

Solana Foundation: Multi Delegator

Security Review

Cantina Managed review by:
Sujith Somraaj, Lead Security Researcher
Matías Barrios, Security Researcher

April 7, 2026

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
- 2.1 Scope . . . 3

3 Findings **5**
- 3.1 High Risk . . . 5
  - 3.1.1 Missing start time validation allows recurring transfers before delegation period begins 5
  - 3.1.2 Malicious merchant can drain subscriber funds by deleting and recreating a plan with an inflated per-period amount . . . 6
  - 3.1.3 Attacker can permanently block PDA creation by over-funding the address beyond the rent-exempt minimum . . . 9
  - 3.1.4 Malicious merchant can siphon subscriber funds indefinitely by deleting and recreating a plan with an extended `end_ts` . . . 11
- 3.2 Medium Risk . . . 14
  - 3.2.1 Fixed delegation creation rejects zero expiry despite being a valid ”no expiry” sentinel 14
  - 3.2.2 Recurring delegation creation rejects zero expiry despite being a valid ”no expiry” sentinel . . . 15
- 3.3 Low Risk . . . 16
  - 3.3.1 Cancellation sets `expires_at_ts` beyond plan's `end_ts`, trapping subscriber rent . 16
  - 3.3.2 Merchant can slash `end_ts` to near-immediate via `update_plan`, bypassing the period-length guard enforced at creation . . . 17
  - 3.3.3 Delegation created with `expiry_ts` in the past (within drift window) succeeds but is immediately expired and unusable . . . 19
- 3.4 Informational . . . 20
  - 3.4.1 PDA account creation propagates generic system program error instead of descriptive program error . . . 20
  - 3.4.2 Sponsor cannot independently reclaim rent from delegation accounts . . . 21
  - 3.4.3 Unchecked arithmetic in subscription transfer period length calculation . . . 22
  - 3.4.4 Re-initializing `MultiDelegate` revives orphaned delegation accounts . . . 22
  - 3.4.5 Test suite tests instructions in isolation . . . 23

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Mar 23rd to Mar 29th the Cantina team conducted a review of multi-delegator on commit hash 18a50bc2. The team identified a total of **14** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	4	4	0
Medium Risk	2	2	0
Low Risk	3	3	0
Gas Optimizations	0	0	0
Informational	5	5	0
Total	**14**	**14**	**0**

2.1 Scope

The security review had the following components in scope for multi-delegator on commit hash 18a50bc2:

```
programs/multi_delegator
├── build.rs
├── Cargo.toml
├── cu_report.md
└── src
    ├── constants.rs
    ├── errors.rs
    ├── event_engine.rs
    ├── lib.rs
    ├── events
    │   ├── fixed_transfer.rs
    │   ├── mod.rs
    │   ├── recurring_transfer.rs
    │   ├── subscription_cancelled.rs
    │   ├── subscription_created.rs
    │   └── subscription_transfer.rs
    ├── instructions
    │   ├── cancel_subscription.rs
    │   ├── close_multidelegate.rs
    │   ├── create_fixed_delegation.rs
    │   ├── create_plan.rs
    │   ├── create_recurring_delegation.rs
    │   ├── delete_plan.rs
    │   ├── emit_event.rs
    │   ├── initialize_multidelegate.rs
    │   ├── mod.rs
    │   ├── revoke_delegation.rs
    │   ├── subscribe.rs
    │   ├── transfer_fixed_delegation.rs
    │   ├── transfer_recurring_delegation.rs
    │   ├── transfer_subscription.rs
    │   ├── update_plan.rs
    │   └── helpers
    │       ├── delegation.rs
    │       ├── mod.rs
    │       ├── plan.rs
    │       ├── program.rs
    │       ├── system.rs
    │       ├── token.rs
    │       ├── traits.rs
```

```
│       ├── transfer_data.rs
│       ├── transfer_utils.rs
│       └── transfer_validation.rs
└── state
    ├── common.rs
    ├── fixed_delegation.rs
    ├── header.rs
    ├── mod.rs
    ├── multi_delegate.rs
    ├── plan.rs
    ├── recurring_delegation.rs
    ├── subscription_delegation.rs
    └── versioning
        ├── mod.rs
        └── v1_to_v2.rs
```

3 Findings

3.1 High Risk

3.1.1 Missing start time validation allows recurring transfers before delegation period begins

Severity: High Risk

Context: transfer_validation.rs#L61

Description: The `validate_recurring_transfer` function does not verify that the current timestamp is at or past the `current_period_start_ts` before allowing a transfer. The function computes `time_since_start` using `saturating_sub`, as shown in the code snippet below. When `current_ts` is less than `current_period_start_ts`, `saturating_sub` returns 0 instead of a negative value. This means the condition `time_since_start >= period_length` evaluates to false, skipping the period advancement logic entirely. Since `amount_pulled_in_period` starts at 0, the full `amount_per_period` budget is available for transfer immediately after delegation creation, regardless of the intended start time.

```
pub fn validate_recurring_transfer(
    transfer_amount: u64,
    amount_per_period: u64,
    period_length_s: u64,
    current_period_start_ts: &mut i64,
    amount_pulled_in_period: &mut u64,
    expiry_ts: i64,
    current_ts: i64,
) -> ProgramResult {
    ...
    let time_since_start = current_ts.saturating_sub(*current_period_start_ts);

    if time_since_start >= period_length {
        ...
    }

    let available = amount_per_period
        .checked_sub(*amount_pulled_in_period)
        .ok_or(MultiDelegatorError::ArithmeticUnderflow)?;
    if transfer_amount > available {
        return Err(MultiDelegatorError::AmountExceedsPeriodLimit.into());
    }
    ...
}
```

A delegatee can pull funds from the delegator's token account before the agreed-upon delegation start time. This affects both `transfer_recurring_delegation` and `transfer_subscription` instructions, as both rely on `validate_recurring_transfer` for period-based budget enforcement. For recurring delegations where `current_period_start_ts` is set to a future date, the full `amount_per_period` can be withdrawn immediately, bypassing the intended time-lock.

Proof of Concept: The following test creates a recurring delegation with `start_ts` set 1 day in the future, then executes a full `transfer_recurring_delegation` instruction at the current time. The transfer succeeds and tokens are moved to the delegatee's account before the delegation period has started.

```
    fn test_recurring_transfer_before_start_ts() {
        let amount_per_period: u64 = 50_000_000;
        let period_length_s: u64 = hours(1);
        // start_ts 1 day in the future.
        let start_ts: i64 = current_ts() + days(1) as i64;
        let expiry_ts: i64 = current_ts() + days(7) as i64;
        let nonce = 0;

        let (mut litesvm, alice, bob, delegation_pda, mint, _, bob_ata, _) =
            setup_recurring_delegation(
                amount_per_period,
                period_length_s,
```

```
            start_ts,
            expiry_ts,
            nonce,
        );

    assert_eq!(get_ata_balance(&litesvm, &bob_ata), 0);

    // