Adevar Labs

Prepared for:
GTSA

July 22, 2026

DGLD Solana Audit Report

Version: after fix review

Table of Contents

1. Executive Summary

About DGLD

DGLD on Solana is a permissioned gold-backed tokenization protocol for issuing and redeeming DGLD tokens, where 1 DGLD = 1 troy ounce of LBMA-grade fine gold held in custody. Its core primitives are a Token-2022 mint with multiple extensions such as **PermanentDelegate**, a singleton **Config** account that stores operational roles and fee routing, and per-bar **Bar** accounts that track the lifecycle of each physical gold bar from issuance to redemption. The main components are initializing the config for protocol setup, issuing **DGLD** against new bars, redeeming **DGLD** for active bars, and recovering **DGLD** for restoring supply after burns. The intended users are the protocol operators managing custody and token supply, along with token holders and integrators who use DGLD as a Solana-native representation of vaulted physical gold.

Audit Summary

The table below summarizes identified vulnerabilities by risk level and remediation status.

Risk Level	Count	Fixed	Acknowledged
Critical	0	0	0
High	0	0	0
Medium	0	0	0
Low	0	0	0

Enhancement Opportunities: 4

Overall Security Posture

The GTSA team built a well-structured permissioned program with sound access control and correct use of Token-2022 extensions. The 4 enhancement opportunities target SBF stack-size overflow guards, cross-chain parity, long-term robustness, and stale account fields rather than active risks.

After Fix Review Summary

The fix review confirmed that 3 of the 4 enhancement opportunities have been resolved according to the recommendations. The remaining redemption fee item was acknowledged, with the team electing to document the Token-2022 transfer fee behavior operationally.

The GTSA team's prompt and precise implementation of the recommendations further demonstrates their commitment to code quality and security best practices.

Before Fix Review Summary

The audit identified 4 enhancement opportunities: raising **MAX_FEE_BIPS** to match the Ethereum reference contract's fee ceiling, making the redemption fee transfer robust to a future nonzero Token-2022 transfer fee, removing the unused **BarStatus** and **redeemed_at** state from the **Bar** account, and preventing Solana SBF stack size limit overflow. The stack-size issue can manifest in earlier versions of **build-sbf**, where large accounts cause **issue_bar** to fail. However, the build script enforces newer versions of **rustc** and **platform-tools** that optimize stack space usage and prevent overflow.

Launch Recommendations

After Fix Review Summary

The GTSA team implemented 3 of the recommended enhancements before launch. The redemption fee robustness recommendation was acknowledged and documented.

Before Fix Review Summary

I. Strongly recommended:

- Raise **MAX_FEE_BIPS** to 10,000 (100%) to match the Ethereum reference contract's fee ceiling, and re-run the **accounting.rs** proptest suite to confirm the new bound holds.
- Add a fee-wallet balance check in **redeem_bar** so the program reverts instead of silently under-collecting if a nonzero Token-2022 transfer fee is ever configured on the mint.
- Remove the unused **BarStatus** enum and **redeemed_at** field from the **Bar** account and correct the stale "never closed" comment in **state.rs**.
- Prevent future stack-size issues in **IssueBar** by boxing the **Config** and **Bar** accounts.

II. Post-Launch Monitoring:

- Monitor the Token-2022 mint's **TransferFeeConfig** authority, now held by the GTSA MPC wallet, and confirm redemption fee accounting stays accurate if the transfer fee is ever raised above 0 bps.
- Monitor the Utila MPC / Squads wallets for any unauthorized access, as these represent a critical trust assumption in the protocol.

Audit Scope

- **Repository:** https://github.com/goldtokensa/gtsa-solana
- **Before-Fix Review Commit Hash:** 4de54dbbf86df9dae4348c1957ea9340064f51a8
- **After-Fix Review Commit Hash:** 504bae3984592d2b83cd0e3baa8e35351c8911c2
- **Files/Modules in Scope:**
  - programs/*

2. Assumptions and Considerations

Audit Assumptions

Assumption 1: Users are expected to buy DGLD on secondary markets to be able to cover redemption fees.

