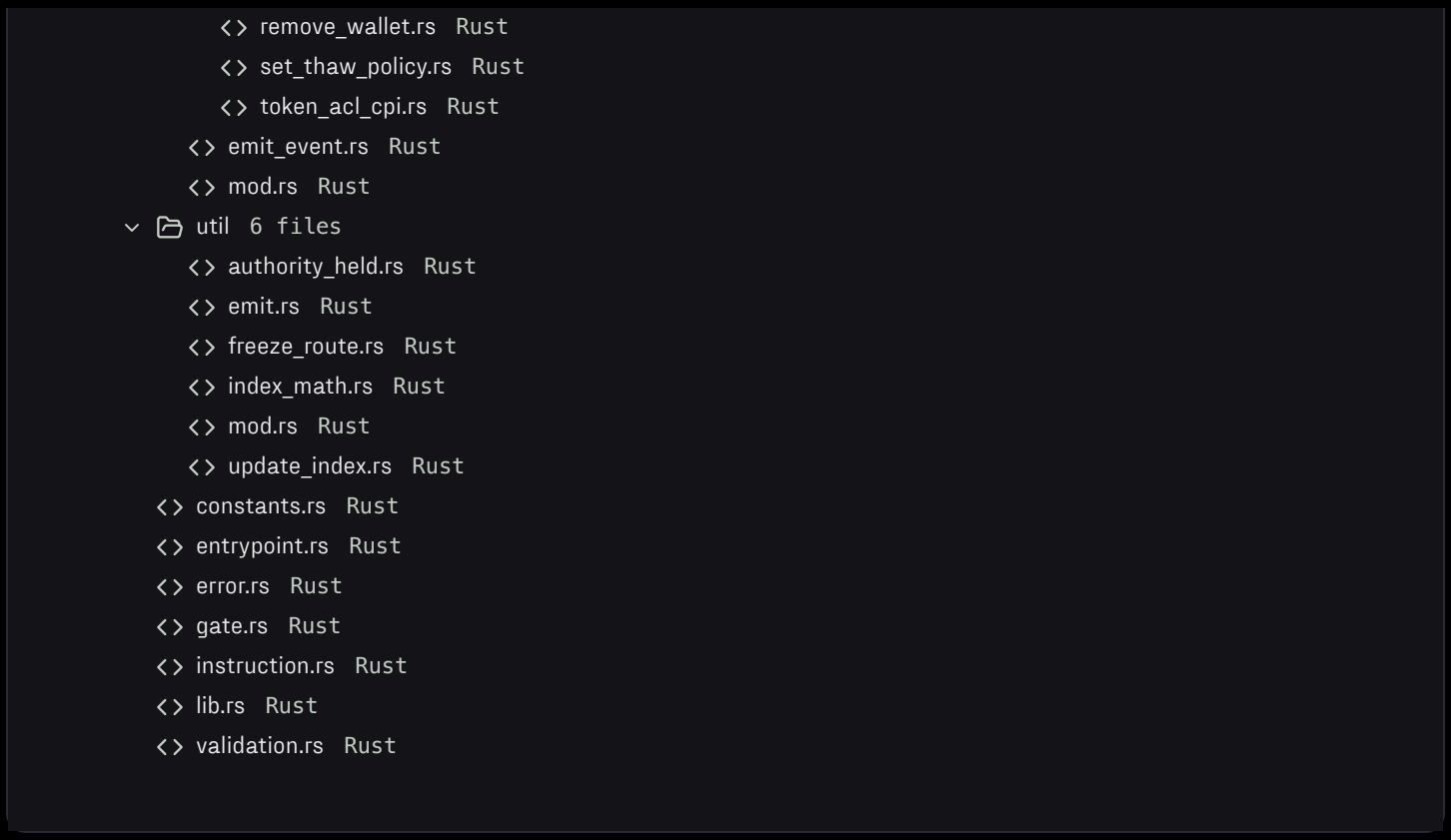
lifecycle 7 files
- abort_adoption.rs Rust
- adjust_stablecoin.rs Rust
- adopt_stablecoin.rs Rust
- complete_adoption.rs Rust
- eject_authority.rs Rust
- mod.rs Rust
- register_stablecoin.rs Rust

metadata 3 files
- mod.rs Rust
- update_token_metadata.rs Rust
- write_token_metadata.rs Rust

rewards 4 files
- claim_accrued.rs Rust
- distribute_rewards.rs Rust
- mod.rs Rust
- set_rewards_recipient.rs Rust

roles 4 files
- close_role_grant.rs Rust
- grant_roles.rs Rust
- mod.rs Rust
- revoke_roles.rs Rust

thaw_gating 5 files
- add_wallet.rs Rust
- mod.rs Rust



REMEDIATION COMMIT ID:

- 1ac1e712152481c66bc39daa0dd34ebb3a4252ad
- 1e6a0e0ddad8ec7f06de85874eb90693227e13d3
- 06198123bb6699d1109c0b923a86826b4b569852
- 2e19f9ba03b0f0c7ce33a131af4f8dd5d1fc264b
- a76bc5f548590f051e5f0e2145366dc283d191b7
- e76de30b4e21e0de8707fe0cd532a59a31cdaec5
- 97f70298edbf144890e64c92ccfb8bd57b5cedf9
- 4856710e9008f2505ba538d162c60d94032971f0

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

Assessment Summary & Findings Overview

#	TITLE	SEVERITY	SCORE	STATUS
HAL-001	Native burns on adopted mints lacking	Medium	5.0	SOLVED 10/23/2026

#	TITLE	SEVERITY	SCORE	STATUS
	PermissionedBurn silently inflate yield payouts			
HAL-002	Permanent transfer lock due to missing Pausable floor check at adoption	Low	2.4	SOLVED 10/23/2026
HAL-003	Bootstrap can brick the deployment due to unvalidated admin key	Low	2.1	SOLVED 10/23/2026
HAL-004	Mint authority can be redirected to an unverified gateway due to unpinned program ID	Low	2.0	SOLVED 10/27/2026
HAL-005	Stale role grants survive admin rotation	Informational	1.4	ACKNOWLEDGED 10/23/2026
HAL-006	Proposal payer loses rent due to signer-picked rent_recipient	Informational	0.6	ACKNOWLEDGED 10/23/2026
HAL-007	Permanent metadata brick due to missing UTF-8 validation at registration	Informational	0.6	SOLVED 10/23/2026
HAL-008	ConfigManager can halt issuance due to missing upper bounds on mint-config knobs	Informational	0.3	SOLVED 10/23/2026
HAL-009	Admin-writable config fields accept zero or malformed values without guards	Informational	0.2	SOLVED 10/23/2026
HAL-010	Cross-proposer griefing due to missing proposer-ownership check	Informational	0.2	ACKNOWLEDGED 10/23/2026

Findings & Tech Details

HAL-001 SOLVED

Native burns on adopted mints lacking PermissionedBurn silently inflate yield payouts

Medium · 5.0 AO:A/AC:L/AX:L/R:P/S:U/C:N/A:L/I:H/D:N/Y:H

DESCRIPTION

The M0 Issuer Gateway's rewards model tracks `principal_of_supply` as the aggregate principal units behind the stablecoin's circulating supply and relies on the invariant `principal_of_supply × latest_index / INDEX_SCALE == mint.supply + unclaimed_yield` across every state change. The invariant is kept by routing every mint through `execute_mint` / `distribute_rewards` and every burn through the gateway's own `burn` handler, with Token-2022's PermissionedBurn extension rejecting any native burn at the mint level so no other path can retire supply.

