Smart Contract Assessment

Gold Token SA / DGLD (MKS Pamp)

HALBORN

Smart Contract Assessment - Gold Token SA / DGLD (MKS Pamp)

Prepared by: HALBORN

Last Updated 07/23/2026

Date of Engagement: July 14th, 2026 - July 15th, 2026

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
8	0	0	0	0	8

TABLE OF CONTENTS

8.8 Missing validation in on role/authority setters may result in no-op and emit misleading state-change events

1. Summary

The `Gold Token SA team` engaged `Halborn` to conduct a security assessment of their **GDGLD Solana program** beginning on July 14, 2026, and ending on July 15, 2026. The security assessment was scoped to the Solana Program provided in gtsa-solana GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The `dgld` program is a Solana program implementing a gold-backed token on the Token-2022 standard, maintaining parity with an existing Ethereum DGLD contract. Each token represents one troy ounce of physical gold, and total supply is designed to stay fully backed by the aggregate fine weight of physical bars, each registered as a PDA that carries provenance metadata and status. Tokens are minted against newly registered bars (issuance), burned directly from a holder via the Token-2022 permanent delegate (forced redemption), and can be restored by the admin when burned out-of-band (recovery mint), with every mint and burn wrapped in strict supply-invariant checks.

2. Assessment Summary

`Halborn` was provided 2 days for the engagement and assigned one full-time security engineer to review the security of the Solana Program 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, some of which have been addressed by `Gold Token SA` team. The primary findings were as follows:

- `Parse and validate the mint's extension data, require PermanentDelegate extension to be present and reject any unexpected extension`
- `Refund the closed Bar PDA rent to the account that originally funded it rather than to the redeemer.`
- `Derive the reported fee from the tokens actually credited to the fee wallet or expose both fees quantities in the event.`
- `Remove the inconsistencies in the documentation and dead code.`
- `Enforce an on-chain link between the redeemed Bar and the debited account, and to preserve the Bar's provenance in the emitted event.`
- `Complete the string validation for certificate_of_deposit, inventory_report and validate that all five fields are printable ASCII.`
- `Add a validate fineness in the handler, enforcing a sensible domain range and reject out-of-range values`
- `Add an idempotency guard to each setter so a no-op reverts explicitly rather than emitting a spurious event.`

3. 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 (`cargo test `)

4. Automated Testing

4.1 Static Analysis Report

4.2 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

```
No vulnerabilities were found.
```

5. 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.

5.1 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
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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:

$$E = \prod m_e$$

5.2 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

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
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:

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

5.3 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
Informational	0 - 1.9

6. SCOPE

REPOSITORY

(a) Repository: gtsa-solana

(b) Assessed Commit ID: 4de54db

(c) Items in scope:

- programs/dgld/Cargo.toml
- programs/dgld/src/errors.rs
- programs/dgld/src/instructions/authorities.rs
- programs/dgld/src/instructions/fees.rs
- programs/dgld/src/instructions/initialize_config.rs
- programs/dgld/src/instructions/issue_bar.rs
- programs/dgld/src/instructions/recover_mint.rs
- programs/dgld/src/instructions/redeem_bar.rs
- programs/dgld/src/instructions/terms.rs
- programs/dgld/src/state.rs
- programs/dgld/src/accounting.rs
- programs/dgld/src/lib.rs
- programs/dgld/src/utils.rs
- programs/dgld/src/events.rs
- programs/dgld/src/constants.rs
- programs/dgld/src/bar_id.rs

FILE

(a) Submitted File: gtsa-solana-main.zip

(b) Items in scope:

- /gtsa-solana-main/programs/dgld/src/instructions/authorities.rs
- /gtsa-solana-main/programs/dgld/src/instructions/fees.rs
- /gtsa-solana-main/programs/dgld/src/instructions/initialize_config.rs
- /gtsa-solana-main/programs/dgld/src/instructions/issue_bar.rs
- /gtsa-solana-main/programs/dgld/src/instructions/mod.rs
- /gtsa-solana-main/programs/dgld/src/instructions/recover_mint.rs
- /gtsa-solana-main/programs/dgld/src/instructions/redeem_bar.rs
- /gtsa-solana-main/programs/dgld/src/instructions/terms.rs
- /gtsa-solana-main/programs/dgld/src/accounting.rs
- /gtsa-solana-main/programs/dgld/src/bar_id.rs
- /gtsa-solana-main/programs/dgld/src/constants.rs
- /gtsa-solana-main/programs/dgld/src/errors.rs

- /gtsa-solana-main/programs/dgld/src/events.rs
- /gtsa-solana-main/programs/dgld/src/lib.rs
- /gtsa-solana-main/programs/dgld/src/state.rs
- /gtsa-solana-main/programs/dgld/src/utils.rs
- /gtsa-solana-main/programs/dgld/Cargo.toml

REMEDIATION COMMIT ID:

- 4d24643
- f745510
- d4ef76a
- 504bae3

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

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

INFORMATIONAL
8

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING ON-CHAIN VALIDATION OF TOKEN-2022 EXTENSION DURING THE INITIALIZATION	INFORMATIONAL	ACKNOWLEDGED - 07/21/2026
EVENT BARREDEEMED FEE AMOUNT MAY NOT REFLECT THE FEE ACTUALLY RECEIVED BY THE PROTOCOL WHEN THE MINT'S TRANSFER FEE IS NON-ZERO	INFORMATIONAL	ACKNOWLEDGED - 07/20/2026
STALE DOCUMENTATION AND DEAD STATE AROUND THE BAR ACCOUNT LIFECYCLE AFTER THE SWITCH TO CLOSING THE PDA ON REDEMPTION	INFORMATIONAL	SOLVED - 07/20/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
FORCED REDEMPTION MAY BURN FROM A DIFFERENT HOLDER UNRELATED TO THE CLOSED BAR	INFORMATIONAL	ACKNOWLEDGED - 07/23/2026
PROVENANCE METADATA IN ISSUE_BAR IS WEAKLY VALIDATED AND OMITTED FROM THE BARISSUED EVENT	INFORMATIONAL	SOLVED - 07/20/2026
ASYMMETRIC BAR PDA RENT FLOW	INFORMATIONAL	ACKNOWLEDGED - 07/23/2026
MISSING VALIDATION OF FINENESS AND ITS OMISSION FROM THE BARISSUED EVENT	INFORMATIONAL	SOLVED - 07/20/2026
MISSING VALIDATION IN ON ROLE/AUTHORITY SETTERS MAY RESULT IN NO-OP AND EMIT MISLEADING STATE-CHANGE EVENTS	INFORMATIONAL	SOLVED - 07/20/2026

8. FINDINGS & TECH DETAILS

8.1 MISSING ON-CHAIN VALIDATION OF TOKEN-2022 EXTENSION DURING THE INITIALIZATION

// INFORMATIONAL

Description

According to the token deployment documentation (`token/README.md`), the DGLD mint is a Token-2022 mint whose locked extension set comprises `PermanentDelegate`, `Pausable`, and `TransferFeeConfig` (configured at 0 bps), together with a base `FreezeAuthority` that is never set to `None`.

During deployment, `initialize_config` is responsible for validating this mint before recording it as the protocol's single source of truth in `config.mint`. However, the validation performed at genesis is restricted to the mint's base fields, decimals and supply. The instruction inspects none of the mint's extensions: it neither confirms that the expected extensions are present, nor rejects unexpected ones such as a `TransferHook`.

This omission is material because the `redeem_bar` instruction relies entirely on the `PermanentDelegate` extension. The `permanent_delegate` account is an unbound signer, and both the fee `transfer_checked` and the `burn_checked` of `W` operate on an arbitrary holder's account using that account as the authority. Token-2022 authorizes an authority acting over a third party's token account only when it is the mint's permanent delegate; if the extension is absent, both cross-program invocations revert unconditionally.

Because `config.mint` is fixed immutably at initialization — no instruction exists to modify it thereafter — and the base checks continue to pass, a mint lacking the permanent delegate would be accepted at genesis and would fail only later, at CPI execution time. In that state, `issue_bar` continues to operate normally, as it mints through the `[mint_authority]` PDA and does not require the delegate, whereas `redeem_bar` reverts on every invocation.

The result is a permanent denial of service on redemption: supply can be issued but never retired from circulation, undermining the token's backing model, with recovery possible only through a program upgrade rather than through normal operation.

programs/dgld/src/instructions/initialize_config.rs

Copy Code

```
83  let (_, mint_authority_bump) =
84          Pubkey::find_program_address(&[MINT_AUTHORITY_SEED], ctx.program_id);
85      let mint_key = ctx.accounts.mint.key();
86
87      let config = &mut ctx.accounts.config;
88      config.version = 1;
89      config.config_bump = ctx.bumps.config;
90      config.mint_authority_bump = mint_authority_bump;
91      config.admin = params.admin;
92      config.pending_admin = Pubkey::default();
93      config.issuer = params.issuer;
94      config.redeemer = params.redeemer;
```

```
96        config.fee_wallet = params.fee_wallet;
          config.mint = mint_key;
```

The same class of gap extends to the remaining extensions. A non-zero `TransferFeeConfig` would silently divert a portion of the redemption fee away from the fee wallet, as mentioned in the other finding, and an unexpected extension such as a `TransferHook` could introduce arbitrary revert conditions or additional logic into transfers and burns.

Although this does not represent an exploitable security vulnerability—this condition is not reachable by an external attacker, as `initialize_config` is deployer-pinned and may be invoked only by the program's upgrade authority—it is recommended to implement the suggestied fix.