Assumption 2: The redeem (**redeem_bar** instruction) path allows the **redeemer** (administrative role) to redeem users' DGLD tokens against any active **Bar**, provided they have enough tokens to cover the **Bar** weight and the redemption fees.

Assumption 3: During initialization, only the token decimals and supply are validated. No checks are performed to confirm the remaining expected properties of the DGLD Mint, which is assumed to be Freezable, Pausable, and to include the **PermanentDelegate**/**TransferFeeConfig** extensions.

Assumption 4: The mint will have all authorities migrated (including the mint authority to the **MINT_AUTHORITY_SEED** PDA) in a timely manner and will not inflate the token supply.

Assumption 5: The program will be built using **scripts/build.sh** that enforces **platform-tools** v1.54 (rust >= 1.85). Older versions may cause SBF stack size limit overflows.

Centralization Risks

Assumption 6: The protocol depends on trusted operators for both business logic and token-level controls. The main centralization risks are:

- Issuance is permissioned
- Redemption is non-consensual from the holder's perspective
- Transfer/Burn is non-consensual from the holder's perspective due to the **PermanentDelegate**
- Admin can reroute fees and rotate operators
- Recovery minting gives admin a controlled re-mint path
- Token-level authorities can alter transfer behavior or availability (freeze/pause)
- Role separation is not enforced on-chain, so multiple powers may be concentrated in one entity. For example, the **PermanentDelegate** can be the same as the **redeemer**.

3. Severity Definitions

Each issue identified in this report is assigned a severity level based on two dimensions: **Impact** and **Likelihood**. These dimensions help convey both the potential consequences of a vulnerability and how likely a vulnerability is to be discovered and exploited in the real world.

Note: Enhancements represent non-blocking improvements, typically usability, observability, or defense-in-depth tweaks that do not pose an immediate asset risk but would improve the product's reliability and/or user experience if implemented.

Impact

Impact reflects the potential consequences of the issue, particularly on **project funds**, **user funds**, and the **availability or integrity** of the protocol.

- **High Impact**: Successful exploitation could result in a complete loss of user or protocol funds, disruption of core protocol functionality, or permanent loss of control over critical components.
- **Medium Impact**: Exploitation could cause significant disruption or partial loss of funds, but not a total compromise. May impact some users or non-core functionality.
- **Low Impact**: The issue has minor or negligible consequences. It may affect edge cases, expose metadata, or degrade performance slightly without putting funds or core logic at serious risk.

Likelihood

Likelihood reflects how easy a vulnerability is to discover and exploit by an attacker, as well as how economically attractive the exploit is to an attacker.

- **High Likelihood**: The vulnerability is trivially exploitable. This means it can be exploited by a wide range of actors without privileged access rights, with minimal capital requirements and low financial risks.
- **Medium Likelihood**: This type of vulnerability can be found and exploited with moderate effort. It might require a significant capital investment, but with manageable financial risk.
- **Low Likelihood**: Exploitation of these vulnerabilities is often technically infeasible or requires highly specialized conditions. They may require extraordinary effort or a significant financial risk for an attacker, with a high chance of failure and minimal potential return.

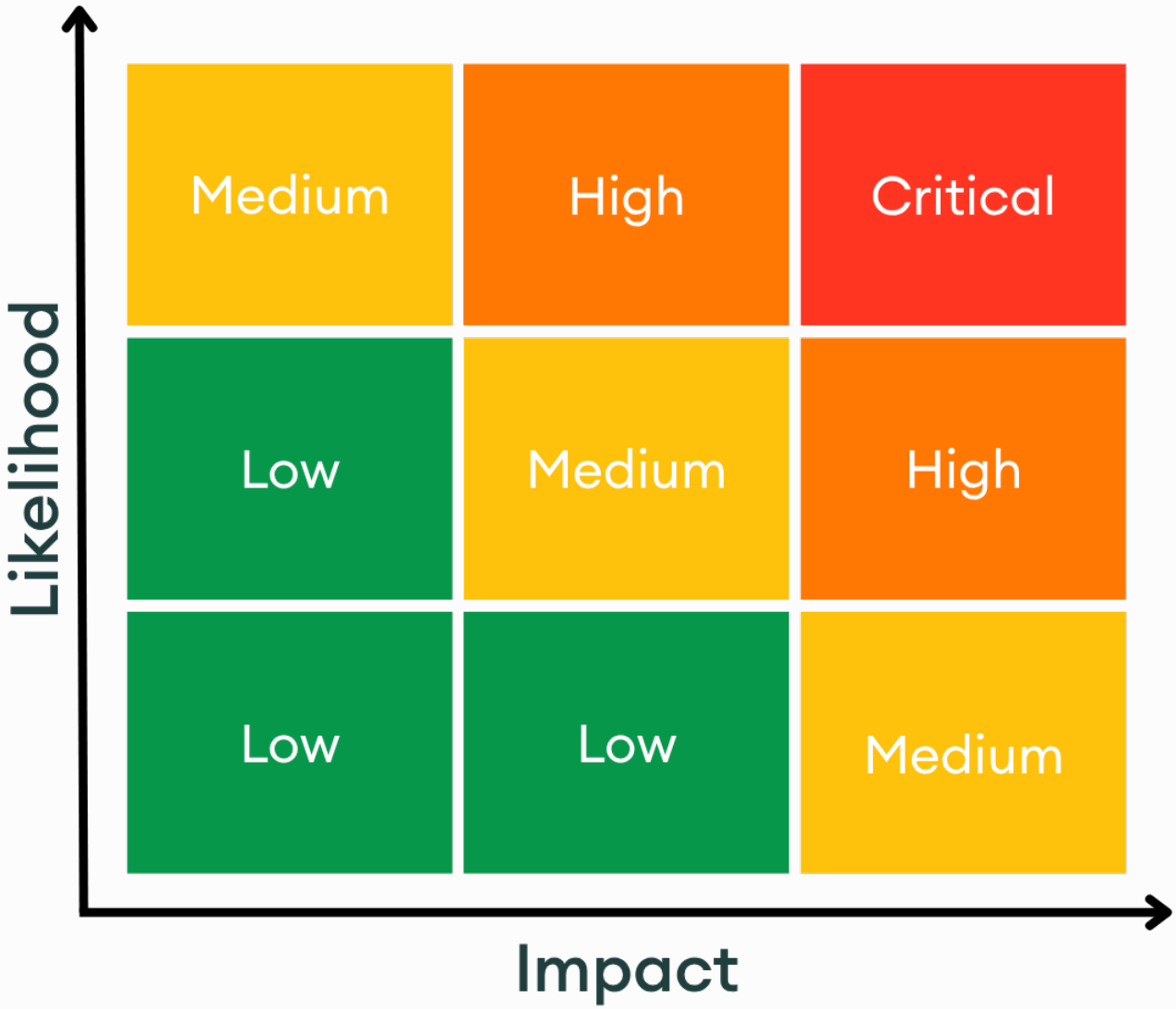
Severity Classification Matrix

By combining **Impact** and **Likelihood**, we assign a severity level using the matrix below:

- **Critical**: High impact + high likelihood (e.g. a bug that could allow anyone to drain a substantial amount of protocol funds with minimal effort)
- **High**: High impact with medium likelihood, or medium impact with high likelihood
- **Medium**: Moderate impact and/or discoverability

- **Low**: Minimal impact or unlikely to be exploited

This structured approach helps teams prioritize fixes and mitigate the most dangerous threats first.



Severity Matrix

4. Enhancement Opportunities

E01: Raise MAX_FEE_BIPS to match the Ethereum reference contract's 100% fee ceiling

LOCATION

gtsa-solana/4de54dbb/programs/dgld/src/accounting.rs:L16

RELEVANT SNIPPET

>_ accounting.rs — RUST

```rust
/// `ceil(amount * bips / 10_000)`. Rejects bips above the hard ceiling.
pub fn compute_fee(amount: u64, bips: u16) -> Result<u64> {
    require!((bips as u64) <= MAX_FEE_BIPS, DgldError::FeeTooHigh);
    let scaled = (amount as u128)
        .checked_mul(bips as u128)
```

DESCRIPTION

MAX_FEE_BIPS is hardcoded to 100 (1%), bounding the per-call issuance_fee_bips/redemption_fee_bips supplied by issuer/redeemer in issue_bar/redeem_bar via compute_fee. The reference Ethereum implementation (DGLDTokenMarch2026.sol) caps the equivalent creationFee/redemptionFee parameters at 10000 bips (100%) in mint()/_canBurn().

The client confirmed the 1% ceiling was an intentional, more conservative choice on the Solana side, but requested it be raised to match Ethereum's 100% cap for behavioral parity across chains, even though fees are not expected to approach that level in practice.

POTENTIAL BENEFIT

Aligns fee-rate flexibility across the Solana and Ethereum deployments and removes a hardcoded ceiling that would otherwise require a program upgrade to adjust later.

RECOMMENDATION

Raise MAX_FEE_BIPS from 100 to 10000, then re-run cargo test --lib to confirm the existing proptest bounds in accounting.rs (already parameterized on MAX_FEE_BIPS) hold at the new ceiling.

RUST

```rust
pub const MAX_FEE_BIPS: u64 = 10_000;
```

FIX REVIEW NOTES

Fixed according to the recommendation. MAX_FEE_BIPS was raised from 100 to 10_000.

E02: IssueBar can exceed the SBF stack size limit with older versions of build-sbf

LOCATION

gtsa-solana/4de54dbb/programs/dgld/src/instructions/issue_bar.rs:L14-L30

RELEVANT SNIPPET

>_ issue_bar.rs RUST

```rust
12:
13: #[derive(AnchorSerialize, AnchorDeserialize, Clone)]
14: pub struct IssueBarParams {
15:     /// Raw bar id; re-validated + re-canonicalized on-chain.
16:     pub bar_id: String,
17:     /// sha256(canonical(bar_id)); binds the Bar PDA. Verified on-chain to equal the c
    anonical hash.
18:     pub bar_id_hash: [u8; 32],
19:     pub fine_weight: u64,
20:     pub fineness: u64,
21:     pub partner_id: String,
22:     pub custodian: String,
23:     pub producer_id: String,
24:     pub certificate_of_deposit: String,
25:     pub inventory_report: String,
26:     pub recipient: Pubkey,
27:     /// Issuance fee rate (bips), supplied per-call. Carved OUT of `fine_weight`; vali
    dated against
28:     /// MAX_FEE_BIPS by `issue_split`/`compute_fee`.
29:     pub issuance_fee_bips: u16,
30: }
31:
32: #[derive(Accounts)]
```

DESCRIPTION

The **issue_bar** instruction is not executable on the Solana SBF target when built using older versions of **platform-tools** and **rustc** because the Anchor-generated **try_accounts** routine for **Context<IssueBar>** (currently sized at 4184 bytes) exceeds Solana's 4 KB stack limit. Newer versions include optimizations that reduce the stack usage below the limit, and this is enforced in **scripts/build.sh**. Therefore, the issue is classified as an enhancement rather than a vulnerability. The following describes the issue when built with older **build-sbf** versions.

IssueBar combines:
- a large instruction payload with multiple dynamically sized **String** fields
- **#[instruction(params: IssueBarParams)]**, which makes the full parameter object available to the generated account parser
- an init PDA whose seeds depend on **params.bar_id_hash**
- several Anchor account constraints and token interface account validations

As a result, the auto-generated account validation path becomes too large. During SBF compilation, the build emits a warning that **IssueBar::try_accounts** requires 4184 bytes of stack, exceeding the 4096-byte limit by 88 bytes. At runtime, calls to **issue_bar** then fail with an access violation before reaching the intended business logic and custom error handling. The core issuance path is effectively unavailable on-chain.

POTENTIAL BENEFIT

Prevents SBF stack size overflows that could result from future changes to **build-sbf** or from adding additional data to the existing accounts.