However, the adoption floor at `check_extensions()` accepts mints lacking the PermissionedBurn extension, and Token-2022's TLV-sealing rule prevents adding the extension to a mint whose `InitializeMint2` has already run. Legacy mints migrating into the gateway (notably v1 m_ext extension mints such as wM, which pre-date Token-2022's PermissionedBurn and cannot acquire it retroactively) therefore clear adoption while two independent accounting failures take hold.

The gateway's own `burn` handler CPIs Token-2022's `PermissionedBurn::Burn`, which refuses mints without the extension. For an adopted mint lacking PermissionedBurn, this CPI reverts and the whole transaction rolls back (including the pre-CPI `principal_of_supply` decrement), so the gateway's sanctioned supply-retirement path is unreachable.

svm/programs/issuer-gateway/src/actions/issuance/burn.rs

RUST 139–147

```rust
    PermissionedBurn::<AccountView>::new(
        ctx.token_program.address(),
        ctx.source,
        ctx.mint,
        ctx.stablecoin, // the Permissioned Burn authority
        ctx.source_owner,
        amount,
    )
    .invoke_signed(&[Signer::from(&seeds)])?;
```

Native Token-2022 `Burn` succeeds freely on these mints because no PermissionedBurn gate intercepts it. `mint.supply` decreases on each native burn, but `principal_of_supply` is only

decremented inside the gateway's own `burn` handler and nowhere else, so the two counters drift apart. On the next `claim_accrued`, `rewards_owed` computes `principal_to_amount_down(principal_of_supply, latest_index) - supply` and the gap is minted to `rewards_recipient` with no bucket consumed (`claim_accrued` uses `update_index`, not `update_index_and_consume`).

svm/programs/issuer-gateway/src/actions/rewards/claim_accrued.rs

RUST 117–128

```rust
        // Settle, then mint the gap between present value and supply. The pricing
        // read must follow the settle's own borrow.
        let latest_index = update_index(ctx.stablecoin)?;
        let supply = { Mint::from_account_view(ctx.mint)?.supply() };
        let amount = {
            let data = ctx.stablecoin.try_borrow()?;
            rewards_owed(
                Stablecoin::load_checked(&data)?.principal_of_supply,
                latest_index,
                supply,
            )?
        };
```

Every native burn of X base units on an adopted PermissionedBurn-less mint therefore results in an equivalent X-unit `MintTo` to `rewards_recipient` on the next `claim_accrued`, with no on-chain detection and no rate-limit cap. Holders who burn their own tokens (through buggy clients, bridge accidents, failed-tx leftovers, or user error) silently transfer the burned value to the yield recipient; the gateway's documented invariant ("all burns route through the gateway") is false for adopted mints. For v1 m_ext migrations the gateway's `burn` handler is additionally unreachable, so no sanctioned retirement path exists at all and native burn with its accounting drift becomes the only exit from the token.

PROOF OF CONCEPT

Given an adopted mint without the `PermissionedBurn` extension:

- the gateway `burn` instruction fails because the `PermissionedBurn` CPI reverts
- a native Token-2022 `Burn` signed by the owner succeeds and reduces the mint supply
- a subsequent `claim_accrued` mints the natively-burned amount to the rewards_recipient