BVSS

AO:S/AC:L/AX:M/R:N/S:U/C:N/A:H/I:M/D:C/Y:N (1.8)

Recommendation

To address the issue, it is recommended to have `initialize_config` parse the mint's extension data on-chain and:

- Require the `PermanentDelegate` extension to be present, rejecting the initialization with a dedicated error otherwise.
- Reject any unexpected extension — notably `TransferHook` — so that the documentation's manual `spl-token display` gate becomes an enforced on-chain invariant.

Additionally, consider to extend this validation to the full locked extension set documented in `token/README.md` to keep the on-chain checks aligned with the intended mint shape: assert that `TransferFeeConfig`, if present, is set to 0 bps; that the base `FreezeAuthority` is not `None`; and that `Pausable` is configured as expected.

Remediation Comment

ACKNOWLEDGED: The `Golden Token SA` team acknowledged this finding.

8.2 EVENT BARREDEEMED FEE AMOUNT MAY NOT REFLECT THE FEE ACTUALLY RECEIVED BY THE PROTOCOL WHEN THE MINT'S TRANSFER FEE IS NON-ZERO

// INFORMATIONAL

Description

The DGLD mint is a Token-2022 mint whose locked extension set includes `TransferFeeConfig`, initialized at deployment with a rate of **0 bps**. On that assumption, `redeem_bar` collects the redemption fee by transferring `debit.fee` tokens from the holder to the protocol's `fee_token_account` via `transfer_checked`, and then emits a `BarRedeemed` event reporting `fee: debit.fee.`

programs/dgld/src/instructions/redeem_bar.rs

Copy Code

```
181 emit!(BarRedeemed {
182         bar: bar_key,
183         bar_id_hash: params.bar_id_hash,
184         holder,
185         fine_weight: w,
186         fee: debit.fee,
187         total_active_fine_weight: total_active,
188         supply_after,
189     });
```

`debit.fee` represents the amount debited from the holder, and the code implicitly treats it as identical to the amount credited to the fee wallet. This holds only while the mint's transfer fee is 0. If the fee rate is ever non-zero, Token-2022 withholds a fraction of the transferred amount into the destination's separate withheld balance — so `fee_token_account.amount` increases by less than `debit.fee`, while the holder is still debited the full amount. In that case the two quantities diverge, yet the event keeps emitting `debit.fee`.

Because `transfer_checked` does not change supply, the supply invariant (`supply_after == supply_before − W`) still passes and nothing surfaces the mismatch on-chain. The consequence is that `BarRedeemed.fee` no longer represents the fee the protocol actually received, which may corrupt any off-chain accounting, indexing, or revenue reconciliation that trusts the event as the source of truth.

programs/dgld/src/instructions/redeem_bar.rs

Copy Code

```
101 let debit = redeem_debit(w, bips)?; // total_debit = W + fee
102
103   // Fee wallet owner binding.
104   require_keys_eq!(
105       ctx.accounts.fee_token_account.owner,
106       ctx.accounts.config.fee_wallet,
107       DgldError::InvalidFeeWallet
108   );
109
110   // Holder must cover W + fee.
111   require!(
112       ctx.accounts.holder_token_account.amount >= debit.total_debit,
113       DgldError::InsufficientHolderBalance
114
```

```
    );

    let supply_before = ctx.accounts.mint.supply;
    let holder = ctx.accounts.holder_token_account.owner;

    // 1. Fee force-transfer holder -> fee wallet (supply-neutral), authorized by the permanent
    //    delegate (Token-2022 accepts the permanent delegate as the transfer authority).
    if debit.fee > 0 {
        transfer_checked(
            CpiContext::new(
                ctx.accounts.token_program.to_account_info(),
                TransferChecked {
                    from: ctx.accounts.holder_token_account.to_account_info(),
                    mint: ctx.accounts.mint.to_account_info(),
                    to: ctx.accounts.fee_token_account.to_account_info(),
                    authority: ctx.accounts.permanent_delegate.to_account_info(),
                },
            ),
            debit.fee,
            DECIMALS,
        )?;
    }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

To address the issue consider one of the following measures:

- `redeem_bar` derive the reported fee from the tokens actually credited to the fee wallet rather than from the pre-transfer debit amount. This can be achieved by reloading `fee_token_account` and computing the balance delta observed across the `transfer_checked` invocation, so that the value emitted in `BarRedeemed` reflects the amount effectively received by the protocol.
- In case downstream systems require full transparency, the event may instead expose both quantities explicitly — `fee_charged`, denoting the amount debited from the holder, and `fee_received`, denoting the amount ultimately credited to the fee wallet — thereby preserving an accurate on-chain record of any discrepancy between the two.

As a complementary operational measure, consider the mint's transfer-fee configuration authority to be renounced (set to `None`) following deployment. This renders the 0-bps rate immutable and guarantees that the debited and credited amounts remain equal, eliminating the underlying condition at its source.

Remediation Comment

ACKNOWLEDGED: The `Gold Token SA` team acknowledged this finding, stating that GTSA, as the token issuer, is the sole recipient of both the transfer fee and the redemption fee. As a result, any divergence between fee charged and fee received while a non-zero transfer fee is active nets to the same beneficiary and does not lead to value leakage.

The client further noted that the current design intentionally prioritizes keeping the redemption flow functional over reverting or introducing additional balance-delta reconciliation logic, an approach the

team assessed as disproportionate given the informational severity and limited practical impact of the discrepancy.

8.3 STALE DOCUMENTATION AND DEAD STATE AROUND THE BAR ACCOUNT LIFECYCLE AFTER THE SWITCH TO CLOSING THE PDA ON REDEMPTION

// INFORMATIONAL

Description

The `Bar` account was redesigned from a *"mark as `Redeemed` and retain"* model to a *"close the PDA on redemption"* model, but the documentation and the data model were not updated to match. The result is a set of contradictory comments, an unreachable enum variant, and dead fields, all clustered around the `Bar` lifecycle.

Individually these are cosmetic; together they misrepresent the account's guarantees to a reader and to any off-chain consumer relying on the code as documentation.

`State` Claims The `Bar` PDA Is Never Closed

The state file documents the account as created with `init` and *"never closed (permanent provenance + permanent dup-guard)"*. This result in an inconsistency.
`redeem_bar` declares `close = redeemer`, which closes the PDA on redemption. Consequently the `init`-based duplicate guard is only *temporary* (it holds while the bar is `Active`), not *permanent* as documented. The `state.rs` comment also directly contradicts `redeem_bar`, which explicitly states the PDA is freed *"so the same physical bar (serial) can be issued again later"*. The two comments describe opposite guarantees.

programs/dgld/src/state.rs

Copy Code

```
/// One per physical bar. PDA `[b"bar", sha256(canonical_bar_id)]`. Created with `init`
/// (hard dup-guard) and never closed (permanent provenance + permanent dup-guard).
#[account]
#[derive(InitSpace)]
pub struct Bar {
```

programs/dgld/src/instruction/redeem_bar.rs

Copy Code

```
#[account(
        mut,
        seeds = [BAR_SEED, params.bar_id_hash.as_ref()],
        bump = bar.bump,
        constraint = bar.status == BarStatus::Active @ DgldError::BarNotActive,
        // Free the Bar PDA on redemption so the same physical bar (serial) can be issued agai
        // (issue_bar's `init` re-creates it). Rent is returned to the redeemer. While a bar i
        // its PDA still exists, so double-issuing an *active* bar stays blocked by that `init
        // redemption record survives off-account in the emitted `BarRedeemed` event.
        close = redeemer,
    )]
    pub bar: Account<'info, Bar>,
```

`BarStatus::Redeemed` Is Defined But Never Written

`BarStatus::Redeemed` is defined in `state.rs`, and this file documents the field as

programs/dgld/src/state.rs

```
4  pub enum BarStatus {
5      Active,
6      Redeemed,
7  }
```

A repository-wide grep confirms `BarStatus::Redeemed` is never assigned in any handler — the only occurrences are the enum definition, the layout comment, and the unrelated `BarRedeemed` event name. `redeem_bar` closes the account instead of transitioning its status, so the `Redeemed` variant is dead / unreachable state.

`Redeemed_at` Is Documented As Populated On Redemption But Never Written.

The `state.rs` file declares `redeemed_at: i64` with the comment `// 0 while Active`, implying a non-zero value is written on redemption. It is only ever set to `0` at issuance; no handler updates it, and the account is closed before any redemption timestamp could be recorded. The field is dead.

programs/dgld/src/instruction/issue_bar.rs

```
195  bar.issued_at = clock.unix_timestamp;
196          bar.issued_slot = clock.slot;
197          bar.redeemed_at = 0;
198      }
```

The Active `Bar Status` Constraint In `Redeem_bar` Is Always True.

`redeem_bar` enforces a constraint to ensure the bar status is Active. Since `Redeemed` is never written (item 2), every existing `Bar` is always `Active`, so the constraint can never fail. It is effectively dead code / a misleading defensive check that suggests a status transition the program does not actually perform.

programs/dgld/src/instruction/redeem_bar.rs

```
46  #[account(
47          mut,
48          seeds = [BAR_SEED, params.bar_id_hash.as_ref()],
49          bump = bar.bump,
50          constraint = bar.status == BarStatus::Active @ DgldError::BarNotActive,
51          // Free the Bar PDA on redemption so the same physical bar (serial) can be issued agai
52          // (issue_bar's `init` re-creates it). Rent is returned to the redeemer. While a bar i
53          // its PDA still exists, so double-issuing an *active* bar stays blocked by that `init
54          // redemption record survives off-account in the emitted `BarRedeemed` event.
55
```

```
        close = redeemer,
    )]
    pub bar: Account<'info, Bar>,
```

`BarRedeemed` Drops The Original Recipient.

`events.rs` documents `BarRedeemed` as recording data *"for audit attribution of the non-consensual redemption"*, but it emits only `holder` (the owner of the burned account) and does not emit `bar.recipient` (the original recipient recorded at issuance). Because the `Bar` account is closed on redemption, that provenance field is lost, and off-chain reconciliation cannot compare the burned holder against the originally-registered recipient.

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:N/I:L/D:N/Y:N (1.3)

Recommendation

To address the issue consider the following measures:

- Update the `Bar` documentation in `state.rs` to reflect that the PDA is closed on redemption, clarifying that the duplicate guard only protects `Active` bars.
- Reconcile the contradictory comments between `state.rs` and `redeem_bar.rs` so a single, accurate lifecycle description remains.
- Remove the dead `BarStatus::Redeemed` variant and the unused `redeemed_at` field (and the always-true `bar.status == Active` constraint) if the close-on-redemption model is intended; alternatively, if a persistent redemption record is desired, revert to marking bars `Redeemed` and populating `redeemed_at` instead of closing the account.
- If off-chain audit attribution requires linking the burned holder to the original recipient, add `bar.recipient` (and, if useful, the redemption timestamp) to the `BarRedeemed` event before the account is closed, since that data is otherwise unrecoverable.

Remediation Comment

SOLVED: The `Gold Token SA` team solved this issue by fully redesigning the `Bar` data model to match the "close-on-redemption" behavior. Specifically:

- `state.rs` no longer contradicts `redeem_bar`.
- The dead `BarStatus` enum was removed entirely and the always-true "Active" constraint was removed from `redeem_bar`.
- The dead `redeemed_at` field was removed from the `Bar` struct, eliminating the field that was documented as populated on redemption but never written.
- `BarRedeemed`'s comment was rewritten to only claim what it actually emits — `holder` for audit attribution — removing the false promise of recording the original recipient.

Remediation Hash

4d246438afe95f7055955fd2e9a5a41ad804a9e9

8.4 FORCED REDEMPTION MAY BURN FROM A DIFFERENT HOLDER UNRELATED TO THE CLOSED BAR

// INFORMATIONAL

Description

`redeem_bar` performs a forced redemption from two independently supplied inputs that the program never cross-checks:

1. The Bar to close, selected by `params.bar_id_hash`, which drives the redeemed weight `W = bar.fine_weight`.
2. The holder account to debit and burn from, `holder_token_account`, whose only constraints are `token::mint = mint` and `token::token_program`.

No constraint or `require!` links these two inputs. The handler force-transfers the fee and burns exactly `W` from the holder under `permanent_delegate` authority, then closes the Bar. The sole check against the holder is that its balance covers `W + fee`.

Consequently, the caller (`redeemer` together with `permanent_delegate`) can pass the `bar_id_hash` of Bar belonging to user **X** and the `holder_token_account` of an unrelated user **Y**, provided Y holds `W + fee`. The result: `W` tokens are burned (plus the fee transferred) from **Y**, while the Bar attributed to **X** is closed — even though X still holds the tokens minted for that Bar.

This does not break supply integrity: `W` tokens are burned and the supply invariant `supply_after == supply_before - W` still holds. The defect is one of **attribution/provenance**: the on-chain link between a specific Bar's gold backing and the account that actually bore its redemption is severed. The correct pairing of holder to Bar rests entirely on off-chain operational discipline.

programs/dgld/src/instructions/issue_bar.rs

Copy Code

```
183 {
184     let bar = &mut ctx.accounts.bar;
185     bar.bump = ctx.bumps.bar;
186     bar.status = BarStatus::Active;
187     bar.fine_weight = w;
188     bar.fineness = params.fineness;
189     bar.partner_id = params.partner_id;
190     bar.custodian = params.custodian;
191     bar.producer_id = params.producer_id;
192     bar.certificate_of_deposit = params.certificate_of_deposit;
193     bar.inventory_report = params.inventory_report;
194     bar.recipient = params.recipient;
195     bar.issued_at = clock.unix_timestamp;
196     bar.issued_slot = clock.slot;
197     bar.redeemed_at = 0;
```

programs/dgld/src/instructions/redeem_bar.rs

Copy Code

```
117 let holder = ctx.accounts.holder_token_account.owner;
118
119     // 1. Fee force-transfer holder -> fee wallet (supply-neutral), authorized by the permanen
120     //    delegate (Token-2022 accepts the permanent delegate as the transfer authority).
121
```