PROOF OF CONCEPT

Running **anchor test** on the codebase with an older version of **build-sbf** returns:
- Build warning: **IssueBar::try_accounts ... Stack offset of 4184 exceeded max offset of 4096**
- Runtime failure: **Access violation in stack frame ...** during Instruction: **IssueBar**

RECOMMENDATION

Box the large **Config** and **Bar** accounts. Boxing moves the deserialized payload to the BPF heap and leaves only an 8-byte pointer in the long-lived struct that flows through **try_accounts** → **Context** → **handler**. This will significantly reduce the data allocated to the stack and eliminate the stack frame size exceeded error.

DIFF

```
pub struct IssueBar<'info> {
    #[account(
        mut,
        seeds = [CONFIG_SEED],
        bump = config.config_bump,
        has_one = mint @ DgldError::MintMismatch,
        has_one = issuer @ DgldError::Unauthorized,
    )]
+    pub config: Box<Account<'info, Config>>,
-    pub config: Account<'info, Config>,
    #[account(
        mut,
        address = config.mint @ DgldError::MintMismatch,
        mint::token_program = token_program,
    )]
    pub mint: InterfaceAccount<'info, Mint>,

    /// CHECK: `[mint_authority]` PDA; signs the mint_to via seeds. Not deserialized.
    #[account(seeds = [MINT_AUTHORITY_SEED], bump = config.mint_authority_bump)]
    pub mint_authority: UncheckedAccount<'info>,

    #[account(
        init,
        payer = issuer,
        space = 8 + Bar::INIT_SPACE,
        seeds = [BAR_SEED, params.bar_id_hash.as_ref()],
        bump,
    )]
+    pub bar: Box<Account<'info, Bar>>,
-    pub bar: Account<'info, Bar>,
...
```

FIX REVIEW NOTES

Fixed according to the recommendation. The **config** and **bar** accounts in **IssueBar** are now **Box<Account<...>>**, moving their deserialized payloads to the heap and shrinking the **try_accounts** stack frame below the 4 KB limit.

E03: Redemption fee transfer is not compatible with Token-2022 transfer fee

LOCATION

gtsa-solana/4de54dbb/programs/dgld/src/instructions/redeem_bar.rs:L119-L135

RELEVANT SNIPPET

>_ redeem_bar.rs RUST

```rust
let holder = ctx.accounts.holder_token_account.owner;

// 1. Fee force-transfer holder -> fee wallet (supply-neutral), authorized by the perm
  anent
//    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,
    )?;
}

// 2. Standard burn_checked of EXACTLY W from the holder, authorized by the permanent
  delegate
```

DESCRIPTION

During **redeem_bar**, the protocol charges the holder a redemption fee by force-transferring **debit.fee** from the holder's token account to the protocol fee account with a standard Token-2022 **transfer_checked** CPI. It then emits **BarRedeemed.fee = debit.fee** and treats that value as the fee collected.

The **DGLD** mint carries the Token-2022 **TransferFeeConfig** extension, deployed at 0 bps (token/README.md:9). While the configured fee is 0, this code is exactly correct: the fee wallet is credited precisely **debit.fee** and the emitted event is accurate.

If a nonzero transfer fee were ever configured on the mint, Token-2022 would treat the **transfer_checked** amount as a gross amount and withhold its own fee from that leg. In that case the holder is still debited **debit.fee**, but the fee wallet's spendable balance increases by only **debit.fee - token2022_fee**; the remainder is moved into Token-2022 withheld-fee accounting. Because the handler does not snapshot and re-check the fee account's balance or withheld amount after the transfer (it reloads only the mint, for the supply invariant), this shortfall would be silent: **BarRedeemed.fee** would report the intended fee, not the amount actually credited to the fee wallet.

The transfer fee is mutable post-deployment. The runbook migrates the **TransferFeeConfig** authority to a GTSA MPC wallet rather than revoking it (token/README.md:44), so the fee can be raised above 0 bps after deployment without any change to DGLD program logic.

This issue is conditional on a trusted authority changing the mint's transfer fee from its deployed 0 bps to a nonzero value. The withheld amount does not disappear; it silently accrues in a separate account that is recoverable by a separate authority.