RUST

```rust
mod common;

use common::asserts::TransactionResultExt;
use common::{set_authority_ix, TestContext};
use solana_instruction::{AccountMeta, Instruction};
```

```rust
use solana_keypair::Keypair;
use solana_pubkey::Pubkey;
use solana_signer::Signer;
use spl_token_2022_interface::extension::ExtensionType;

const AUTHORITY_MINT_TOKENS: u8 = 0;
const AUTHORITY_FREEZE: u8 = 1;

/// Create a legacy-shaped mint (no PermissionedBurn extension), adopt it, move
/// its mint and freeze authorities to the Stablecoin PDA, complete adoption,
/// and mint `supply` units to a fresh holder account. The adopted stablecoin
/// stores `rewards_recipient` on its PDA. Returns
/// `(admin, mint, holder, token_account)`.
fn adopted_mint_without_permissioned_burn(
    ctx: &mut TestContext,
    supply: u64,
    rewards_recipient: Pubkey,
) -> (Keypair, Pubkey, Keypair, Pubkey) {
    let admin = ctx.bootstrap();
    let (mint, authority) = ctx.create_external_mint(6);
    let mint_key = mint.pubkey();

    // Adopt. The floor at `check_extensions` accepts a mint missing
    // PermissionedBurn: `get_extension::<PermissionedBurnConfig>()` returns Err
    // and the `is_ok_and` short-circuits before any field check runs.
    let params = sdk::AdoptStablecoinParams {
        rewards_recipient,
        ..common::adopt_params()
    };
    let ix = sdk::adopt_stablecoin(&ctx.payer.pubkey(), &admin.pubkey(), &mint_key, &params);
    ctx.send(&[ix], &[&admin]).assert_ok();

    // Hand off mint and freeze authorities to the Stablecoin PDA. This
    // simulates the V1 handoff step of the migration cutover.
    let (stablecoin_pda, _) = sdk::find_stablecoin_pda(&mint_key);
    for authority_type in [AUTHORITY_MINT_TOKENS, AUTHORITY_FREEZE] {
        let ix = set_authority_ix(
            &mint_key,
            authority_type,
            &authority.pubkey(),
            &stablecoin_pda,
        );
        ctx.send(&[ix], &[&authority]).assert_ok();
    }

    // Complete adoption. The completion floor at `check_no_retained_authorities`
    // also accepts the missing extension because `require_pda(None, ...)` returns Ok.
    let ix = sdk::complete_adoption(&admin.pubkey(), &mint_key);
    ctx.send(&[ix], &[&admin]).assert_ok();

    // Mint supply via the standard gateway flow. The Stablecoin PDA now holds
    // the mint authority, and the propose/execute path works end to end.
    let holder = Keypair::new();
    ctx.svm.airdrop(&holder.pubkey(), 1_000_000_000).unwrap();
    let token_account = ctx
        .create_token_account(&mint_key, &holder.pubkey())
        .pubkey();
    ctx.mint_tokens(&admin, &mint_key, &token_account, supply);

    (admin, mint_key, holder, token_account)
}

#[test]
fn the_gateway_burn_fails_on_a_mint_without_permissioned_burn() {
    let mut ctx = TestContext::new();
    let (admin, mint, holder, token_account) =
```

```
        adopted_mint_without_permissioned_burn(&mut ctx, 1_000, Pubkey::default());

    // Confirm the extension was never added by the adoption flow.
    assert!(
        !ctx.mint_extension_types(&mint)
            .contains(&ExtensionType::PermissionedBurn),
        "the adoption flow does not add PermissionedBurn to a mint that did not have it, \
         and Token-2022's TLV-sealing rule forbids adding it after InitializeMint2"
    );

    // The gateway's `burn` handler CPIs Token-2022's PermissionedBurn::Burn,
    // which reverts on a mint lacking the extension. The surrounding
    // transaction rolls back, including the pre-CPI `principal_of_supply`
    // decrement, so nothing persists.
    let ix = sdk::burn(
        &admin.pubkey(),
        &mint,
        &token_account,
        &holder.pubkey(),
        400,
    );
    ctx.send(&[ix], &[&admin, &holder]).assert_err_raw();

    assert_eq!(
        ctx.token_balance(&token_account),
        1_000,
        "the holder's balance is unchanged because the transaction reverted"
    );
    assert_eq!(ctx.supply(&mint), 1_000, "the mint supply is unchanged");
}

#[test]
fn a_native_burn_inflates_claim_accrued_by_the_burned_amount() {
    const INITIAL_SUPPLY: u64 = 1_000;
    const BURN_AMOUNT: u64 = 400;

    let mut ctx = TestContext::new();
    let partner = Keypair::new();
    let (admin, mint, holder, token_account) =
        adopted_mint_without_permissioned_burn(&mut ctx, INITIAL_SUPPLY, partner.pubkey());

    // Startup state: no streaming rate, no native burns yet, so no drift and
    // no legitimate yield. `rewards_owed` is zero.
    assert_eq!(ctx.rewards_owed(&mint), 0, "no drift at startup");

    // The holder natively burns `BURN_AMOUNT` tokens. With no PermissionedBurn
    // extension there is no gate to stop them.
    let data = [&[8u8][..], &BURN_AMOUNT.to_le_bytes()].concat();
    let native_burn = Instruction {
        program_id: sdk::TOKEN_2022_PROGRAM_ID,
        accounts: vec![
            AccountMeta::new(token_account, false),
            AccountMeta::new(mint, false),
            AccountMeta::new_readonly(holder.pubkey(), true),
        ],
        data,
    };
    ctx.send(&[native_burn], &[&holder]).assert_ok();
    assert_eq!(ctx.supply(&mint), INITIAL_SUPPLY - BURN_AMOUNT);

    // The gateway's `principal_of_supply` did not update, so `rewards_owed`
    // now reports the burned amount as "unclaimed yield" even though there
    // was no streaming rate at any point.
    let (_latest_index, principal_of_supply) = ctx.index_state(&mint);
    assert_eq!(
        principal_of_supply, INITIAL_SUPPLY,
```

```
        "principal_of_supply is unchanged because the native burn bypassed the gateway"
    );
    assert_eq!(
        ctx.rewards_owed(&mint),
        BURN_AMOUNT,
        "the drift surfaces as phantom yield equal to the burned amount"
    );

    // `claim_accrued` mints that phantom yield straight to the rewards_recipient.
    let partner_account = ctx.create_token_account(&mint, &partner.pubkey()).pubkey();
    let ix = sdk::claim_accrued(&admin.pubkey(), &mint, &partner_account);
    ctx.send(&[ix], &[&admin]).assert_ok();

    // The rewards_recipient receives the natively-burned value as newly-minted
    // supply, so the holder's loss has been silently redirected to them.
    assert_eq!(
        ctx.token_balance(&partner_account),
        BURN_AMOUNT,
        "the rewards_recipient receives the natively-burned value as fresh mint"
    );
    assert_eq!(
        ctx.supply(&mint),
        INITIAL_SUPPLY,
        "mint supply back to the pre-burn level: the burned supply reappeared on the
rewards_recipient"
    );
```

```
running 2 tests
test the_gateway_burn_fails_on_a_mint_without_permissioned_burn ... ok
test a_native_burn_inflates_claim_accrued_by_the_burned_amount ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.11s
```

RECOMMENDATION

To address this finding, it is recommended to reconcile `principal_of_supply` against the actual `mint.supply` on every rewards-reading path rather than tightening the adoption floor (requiring PermissionedBurn at adoption would eliminate the issue at its source but is unavailable for v1 m_ext migrations whose mints structurally cannot acquire the extension). Specifically, in `claim_accrued` after `update_index` but before computing `rewards_owed`, read `mint.supply` and derive the implied principal `amount_to_principal_up(mint.supply, latest_index)`; whenever `sc.principal_of_supply` exceeds this derived value, reduce it to match before computing the payout.

REMEDIATION COMMENT

Addressed in: `1ac1e712152481c66bc39daa0dd34ebb3a4252ad`

SOLVED: The `M0` team resolved this finding by reconciling the `principal_of_supply` on `claim_accrued`, `distribute_rewards`, `execute_mint`, `burn` paths so out-of-band native burns debit `principal_of_supply` instead of being minted as yield.

HAL-002 SOLVED

Permanent transfer lock due to missing Pausable floor check at adoption

Low · 2.4 AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:M/D:N/Y:L

DESCRIPTION

The `AdoptStablecoin` instruction brings an existing Token-2022 mint under gateway control, and its `check_extensions()` helper is the adopt-time floor check that reads the mint's extensions and refuses any state that `complete_adoption` would later be unable to repair.

However, `check_extensions()` never inspects the Pausable extension. A mint whose `PausableConfig.authority` is `None` and whose `paused` flag is already `true` therefore clears adoption. Completion then accepts the same state because `check_no_retained_authorities()` groups the Pausable authority into its `pda_or_vacant` bucket (which accepts `None`), and `plan_resets()` carries no reset path for `PausableConfig.paused`, so the `paused=true` flag survives unchanged into the Adopted lifecycle. The absence of a Pausable branch in `check_extensions()` sits between the refusals for `PermissionedBurn` and `TokenMetadata` and the function's `Ok(present)` return, with no intervening read of the Pausable extension's authority or `paused` field.

svm/programs/issuer-gateway/src/actions/lifecycle/adopt_stablecoin.rs

RUST 119–131

```rust
    // A cleared burn authority reopens holder self-burns for good.
    if mint
        .get_extension::<PermissionedBurnConfig>()
        .is_ok_and(|burn| burn.authority.as_ref().is_none())
    {
        return Err(GatewayError::BurnAuthorityRevoked.into());
    }
    // Renounced metadata can never be repaired.
    let metadata_authority = token_metadata_update_authority(mint)?;
    if metadata_authority.is_some_and(|a| a.as_ref().is_none()) {
        return Err(GatewayError::MetadataAuthorityRevoked.into());
    }
    Ok(present)
```

On the completion side, `check_no_retained_authorities()` buckets the Pausable authority alongside ScaledUi and the other extensions whose authority is allowed to be either the stablecoin PDA or `None`, and the subsequent loop refuses only non-PDA keys, never `None`.

svm/programs/issuer-gateway/src/actions/lifecycle/complete_adoption.rs

RUST 88–99

```rust
        mint.get_extension::<ScaledUiAmountConfig>()
            .ok()
            .map(|e| &e.authority),
        mint.get_extension::<PausableConfig>()
            .ok()
            .map(|e| &e.authority),
    ];
    for slot in pda_or_vacant.into_iter().flatten() {
        if slot.as_ref().is_some_and(|k| k != stablecoin) {
            return Err(GatewayError::AuthorityRetained.into());
        }
    }
```

Once `complete_adoption` runs, the mint lives in the Adopted lifecycle with `paused=true` and no principal able to toggle the flag: Token-2022 cannot reinstate a renounced authority, the gateway's pause and resume paths route through the Stablecoin PDA which never held the Pausable authority in the first place, and `adopt_stablecoin` cannot be re-run against a mint already Adopted. `transfer` and `transfer_checked` fail for every holder for the lifetime of the mint. The precondition is operator error during the pending window: the outgoing authority key must set `paused=true` and renounce the Pausable authority before `complete_adoption` runs, which is the same key the migration doc requires to move authorities to the PDA rather than renounce them. The `ScaledUi` authority sits in the same `pda_or_vacant` bucket and loses UI-multiplier adjustments on renounce; the impact there is capability loss rather than a transfer lock, but the fix shape is the same.

RECOMMENDATION

To address this finding, it is recommended to tighten the adoption floor so no state passes `check_extensions()` that `complete_adoption` cannot later repair. Specifically, refuse `PausableConfig.paused=true` in `check_extensions()`, since completion carries no reset path for the flag, and promote `PausableConfig.authority` from the `pda_or_vacant` bucket to a `require_pda` check in `check_no_retained_authorities()`, since the gateway advertises pause as a core compliance control. Apply the same promotion to `ScaledUiAmountConfig.authority` if M0 treats UI-multiplier adjustments as a baseline capability; otherwise, document the capability-loss case explicitly in the `StablecoinAdopted` event by recording which authority slots were waived on renunciation.

REMEDIATION COMMENT

Addressed in: 1e6a0e0ddad8ec7f06de85874eb90693227e13d3

SOLVED: The `M0` team resolved this finding by refusing renounced `Pausable` and `ScaledUI` authorities and adding a `Resume` CPI in `complete_adoption` so a pending-window pause is cleared before the mint enters the Adopted lifecycle

HAL-003 SOLVED

Bootstrap can brick the deployment due to unvalidated admin key

Low · 2.1 AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:L/D:N/Y:N

DESCRIPTION

The `initialize_issuer_config` instruction is the one-shot bootstrap that creates the `IssuerConfig` singleton PDA and records the admin key every later handler checks the signer against. Admin rotation afterward is a two-step `propose_admin` / `accept_admin` sequence where the current admin nominates and the nominee's signature finalizes the transfer.

However, `process_initialize_issuer_config` writes the admin account's address into the `IssuerConfig` without verifying that the admin signed, that it is non-zero, that it is not a program-owned PDA, and that it is not the system program id. The handler docstring states the admin is "deliberately unvalidated and not a signer", so the behavior is intentional, but it means an operator typo or a mis-prepared transaction permanently fixes an uncontrollable address in the root of trust.

In the handler body, the admin is read as `*ctx.admin.address()` and copied straight into the new `IssuerConfig` with no intermediate validation.

svm/programs/issuer-gateway/src/actions/admin/initialize_issuer_config.rs

RUST 82-104

```rust
    if ctx.payer.address() != &INITIALIZER {
        return Err(GatewayError::UnauthorizedInitializer.into());
    }

    let issuer_config = &*ctx.issuer_config;
    let admin = *ctx.admin.address();

    // Derive the canonical bump. `invoke_signed` alone cannot enforce a singleton.
    let bump = check_canonical_pda(issuer_config, &issuer_config_pda_seeds())?;
    let bump_seed = [bump];

    // Canonical `["event_authority"]` bump, cached so every later emit signs with
    // one hash instead of a search.
    let event_authority_bump =
        check_canonical_pda(ctx.event_authority, &event_authority_pda_seeds())?;

    let seeds = issuer_config_signer_seeds(&bump_seed);
    create_program_account(
        ctx.payer,
        ctx.issuer_config,
        &seeds,
        IssuerConfig::new(bump, &admin, event_authority_bump),
    )?;
```

Any value the INITIALIZER passes in the admin slot is final: a zero key, the `issuer_config` or `event_authority` PDA, the system program id, this program's own id, or a mistyped pubkey. Nothing holds a signature for any of those addresses, so every admin-gated instruction (stablecoin registration, role grants, pause, breaker reset, admin rotation) becomes unreachable. `propose_admin` cannot start a rotation because it requires the stored admin to sign, and `check_uninitialized` on the singleton PDA refuses a corrective re-run. The only on-chain recovery is a BPF program upgrade, which depends on a separate authority that may not act promptly.

RECOMMENDATION

To address this finding, it is recommended to prove key control over the recorded admin before it becomes irrevocable. The strongest version is a nominate-then-accept pattern at bootstrap: `initialize_issuer_config` writes the supplied address into `pending_admin` and leaves `admin` set to a sentinel (zero or the `INITIALIZER` itself), and a subsequent `accept_admin` by the nominee finalizes the role. This mirrors the rotation path and makes a typo recoverable without a program upgrade.

Alternatively, require the admin to be a signer on the init transaction by calling `check_signer(admin)` before the write, so the operator cannot bootstrap with a key they do not control.

Additionally, reject obviously-wrong values (`Address::default()`, the `issuer_config` and `event_authority` PDAs, the system program id, and this program's own id) in both the init and the rotation paths, so a typo that happens to land on a well-known constant also fails fast.

REMEDIATION COMMENT

Addressed in: 06198123bb6699d1109c0b923a86826b4b569852

SOLVED: The `M0` team resolved this finding by ensuring that the admin must sign the instruction.

HAL-004 SOLVED

Mint authority can be redirected to an unverified gateway due to unpinned program ID

Low · 2.0 AO:S/AC:L/AX:M/R:N/S:U/C:N/A:C/I:C/D:C/Y:N

DESCRIPTION

The `Handoff` instruction is the one-shot v1→v2 migration path in the m_ext program. It transfers the ext_mint's mint authority to a V2 Issuer Gateway Stablecoin PDA sitting in `AdoptionPending`, sweeps the vault's $M to an admin-chosen treasury account, closes the vault token account, and clears `wrap_authorities`.

However, the target gateway program is supplied as an admin-passed `gateway_program: Pubkey` argument to the handler, and nothing on-chain pins the expected MO Issuer Gateway program ID. Both the Stablecoin PDA derivation and the account-owner check are self-consistent with whatever argument the admin names, so a program that reproduces the PDA seed and the four pinned field offsets (`disc=2`, `version=1`, `mint_origin=1`, `mint=ext_mint`) passes every validation inside the handler.