```
    if debit.fee > 0 {
        transfer_checked(
            CpiContext::new(
                ctx.accounts.token_program.to_account_info(),
                TransferChecked {
                    from: ctx.accounts.holder_token_account.to_account_info(),
                    mint: ctx.accounts.mint.to_account_info(),
                    to: ctx.accounts.fee_token_account.to_account_info(),
                    authority: ctx.accounts.permanent_delegate.to_account_info(),
                },
```

Furthermore, The `BarRedeemed` event emits `holder` but not `bar.recipient`. Because the Bar account is closed on redemption, its `recipient` (provenance) field is destroyed on-chain and is absent from the event as well. An off-chain observer therefore cannot detect a mismatch (redeemed `holder` ≠ the Bar's original `recipient`) from the event stream alone, removing the only realistic avenue for reconciliation.

Related Observations

- **No `redeemer != permanent_delegate` enforcement**: a single key may satisfy both required signers, collapsing the intended two-party control. Combined with the absence of any holder anchoring, a single compromised or erroneous role can execute an arbitrary-holder burn against an unrelated Bar.
- **Caller-controlled fee**: `redemption_fee_bips` is supplied per call and imposed on a non-consenting holder, bounded only by `MAX_FEE_BIPS` (1%).

BVSS

AO:S/AC:L/AX:L/R:P/S:U/C:N/A:N/I:H/D:C/Y:N (1.2)

Recommendation

To address the issue, it is recommended to enforce an on-chain link between the redeemed `Bar` and the debited account, and to preserve the Bar's provenance in the emitted event:

- **Anchor the holder to the Bar:** Add a constraint on `holder_token_account` requiring its owner to equal `bar.recipient`, so tokens can only ever be burned from the account that actually backs the selected Bar:

Copy Code

```
#[account(
    mut,
    token::mint = mint,
    token::token_program = token_program,
    constraint = holder_token_account.owner == bar.recipient @ DgldError::HolderRecipientMismatch,
)]
pub holder_token_account: InterfaceAccount<'info, TokenAccount>,
```

If the design must support tokens that have been transferred away from the original recipient, instead require the caller to declare the target holder as an explicit, signed parameter (`params.expected_holder`) and verify it with `require_keys_eq!(holder_token_account.owner, params.expected_holder, ...)`, turning a silent mismatch into an auditable, committed value.

- **Preserve provenance in the event:** Capture `bar.recipient` before the Bar is closed and add it to `BarRedeemed` alongside `holder`, so that any `recipient != holder` mismatch remains detectable from the event stream after the account is destroyed

Remediation Comment

ACKNOWLEDGED: The `Gold Token SA` team acknowledged this finding, stating that The Bar PDA is not coupled on-chain to a specific owner, and redemption targets whoever holds the tokens at redemption time — who, by design, may or may not be the original recipient recorded at issuance, since the tokens may have been transferred any number of times.

8.5 PROVENANCE METADATA IN ISSUE_BAR IS WEAKLY VALIDATED AND OMITTED FROM THE BARISSUED EVENT

// INFORMATIONAL

Description

`issue_bar` accepts five provenance-metadata fields in `IssueBarParams` — `partner_id`, `custodian`, `producer_id`, `certificate_of_deposit`, and `inventory_report` — that describe the custodial backing of each newly issued bar. The last two are custody documents (a certificate-of-deposit reference and an inventory report) that legally substantiate the physical gold behind the token.

However, these fields are validated **only** against a maximum byte length, raising `FieldTooLong` when exceeded. There is no lower bound and no content validation. As a result, a bar can be issued with `certificate_of_deposit = ""`, an empty `inventory_report`, or fields containing arbitrary non-printable or non-ASCII bytes. The values are written verbatim into the `Bar` account and become the permanent on-chain provenance record.
This may result inconsistent with the validation applied to other strings in the same program, which is notably stricter for data of lower legal significance:

- `bar_id` is passed through `canonicalize()` (bar_id.rs:20-27), which rejects empty, oversize, non-ASCII, and disallowed-charset values (`InvalidBarId`).
- `terms_url` is rejected when empty in both `set_terms_url` (terms.rs:24-27) and `initialize_config` (`InvalidTermsUrl`).

programs/dgld/src/instructions/issue_bar.rs

Copy Code

```
    // 4. Bounded metadata length checks (account space is pre-allocated for the max; never reall
    require!(
        params.partner_id.len() <= PARTNER_ID_MAX,
        DgldError::FieldTooLong
    );
    require!(
        params.custodian.len() <= CUSTODIAN_MAX,
        DgldError::FieldTooLong
    );
    require!(
        params.producer_id.len() <= PRODUCER_ID_MAX,
        DgldError::FieldTooLong
    );
    require!(
        params.certificate_of_deposit.len() <= COD_MAX,
        DgldError::FieldTooLong
    );
    require!(
        params.inventory_report.len() <= INVENTORY_MAX,
        DgldError::FieldTooLong
    );
```

programs/dgld/src/bar_id.rs

Copy Code

```
pub fn canonicalize(raw: &str) -> Result<String> {
    let c = raw.trim().to_ascii_uppercase();
```