POTENTIAL BENEFIT

- The **BarRedeemed** event emits what the fee wallet actually received rather than what was intended
- The program reverts instead of silently under-collecting in the fee wallet
- Clear separation of fee types (transfer vs redeem)

RECOMMENDATION

Enforce that the fee wallet receives exactly **debit.fee**:

DIFF

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

+ let fee_balance_before = ctx.accounts.fee_token_account.amount;

    // 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,
        )?;

+       // The fee wallet MUST net exactly `debit.fee`. If a Token-2022 transfer fee withheld
   any
+       // part of it, the credited delta is short — revert instead of silently
  under-collecting
+       // and over-reporting `BarRedeemed.fee`.
+       ctx.accounts.fee_token_account.reload()?;
+       require!(
+           ctx.accounts.fee_token_account.amount
+               == fee_balance_before
+                   .checked_add(debit.fee)
+                   .ok_or(DgldError::MathOverflow)?,
+           DgldError::FeeUndercollected
+       );
    }
```

DEVELOPER RESPONSE

Acknowledged: *"GTSA as the token issuer is the sole recipient entity of both the transfer fee and the redemption fee, so any divergence between fee_charged and fee_received when a non-zero transfer fee is active nets to the same beneficiary and does not result in value leakage. The current design prioritizes keeping the redemption flow functional over reverting or introducing additional balance-delta reconciliation logic, which the team assessed as disproportionate to the informational severity and limited practical impact of the discrepancy."*

E04: Removing the unused BarStatus enum will simplify the Bar account

LOCATION

gtsa-solana/4de54dbb/programs/dgld/src/state.rs:L4-L7

RELEVANT SNIPPET

>_ state.rs RUST

```
2:
3: #[derive(AnchorSerialize, AnchorDeserialize, Clone, Copy, PartialEq, Eq, InitSpace, De
   bug)]
4: pub enum BarStatus {
5:     Active,
6:     Redeemed,
7: }
8:
9: /// Singleton program config. PDA `[b"config"]`.
```

OTHER LOCATIONS

- gtsa-solana/4de54dbb/programs/dgld/src/instructions/redeem_bar.rs:L55

DESCRIPTION

The **Bar** account carries a **status: BarStatus** field with two variants, **Active** and **Redeemed**. Tracing every access shows the field is effectively dead:
- **issue_bar** is the only place **status** is ever written, and it always sets **BarStatus::Active**.
- **BarStatus::Redeemed** is never assigned anywhere in the program.
- **redeem_bar** closes the **Bar** PDA (**close = redeemer**) instead of flipping its status.

Because a **Bar** account only exists while it is **Active** and is closed on redemption (then re-created as **Active** if the bar is re-issued), the field holds exactly one value for its entire lifetime.

Similarly, the **redeemed_at** field, which is set here, is only ever set to **0** at issuance. Since the **Bar** account is closed on redeem, it can probably also be removed.

>_ issue_bar.rs RUST

```
191:     bar.issued_at = clock.unix_timestamp;
192:     bar.issued_slot = clock.slot;
193:     bar.redeemed_at = 0;
194: }
195:
```

There is also a documentation contradiction: this comment describes the **Bar** as "never closed (permanent provenance + permanent dup-guard)," which conflicts with the implemented design.

>_ state.rs RUST