The validator derives the expected PDA from the argument and compares the account's owner to the same argument, which makes the check tautological against a hostile program whose PDA matches its own derivation.

programs/m_ext/src/instructions/handoff.rs

RUST 126-134

```rust
126     // The target is derived, never accepted: the gateway's
127     // `["stablecoin", ext_mint]` PDA for the id passed in.
128     let (expected, _) = Pubkey::find_program_address(
129         &[STABLECOIN_SEED, self.ext_mint.key().as_ref()],
130         gateway_program,
131     );
132     if self.stablecoin.key() != expected || self.stablecoin.owner != gateway_program {
133         return err!(ExtError::InvalidAccount);
134     }
```

An attacker who deploys a program under the matching seed with a byte-layout stub meeting those four offsets can be named as `gateway_program` by an admin whose signing key is compromised at the moment of handoff, or who is socially engineered into trusting the wrong program ID. The irreversible instruction then transfers the ext_mint's mint authority to the attacker's PDA while the vault's full $M balance lands at the admin-named `treasury_m_token_account`. The attacker's PDA

can mint new ext_mint tokens at will, diluting every existing holder, and the vault's $M is gone from the extension's control. The m_ext program is deployed per extension token, so each extension has exactly one legitimate gateway target, which can be baked into the fork at build time rather than taken as a runtime argument.

RECOMMENDATION

To address this finding, it is recommended to pin the gateway program ID at compile time by adding a `pub const EXPECTED_GATEWAY_PROGRAM: Pubkey` to `programs/m_ext/src/lib.rs` alongside the existing `declare_id!`, document setting it per fork as part of the extension-creation runbook (the same step that updates the program ID), and drop the `gateway_program` parameter from `handoff()`. The handler then derives the Stablecoin PDA under the pinned constant, closing the window in which an incorrect or misconfigured program ID, whether due to a compromised admin or a configuration error, can redirect the mint authority during the irreversible handoff.

REMEDIATION COMMENT

Addressed in: `2e19f9ba03b0f0c7ce33a131af4f8dd5d1fc264b`

SOLVED: The `M0` team solved this finding in the specified commit by pinning the gateway program id at build time through `GATEWAY_PROGRAM_ID` and removing the `gateway_program` argument. `handoff` now requires the stablecoin account to be that program's PDA and owned by it. The commit is on `proto-1007-mext-handoff`, not on `main`.

HAL-005 ACKNOWLEDGED

Stale role grants survive admin rotation

Informational · 1.4 AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:L/D:M/Y:N

DESCRIPTION

The `AcceptAdmin` instruction on the MO Issuer Gateway completes the two-step admin rotation protocol, promoting the signer into `IssuerConfig.admin` once the sitting admin has nominated them via `ProposeAdmin`, and is framed by its module documentation as the recovery path when the admin key is compromised.

However, `process_accept_admin()` touches only the `admin` and `pending_admin` fields of `IssuerConfig` and leaves every `RoleGrant` PDA minted by the outgoing admin untouched. The gate helper `verify_grant()` keys on `(mint, authority)` alone, with no record of which admin issued a grant and no epoch field, so grants delegated by a compromised admin remain exercisable under the new admin until each one is enumerated and closed.

`AcceptAdmin`'s rewrite of `IssuerConfig` is a two-field update: the signer is moved into `admin` and the nomination slot is cleared, nothing else is written.
svm/programs/issuer-gateway/src/actions/admin/accept_admin.rs

RUST 66–71

```rust
    {
        let mut data = ctx.issuer_config.try_borrow_mut()?;
        let cfg = IssuerConfig::load_mut_checked(&mut data)?;
        cfg.admin = pending;
        cfg.pending_admin = zero;
    }
```

On the gate side, `verify_grant()` validates a presented `role_grant` against a PDA derived from `(mint, signer)` and checks only that the stored grant carries the requested role bit; its signature carries neither an issuing admin nor an epoch, so an authority that was granted a role under the previous admin passes the check identically to one granted under the new admin.

svm/programs/issuer-gateway/src/gate.rs

RUST 178–191

```rust
pub(crate) fn verify_grant(
    role_grant: &AccountView,
    mint: &Address,
    signer: &Address,
    role: Role,

```

```
    denied: GatewayError,
) -> Result<u64, ProgramError> {
    // ...
    check_pda(
        role_grant,
        &role_grant_pda_seeds_with_bump(mint, signer, &[grant_bump]),
    )?;
    // ...
}
```

When admin rotation is invoked as incident response for a compromised key, every delegated role the compromised admin minted before the gateway was paused remains active for its grantee once the new admin resumes the gateway. The new admin must enumerate every outstanding `RoleGrant` PDA, by replaying `RolesGranted` events or scanning for the `RoleGrant` discriminator across program-owned accounts, and call `close_role_grant` on each before `resume_gateway`; a missed grant keeps its holder's role live across the rotation. No new privilege boundary is crossed, since the compromised admin key already carried every direct power, but the module documentation's framing of rotation as "incident recovery" is incomplete without this grant-eviction step.

RECOMMENDATION

To address this finding, it is recommended to amend the `AcceptAdmin` module documentation and the operator runbook so admin rotation, when used for incident recovery, explicitly requires the incoming admin to enumerate every outstanding `RoleGrant` PDA and close each one with `close_role_grant` in the same transaction as `accept_admin`, before any subsequent `resume_gateway` call.

REMEDIATION COMMENT

Addressed in: a76bc5f548590f051e5f0e2145366dc283d191b7

ACKNOWLEDGED: The `M0` team documented the recovery for a leaked admin key and did not invalidate existing role grants. `accept_admin` still rewrites only the admin fields, and `verify_grant` still keys on the mint and the authority. The spec says to pause the gateway, rotate, close every unrecognized `RoleGrant`, and only then resume.

HAL-006 ACKNOWLEDGED

Proposal payer loses rent due to signer-picked rent_recipient

Informational · 0.6 AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:L/Y:N

DESCRIPTION

The `process_execute_mint` and `process_cancel_mint` handlers in the SVM Issuer Gateway close an outstanding mint proposal and credit its rent-exempt lamports to a `rent_recipient` account supplied by the caller. The caller is the issuer admin, a `Role::MintExecutor` holder (for execute), or a `Role::MintProposer` holder (for cancel).

However, neither handler constrains `rent_recipient` to any particular identity. The only validation on the account is a `check_writable` call. The `MintProposal` PDA records `proposer`, `destination`, and `destination_owner` but not the account that funded the proposal, so nothing at close time can relate `rent_recipient` back to the party whose lamports funded the proposal.

The context builder for `process_execute_mint` runs a flat sequence of per-account checks and stops at `check_writable(rent_recipient)`. The matching builder for `process_cancel_mint` repeats the pattern verbatim.

svm/programs/issuer-gateway/src/actions/issuance/execute_mint.rs

RUST 78–86

```rust
        check_signer(authority)?;
        check_program_owned(issuer_config)?;
        check_program_owned(stablecoin)?;
        check_writable(stablecoin)?;
        check_writable(mint)?;
        check_writable(destination)?;
        check_writable(proposal)?;
        check_writable(rent_recipient)?;
        check_token_2022_program(token_program)?;
```

Once execution reaches `close_program_account`, every lamport on the proposal account, which carries the rent-exempt reserve for a 187-byte PDA (about 0.002 SOL at current rent rates), transfers to that account.

svm/programs/issuer-gateway/src/actions/issuance/execute_mint.rs

RUST 201–202

```rust
    // Close the spent proposal instead of marking it, so it cannot be replayed.