```
    require!(!c.is_empty(), DgldError::InvalidBarId);
    require!(c.len() <= BAR_ID_MAX_LEN, DgldError::InvalidBarId);
    require!(c.is_ascii(), DgldError::InvalidBarId);
    require!(c.chars().all(is_allowed), DgldError::InvalidBarId);
    Ok(c)
```

Separately, none of the five metadata fields are included in the `BarIssued` event, which emits only `bar`, `bar_id`, `bar_id_hash`, `fine_weight`, `fee`, `to_recipient`, `recipient`, `total_active_fine_weight`, and `supply_after`.

Provenance data (custodian, producer, and both custody documents) exists solely in the `Bar` account and never appears in the event stream. Off-chain indexers, auditors, or reconciliation systems that consume events as their source of truth cannot observe the provenance recorded at issuance without separately fetching and decoding each `Bar` account.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

It is recommended to apply the same rigor used for `bar_id`/`terms_url` to the provenance fields — at minimum reject empty values for the custody documents (`certificate_of_deposit`, `inventory_report`), and validate that all five fields are printable ASCII (or otherwise conform to their expected format).

Additionally, consider to include provenance in the event, if it is convenient. For this purpose, add the metadata fields to `BarIssued` so that the complete provenance recorded at issuance is available directly from the event stream.

Remediation Comment

SOLVED: The `Gold Token SA` team solved this issue by:

- Adding new function `validate_provenance_field` which is called for validation in `issue_bar` to enforce that the provenance metadata fields are non-blank, within their length cap, and printable ASCII.
- Adding the corresponding error variants to back the new validation.

Remediation Hash

f7455107ff37a07c9d7fea6a49ba8c90f3d63fe5

8.6 ASYMMETRIC BAR PDA RENT FLOW

// INFORMATIONAL

Description

`issue_bar` creates the `Bar` PDA with `init` and `payer = issuer`, so the `issuer` funds the rent-exempt balance of the account on every issuance. `redeem_bar` **closes** the `Bar` PDA with `close = redeemer`, which sends the account's entire rent-exempt lamport balance to the `redeemer`.

The `issuer` and `redeemer` are separate on-chain roles (`config.issuer` / `config.redeemer`), each intended to be a distinct Utila MPC / Squads wallet under a two-party operational model. The rent leg of the bar lifecycle is therefore asymmetric: the lamports are paid out of the `issuer`'s wallet at issuance and collected by the `redeemer`'s wallet at redemption.

Whenever these two roles are held by different entities — which is the expected production configuration — every issue → redeem cycle moves the per-bar rent from the issuer to the redeemer.

programs/dgld/src/instructions/issue_bar.rs

Copy Code

```
55  #[account(
56          init,
57          payer = issuer,
58          space = 8 + Bar::INIT_SPACE,
59          seeds = [BAR_SEED, params.bar_id_hash.as_ref()],
60          bump,
61      )]
62      pub bar: Box<Account<'info, Bar>>,
63
64      #[account(mut)]
65      pub issuer: Signer<'info>,
```

programs/dgld/src/instructions/redeem_bar.rs

Copy Code

```
55  #[account(
56          mut,
57          seeds = [BAR_SEED, params.bar_id_hash.as_ref()],
58          bump = bar.bump,
59          constraint = bar.status == BarStatus::Active @ DgldError::BarNotActive,
60          // Free the Bar PDA on redemption so the same physical bar (serial) can be issued agai
61          // (issue_bar's `init` re-creates it). Rent is returned to the redeemer. While a bar i
62          // its PDA still exists, so double-issuing an *active* bar stays blocked by that `init
63          // redemption record survives off-account in the emitted `BarRedeemed` event.
64          close = redeemer,
65      )]
66      pub bar: Account<'info, Bar>,
67
68      /// Operator authorizing the redemption (role gate; must equal `config.redeemer`). NOTE: t
69      /// program does NOT enforce `redeemer != permanent_delegate` — role mutual-distinctness w
70      /// deliberately dropped (Utila MPC wallets may overlap). Keeping these two as distinct wa
71      /// for genuine two-party control over redemption is an OFF-CHAIN Utila policy, not an on-
72      /// guarantee. On-chain, the same key may satisfy both signers.
73      /// Marked `mut` because it receives the closed Bar PDA's rent (see `close = redeemer` abo
74      #[account(mut)]
75      pub redeemer: Signer<'info>,
```

Because the amount per bar is significantly small there is no supply, backing, or accounting-invariant impact, and no invariant surfaces the transfer.

The consequence is purely economic and operational: over many bars the issuer systematically subsidizes the redeemer's lamport balance, and the party that funded the storage is not the party made whole when it is released.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (0.8)

Recommendation

It is recommended to refund the closed `Bar` PDA rent to the account that originally funded it rather than to the `redeemer`.

Remediation Comment

ACKNOWLEDGED: The Gold Token SA team acknowledged this finding and stated that in GTSA's model `config.issuer` and `config.redeemer` are the same economic party (both GTSA-controlled wallets), so the rent is an internal float that nets to zero across each bar's issue-redeem cycle — a bookkeeping movement, not a cost or leak.

8.7 MISSING VALIDATION OF FINENESS AND ITS OMISSION FROM THE BARISSUED EVENT

// INFORMATIONAL

Description

`issue_bar` accepts a caller-supplied `fineness: u64` field that encodes the millesimal purity of the gold bar. The value is written verbatim into the `Bar` account — `bar.fineness = params.fineness;` — and the `Bar` state field is documented as an "opaque numeric, exactly as supplied".

However, two related weaknesses may affect this field:

1. **No range validation:** The handler performs no bounds check on `fineness`. Values such as `0`, `u64::MAX`, or any other physically meaningless quantity are accepted and persisted. A coherent domain constraint would be `0 < fineness <= 1_000_000` (0 to 1000.000 permille — purity cannot exceed 100%).
2. **Absence from the `BarIssued` event:** `fineness` is not among the fields emitted in `BarIssued` (events.rs:16-26), which carries `bar`, `bar_id`, `bar_id_hash`, `fine_weight`, `fee`, `to_recipient`, `recipient`, `total_active_fine_weight`, and `supply_after`. Consequently, the recorded purity is not propagated through the event stream consumed by indexers and off-chain UIs.

rograms/dgld/src/issue_bar.rs

Copy Code