```
44: }
45:
46: /// One per physical bar. PDA `[b"bar", sha256(canonical_bar_id)]`. Created with `init
    `
47: /// (hard dup-guard) and never closed (permanent provenance + permanent dup-guard).
48: #[account]
49: #[derive(InitSpace)]
```

POTENTIAL BENEFIT

Make the code cleaner and aligned with the current implementation.

RECOMMENDATION

Remove the unused state and fix the stale comment.

FIX REVIEW NOTES

Fixed according to the recommendation. The **BarStatus** enum, the **Bar.status** field, and the **Bar.redeemed_at** field were removed.

About Us

About Adevar Labs

Adevar Labs is a boutique blockchain security firm specializing in web3 audits.

Built by a mix of experienced professionals in traditional enterprise and crypto natives who have contributed to some of the most critical projects in blockchain infrastructure.

Our team's background spans companies like Bitdefender, Asymmetric Research, Quantstamp, Chainproof, and Juicebox, and includes experience securing smart contracts, bridges, and L1 and L2 protocols across ecosystems like Solana, Ethereum, Polkadot, Cosmos, and MultiversX.

With over 100 audits completed and a portfolio that includes custom fuzzers, exploit modeling, and runtime testing frameworks, Adevar Labs brings both depth and precision to every engagement.

Our auditors have discovered critical vulnerabilities, built high-impact tooling, and placed in top positions in premier audit competitions including Code4rena and Sherlock.

Team members hold distinctions such as PhDs in software protection, and key roles at Fortune 500 companies, and leadership of flagship conferences like ETH Bucharest.

With team members having publications with over 1,100 academic citations and trusted by projects securing over $500M in on-chain value, Adevar Labs blends elite technical rigor with real-world security impact.

We also collaborate with some of the best independent security researchers in the web3 space.

Projects may optionally request specific contributors to be part of their audit.

Audit Methodology

Our audit methodology is specialized to provide thorough security assessments of Solana programs. We focus explicitly on rigorous manual analysis, detailed threat modeling, and careful validation of implemented fixes to ensure your programs operate securely and as intended.

1. Program Context and Architecture Analysis

Our auditors begin by deeply examining your Solana program's documentation, intended functionality, and account design. We meticulously map out program interactions, instruction processing flows, and state management logic. Special attention is given to understanding how your program interfaces with critical Solana system programs, such as the SPL Token Program, Stake Program, and System Program. Last but not least we check external integrations with other Solana projects and verify if the inputs and return values are handled properly.

2. Threat Modeling

We conduct targeted threat modeling tailored specifically for Solana's execution environment. We carefully define attacker capabilities and identify potential vulnerabilities that may arise from Solana-specific issues, including but not limited to:

- Unauthorized account data manipulation
- Improper ownership or signer verification
- Misuse of Program-Derived Addresses (PDAs)
- Incorrect use of Cross-Program Invocations (CPI)
- Failure to adequately handle account privileges or account states
- Risks stemming from rent-exemption and account initialization logic

3. In-depth Manual Security Review

Our experienced auditors perform an extensive manual security review of your Rust-based Solana programs. This involves a comprehensive line-by-line inspection of source code, focusing on common Solana vulnerabilities including, but not limited to:

- Missing or insufficient ownership checks
- Inadequate signer checks
- Incorrect handling of CPI calls and invocation privileges
- Arithmetic and integer overflow or underflow errors
- Unsafe deserialization and serialization of account data structures
- Improper token transfers and SPL-token logic issues
- Logic flaws in financial operations or state transitions
- Edge-case handling in instruction input validation
- Potential denial-of-service vectors related to transaction execution and account handling

During this stage, we clearly document any discovered vulnerabilities, including detailed descriptions, precise severity ratings, and recommendations for secure implementation. In situations where it is not clear how the vulnerability might be exploited we may also include a detailed proof-of-concept exploit including code snippets and instructions on how the exploit could be performed.

4. Detailed Fix Review and Validation

After the initial audit and your team's subsequent remediation efforts, we perform a comprehensive fix review to ensure the vulnerabilities identified have been effectively resolved.

We verify each fix individually, confirming that:

- Corrections effectively eliminate the security risks
- Changes do not inadvertently introduce new vulnerabilities or regressions
- The fixes align closely with Solana best practices and secure coding guidelines

Our detailed validation ensures that security improvements are robust, complete, and aligned with best practices specific to the Solana development ecosystem.

Confidentiality Notice

This report, including its content, data, and underlying methodologies, is subject to the confidentiality and feedback provisions in your agreement with Adevar Labs. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Adevar Labs.

Legal Disclaimer

The review and this report are provided by Adevar Labs on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Adevar Labs disclaims all warranties, expressed or implied, in connection with this report, its content, and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

You agree that access to and/or use of the report and other results of the review, including any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided.

You acknowledge that blockchain technology remains under development and is subject to unknown risks and flaws. Adevar Labs does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open-source or third-party software, code, libraries, materials, or information accessible through the report. As with the purchase or use of a product or service in any environment, you should use your best judgment and exercise caution where appropriate.