```

```
        close_program_account(ctx.proposal, ctx.rent_recipient)?;
```

The party that funded the proposal, usually the proposer's own wallet in the CLI flow, loses the rent of every proposal an executor or canceller closes. The amount per event is small, but every executed or cancelled proposal repeats the loss. Because `rent_recipient` is unrestricted, an executor can quietly accumulate those lamports in a wallet of their choice across the lifetime of the stablecoin, with no on-chain evidence of the diversion beyond the recipient listed in the raw transaction.

RECOMMENDATION

To address this finding, it is recommended to record the payer in `MintProposal` at `process_propose_mint` time and require `rent_recipient` to match it on close in both `process_execute_mint` and `process_cancel_mint`. Alternatively, derive `rent_recipient` from the proposal itself, closing to `proposal.proposer` or to `destination_owner`, so no new field is needed. Preserve the current "any signer picks" behavior only in the role-revocation cleanup path documented in `process_cancel_mint`, where no funder exists to refund.

REMEDIATION COMMENT

ACKNOWLEDGED: The `M0` team acknowledged this finding, decided to keep the original implementation, and added the following note:

The exposure is about 0.002 SOL of rent per proposal, and only a `MintProposer` or the admin can choose `rent_recipient`. Fixing it means storing the payer, which changes the `MintProposal` account layout, and that cost outweighs the risk.

HAL-007 SOLVED

Permanent metadata brick due to missing UTF-8 validation at registration

Informational · 0.6 AO:S/AC:L/AX:L/R:N/S:U/C:N/A:L/I:L/D:N/Y:N

DESCRIPTION

The `RegisterStablecoin`, `WriteTokenMetadata`, and `UpdateTokenMetadata` instructions accept `name`, `symbol`, and `uri` as fixed-size byte arrays, which the gateway stores in the Stablecoin PDA and later emits into the mint's Token-2022 TokenMetadata TLV through Borsh-encoded CPIs.

However, the gateway's only content check on these fields is `check_null_padded()`, which verifies NUL padding without inspecting the prefix as UTF-8. Token-2022's TokenMetadata extension deserialises every field value as a Borsh `String`, which runs `core::str::from_utf8` and rejects any non-UTF-8 sequence, so bytes the Token-2022 CPI will later reject can still be persisted in the Stablecoin PDA.

`check_null_padded()` locates the first `0x00` and verifies the tail is NUL; the prefix is never examined.

svm/programs/issuer-gateway/src/validation.rs

RUST 15–21

```rust
pub fn check_null_padded(field: &[u8]) -> Result<usize, GatewayError> {
    let len = field.iter().position(|&b| b == 0).unwrap_or(field.len());
    if field[len..].iter().any(|&b| b != 0) {
        return Err(GatewayError::FieldNotNullPadded);
    }
    Ok(len)
}
```

Downstream, `BorshWriter::string()` serialises those bytes into the CPI payload with a length prefix and no further check, deferring UTF-8 validation entirely to Token-2022's deserialiser.

svm/programs/issuer-gateway/src/actions/metadata/mod.rs

RUST 44–46

```rust
    fn string(self, s: &[u8]) -> Result<Self, ProgramError> {
        self.bytes(&(s.len() as u32).to_le_bytes())?.bytes(s)
    }
```

Once the PDA holds non-UTF-8 bytes, every subsequent `write_token_metadata` call reverts inside Token-2022 when `Initialize`'s Borsh deserialiser fails on the field. `update_token_metadata` cannot repair the PDA because its `UpdateField` CPIs run before the PDA write and require an existing TLV. No other instruction writes `sc.name`, `sc.symbol`, or `sc.uri`, and `register_stablecoin` cannot re-run under the same name. The admin's only remaining path is to abandon the misconfigured stablecoin (whose mint and freeze authorities stay held by its unrecoverable PDA) and register a replacement under a different name.

RECOMMENDATION

To address this finding, it is recommended to validate the prefix bytes of every text field as UTF-8 inside `check_null_padded()` with a `core::str::from_utf8` call before the bytes reach the Stablecoin PDA or the CPI payload.

REMEDIATION COMMENT

Addressed in: e76de30b4e21e0de8707fe0cd532a59a31cdaec5

SOLVED: The `M0` team resolved this finding by refusing the `name`, `symbol` and `uri` parameters that are outside of the printable ASCII subset.

HAL-008 SOLVED

ConfigManager can halt issuance due to missing upper bounds on mint-config knobs

Informational · 0.3 AO:S/AC:L/AX:L/R:F/S:U/C:N/A:M/I:L/D:N/Y:N

DESCRIPTION

The `adjust_stablecoin` instruction lets a holder of `Role::ConfigManager`, or the admin, update the per-stablecoin mint configuration knobs (the mint-proposal delay and the token-bucket rate limit) that bound how a mint executor can issue supply. The same three knobs are set at registration and adoption time through `check_stablecoin_config`.

However, `AdjustStablecoinArgs.parse()` validates only two edge cases on these knobs, a negative `mint_delay` and a zero `rate_limit_capacity`. No upper bound is enforced on any of the three values, so a ConfigManager holder can write a delay or bucket parameters outside any realistic tuning range. The sibling validation at `check_stablecoin_config` in `register_stablecoin.rs`, called by both `process_register_stablecoin` and `process_adopt_stablecoin`, applies the same lower-bound-only check, so the missing upper bound is also reachable at registration and adoption time on admin-only paths.

The parse validation and the admin/ConfigManager write path are shown below.
svm/programs/issuer-gateway/src/actions/lifecycle/adjust_stablecoin.rs

RUST 130–137

```rust
        // A negative delay would make every later proposal executable on arrival.
        if args.mint_delay.is_some_and(|v| v < 0) {
            return Err(GatewayError::InvalidArgument.into());
        }
        // A zero capacity would refuse every mint; unlimited is `u64::MAX`.
        if args.rate_limit_capacity == Some(0) {
            return Err(GatewayError::InvalidArgument.into());
        }
```

The downstream write accepts the parsed values verbatim, with no further bounds check once ConfigManager (or admin) authority is satisfied.
svm/programs/issuer-gateway/src/actions/lifecycle/adjust_stablecoin.rs

RUST 212–222

```rust
        if let Some(v) = args.mint_delay {
            sc.mint_delay = v;
        }
        if args.rate_limit_capacity.is_some() ||
args.rate_limit_refill_per_second.is_some() {
```

```
            sc.set_mint_rate_limit(
                now,
                args.rate_limit_capacity.unwrap_or(sc.rate_limit_capacity),
                args.rate_limit_refill_per_second
                    .unwrap_or(sc.rate_limit_refill_per_second),
            );
        }
```