```
let clock = Clock::get()?;
   let bar_key = ctx.accounts.bar.key();
   {
       let bar = &mut ctx.accounts.bar;
       bar.bump = ctx.bumps.bar;
       bar.status = BarStatus::Active;
       bar.fine_weight = w;
       bar.fineness = params.fineness;
```

Although this does not represent an exploitable security vulnerability — the impact is limited to the integrity of descriptive metadata surfaced to off-chain consumers, with no effect on funds or token supply — it is recommended to enforce the field's validity and propagate it through the event stream.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.5)

Recommendation

It is recommended to add a validate `fineness` in the handler, enforcing a sensible domain range (e.g. `0 < fineness <= 1_000_000`) and reject out-of-range values with a dedicated error, consistent with the validation already applied to `bar_id` and `terms_url`.

Consider also to include `fineness` in the `BarIssued` event, if it is convenient.

Remediation Comment

SOLVED: The `Gold Token SA` team solved this issue by:

- Adding a max fineness value as constant and a validation in `issue_bar` to enforce the range.
- Adding the `fineness` field in the `BarIssued` event emission.

Remediation Hash

d4ef76a31aff71bd302751941b57d735e00d6f18

8.8 MISSING VALIDATION IN ON ROLE/AUTHORITY SETTERS MAY RESULT IN NO-OP AND EMIT MISLEADING STATE-CHANGE EVENTS

// INFORMATIONAL

Description

The program's update instructions validate the incoming pubkey only against the default public key, but never against the value they are about to overwrite. As a result, an admin can submit the currently configured value and the instruction will succeed as a no-op while still emitting an event that signals a genuine governance transition.

Three instructions are affected:

- `initiate_admin_handoff` : `new_admin` is not checked against `config.admin` nor against the existing `config.pending_admin`. Passing `config.admin` sets `pending_admin` to the sitting admin; a subsequent `accept_admin_handoff` then performs `admin = admin` — a full two-step handoff cycle that transfers nothing. Passing the already-pending value re-arms the same candidate.
- `set_fee_wallet`: `new_fee_wallet` is not checked against `config.fee_wallet`. Passing the current wallet rewrites the same value.
- `rotate_authority`: `new_key` is not checked against the current `issuer` / `redeemer`.

programs/dgld/src/instructions/authorities.rs

Copy Code

```
    has_one = admin @ DgldError::Unauthorized,
    )]
    pub config: Account<'info, Config>,
    pub admin: Signer<'info>,
}

pub fn initiate_admin_handoff(ctx: Context<InitiateAdminHandoff>, new_admin: Pubkey) -> Result
    require!(new_admin != Pubkey::default(), DgldError::RoleIsDefault);
    let config = &mut ctx.accounts.config;
    let current = config.admin;
    config.pending_admin = new_admin;

emit!(AdminHandoffInitiated {
    current_admin: current,
    pending_admin: new_admin,
```

programs/dgld/src/instructions/fees.rs

Copy Code

```
pub fn set_fee_wallet(ctx: Context<SetFeeWallet>, new_fee_wallet: Pubkey) -> Result<()> {
    require!(
        new_fee_wallet != Pubkey::default(),
        DgldError::RoleIsDefault
    );
    let config = &mut ctx.accounts.config;
    let old = config.fee_wallet;
    config.fee_wallet = new_fee_wallet;
    emit!(FeeWalletChanged {
        old_fee_wallet: old,
        new_fee_wallet,
```

BVSS

AO:S/AC:L/AX:L/R:P/S:U/C:N/A:N/I:L/D:L/Y:N (0.3)

Recommendation

It is recommended to add an idempotency guard to each setter so a no-op reverts explicitly rather than emitting a spurious event. Additionally, consider to introduce a dedicated error (e.g. `NoStateChange`) in `errors.rs`.

Remediation Comment

SOLVED: The `Gold Token SA` team solved this issue by adding the suggested validation to prevent no-op.

Remediation Hash

504bae3984592d2b83cd0e3baa8e35351c8911c2