The out-of-range values then surface in two downstream handlers. In `propose_mint`, `now.checked_add(delay)` fails closed on `mint_delay = i64::MAX`, so every subsequent proposal reverts with `ArithmeticOverflow`.

svm/programs/issuer-gateway/src/actions/issuance/propose_mint.rs

RUST 148–151

```rust
    let delay = Stablecoin::load_checked(&ctx.stablecoin.try_borrow()?)?.mint_delay;
    let ready_at = now
        .checked_add(delay)
        .ok_or(ProgramError::ArithmeticOverflow)?;
```

A ConfigManager holder can therefore push the knobs outside their intended tuning scope. Setting `mint_delay = i64::MAX` halts every subsequent `propose_mint`; setting `rate_limit_capacity = 1` with `refill = 0` makes `execute_mint` and `distribute_rewards` revert with `RateLimitExceeded`. The same bad value is also reachable at registration and adoption time, where an admin typo or a mis-encoded client payload sets `mint_delay` near `i64::MAX` and ships a stablecoin whose issuance is DoSed from the first proposal. The role nominally tunes the throttles that bound minters; being able to halt issuance altogether exceeds that nominal scope. Issuance remains gated by the separate `MintProposer`, `MintExecutor`, and circuit-breaker controls, so the behavior does not grant any additional mint power, and every adjustment emits `StablecoinAdjusted` for monitoring. The behavior matches the EVM reference (`IssuerGateway.setMintConfig`) and is the designed trust model for the role; the observation is a defense-in-depth hardening note rather than a reachable unprivileged exploit.

RECOMMENDATION

To address this finding, it is recommended to enforce admin-configured upper bounds on `mint_delay`, `rate_limit_capacity`, and `rate_limit_refill_per_second` at every write path (`AdjustStablecoinArgs.parse` in `adjust_stablecoin.rs` and `check_stablecoin_config` in `register_stablecoin.rs`) so the role literally cannot leave its intended tuning range. At a minimum, cap `mint_delay` at a value small enough that `now + delay` cannot overflow for any realistic future block timestamp (for example, 30 days), closing the overflow-DoS path on `propose_mint`.

Alternatively, restrict ConfigManager to tightening-only writes (raise the delay, lower the capacity, lower the refill) and require the admin, or a timelock, to loosen them.

REMEDIATION COMMENT

Addressed in: 97f70298edbf144890e64c92ccfb8bd57b5cedf9

SOLVED: The `M0` team resolved this finding by capping `mint_delay` at `u32::MAX` seconds to close the `propose_mint` overflow DoS, and deferring the broader requirement to constrain the mint-config parameters within an operationally reasonable range (including the rate-limit knobs) to admin discretion.

HAL-009 SOLVED

Admin-writable config fields accept zero or malformed values without guards

Informational · 0.2 AO:S/AC:L/AX:L/R:F/S:U/C:N/A:L/I:L/D:N/Y:N

DESCRIPTION

The gateway exposes several admin-reachable writes that configure or operate a stablecoin: `register_stablecoin` and `adopt_stablecoin` take a `currency` byte string and a `rewards_recipient` through `check_stablecoin_config`, `set_rewards_recipient` writes a new recipient, and `force_transfer` takes a seizure `amount`. All of these are gated behind privileged roles — the admin or an admin-granted holder. However, four admin-reachable write sites accept malformed or all-zero inputs without an explicit guard.

`check_stablecoin_config` validates `mint_delay`, `rate_limit_capacity`, `earner_rate_bps`, `name`, and `symbol`, but never validates `currency` or `rewards_recipient`. `process_set_rewards_recipient` writes the supplied `new_recipient` into the Stablecoin PDA with no validation. `process_force_transfer` reads `args.amount` and proceeds through the full thaw-transfer-refreeze sequence even when the amount is zero. In `check_stablecoin_config`, five fields are vetted, but `currency` and `rewards_recipient` are not among them.

svm/programs/issuer-gateway/src/actions/lifecycle/register_stablecoin.rs

RUST 208–226

```rust
/// The configuration `register_stablecoin` and `adopt_stablecoin` both accept.
pub(crate) fn check_stablecoin_config(
    rewards_model: u8,
    mint_delay: i64,
    rate_limit_capacity: u64,
    earner_rate_bps: u16,
    name: &[u8],
    symbol: &[u8],
) -> ProgramResult {
    RewardsModel::try_from(rewards_model)?;
    // A negative delay would make every later proposal executable on arrival.
    // A zero capacity would refuse every mint; unlimited is `u64::MAX`.
    if mint_delay < 0 || rate_limit_capacity == 0 || earner_rate_bps > MAX_RATE_BPS {
        return Err(GatewayError::InvalidArgument.into());
    }
    check_null_padded(name)?;
    check_null_padded(symbol)?;
    Ok(())
}
```

`check_null_padded` enforces only the null-padded portion of the documented format for `name` and `symbol` (rejecting intermediate NULs such as `U\0SD`), but does not inspect the prefix bytes for ASCII or UTF-8 validity (that concern is treated in a separate finding). `currency` is documented in the state crate as the same ASCII, null-padded, immutable format, but it never reaches even `check_null_padded`. A `currency` value with an intermediate NUL or a non-ASCII byte such as `0xFF` is stored verbatim into the Stablecoin PDA and emitted in the `StablecoinRegistered` and `StablecoinAdopted` events. The CLI and off-chain indexers then read malformed identity metadata that cannot be corrected on-chain, since `currency` is immutable after registration. In `process_set_rewards_recipient`, the supplied `new_recipient` is written directly into the Stablecoin PDA after the role gate with no further validation.

svm/programs/issuer-gateway/src/actions/rewards/set_rewards_recipient.rs

RUST 68–90

```rust
#[inline(always)]
pub fn process_set_rewards_recipient(accounts: &mut [AccountView], _data: &[u8]) ->
ProgramResult {
    let ctx = SetRewardsRecipientAccounts::context(accounts)?;

    let mint_key = *ctx.mint.address();
    let new_recipient = *ctx.new_recipient.address();

    Gate::require(
        Role::RewardsRecipientManager,
        ctx.authority,
        ctx.role_grant,
        &mint_key,
        ctx.issuer_config,
        ctx.stablecoin,
    )?;

    let old_recipient = {
        let mut data = ctx.stablecoin.try_borrow_mut()?;
        let sc = Stablecoin::load_mut_checked(&mut data)?;
        let old = sc.rewards_recipient;
        sc.rewards_recipient = new_recipient;
        old
    };
```

In `process_force_transfer`, the `amount` field is read but not checked for zero before the role gate and the full thaw-transfer-refreeze sequence runs.

svm/programs/issuer-gateway/src/actions/compliance/force_transfer.rs

RUST 127–132

```rust
#[inline(always)]
pub fn process_force_transfer(accounts: &mut [AccountView], data: &[u8]) -> ProgramResult
{
    let ctx = ForceTransferAccounts::context(accounts)?;
```

```
131
132         let args = ForceTransferArgs::load(data)?;
            let amount = args.amount;
```

A zero `rewards_recipient` bricks every subsequent `claim_accrued` call on the affected stablecoin: the destination owner check in `claim_accrued` compares `destination.owner()` against the stored recipient, and no Token2022 account owner can hold the all-zero address. Accrued rewards are not lost — they continue to accumulate in the index until a corrective `set_rewards_recipient` fixes the value — but the denial of service is admin-reachable and trivially avoidable with a parse-time check.

A malformed `currency` is strictly worse to recover from: the field is immutable at registration, so a non-ASCII or embedded-NUL value is permanently stored in the PDA and in the registration events, degrading every downstream integrator that parses the identity bytes.

A zero `amount` in `force_transfer` succeeds end-to-end: both accounts are thawed, a no-op `TransferChecked(0)` is performed, both accounts are refrozen, and a `ForceTransferred { amount: 0 }` event is emitted. No funds move, but the event log is polluted with spurious seizure records, degrading audit-trail quality.

All failure modes require a trusted role holder — admin, `RewardsRecipientManager`, or `ForcedTransferManager` — making this a defense-in-depth hardening concern against operator typos and misconfigured deployments rather than an exploitable vulnerability.

RECOMMENDATION

To address this finding, it is recommended to reject malformed and all-zero values at every admin-reachable write path, so that a typo cannot reach the stablecoin state. Specifically:

- In `check_stablecoin_config`, pass `currency` and `rewards_recipient` into the validator, apply `check_null_padded` to `currency` (ideally paired with an ASCII-only check matching the state crate's documented format), and reject `rewards_recipient == Address::default()`. Both `register_stablecoin` and `adopt_stablecoin` route through this validator, so one fix covers both callers.
- In `process_set_rewards_recipient`, reject `new_recipient == Address::default()` with a dedicated error such as `InvalidArgument` or a new `RewardsRecipientUnset`.
- In `process_force_transfer`, reject `amount == 0` right after `ForceTransferArgs::load`, matching the existing `ZeroMintAmount` convention used in `propose_mint`.

Additionally, a single `is_renounced(addr)` or `Address::is_default()` helper in `util/` would centralize the sentinel-address convention.

REMEDIATION COMMENT

Addressed in: 4856710e9008f2505ba538d162c60d94032971f0

SOLVED: The `M0` team resolved this finding by rejecting malformed and all-zero inputs at every admin-reachable write path flagged in the finding.

HAL-010 ACKNOWLEDGED

Cross-proposer griefing due to missing proposer-ownership check

Informational · 0.2 AO:S/AC:L/AX:L/R:F/S:U/C:N/A:L/I:L/D:N/Y:N

DESCRIPTION

The `process_cancel_mint` handler in the SVM Issuer Gateway discards an outstanding mint proposal before it executes. It is callable by the issuer admin or by any holder of `Role::MintProposer` for the stablecoin.

However, `process_cancel_mint` never compares the signing authority against the proposal's recorded `proposer`. Any `MintProposer` holder for a given mint can therefore close any live proposal on that mint, including proposals created by the admin or by a peer `MintProposer`.

After loading the account context, the handler calls `Gate::require(Role::MintProposer, ...)` to authorize the caller. It then loads the proposal and extracts its `proposer` field, but that field flows only into the emitted `MintCancelled` event, never into an identity check. The handler proceeds straight to `close_program_account`, which refunds the proposal's rent to a `rent_recipient` the caller picks.

svm/programs/issuer-gateway/src/actions/issuance/cancel_mint.rs

RUST 77–111

```rust
    Gate::require(
        Role::MintProposer,
        ctx.authority,
        ctx.role_grant,
        &mint_key,
        ctx.issuer_config,
        ctx.stablecoin,
    )?;

    if ctx.proposal.data_len() == 0 {
        return Err(GatewayError::MintProposalNotFound.into());
    }
    check_program_owned(ctx.proposal)?;

    // Prove the account really is this mint's proposal before destroying it.
    let (nonce, bump, amount, proposer, destination, destination_owner) = {
        let data = ctx.proposal.try_borrow()?;
        let proposal = MintProposal::load_checked(&data)?;
        (
            proposal.nonce,
            proposal.bump,
            proposal.amount,
            proposal.proposer,
            proposal.destination,
            proposal.destination_owner,

```

```
            )
        };
        check_pda(
            ctx.proposal,
            &mint_proposal_pda_seeds_with_bump(&mint_key, &nonce, &[bump]),
        )?;

        // Zeroes the data before releasing the account, so nothing later in
        // this transaction can read a cancelled proposal as live.
        close_program_account(ctx.proposal, ctx.rent_recipient)?;
```

A single misbehaving or compromised `MintProposer` delegate can therefore halt all issuance on its mint until the admin revokes the grant: by repeatedly cancelling the admin's and other proposers' proposals, it keeps every pending mint from maturing into `process_execute_mint`. The rent of each cancelled proposal is also redirected to a `rent_recipient` of the attacker's choice. The behavior diverges from the EVM `IssuerGateway.cancelMint()`, which lets only the original minter or a `MINT_CANCELER_ROLE` holder close a proposal; the SVM port drops the separation of duties between proposing and cancelling.

RECOMMENDATION

To address this finding, it is recommended to require that `ctx.authority` equals the proposal's `proposer`, or that the authority is the issuer admin, inside `process_cancel_mint`. Alternatively, introduce a dedicated `Role::MintCanceller` and verify it with `verify_grant` when the signer is not the proposer. Preserve the ability for anyone to clean up proposals whose proposer no longer holds `MintProposer`, following the same `since_slot` logic already used in `process_execute_mint`.

REMEDIATION COMMENT

ACKNOWLEDGED: The `M0` team acknowledged this finding, decided to keep the original implementation, and added the following note:

Any MintProposer may cancel any pending proposal by design, an intentional divergence from EVM that the spec documents. A griefing proposer is a delegated key the admin granted, and the admin removes it with revoke_roles or close_role_grant. The admin's own proposals cannot be cancelled away permanently, since the admin can re-propose at any time.

Automated Testing

Static Analysis Report

Description

Halborn used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was `cargo audit`, a security scanner for vulnerabilities

reported to the RustSec Advisory Database. All vulnerabilities published in `https://crates.io` are stored in a repository named The RustSec Advisory Database. `cargo audit` is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Cargo Audit Results

ID	Crate	Description
RUSTSEC-2026-0343	quinn-proto	Outgoing datagram buffer accounting can panic the sender in quinn-proto
RUSTSEC-2026-0338	quinn-proto	Remote peer can panic quinn-proto with zero-length STREAM frames during unordered reads
RUSTSEC-2026-0340	quinn-proto	Peer minimum ACK delay can panic quinn-proto when ACK_FREQUENCY transmission is enabled
RUSTSEC-2026-0285	rustls	TLS 1.3 handshake messages incorrectly accepted across encryption level boundaries

Disclaimer

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.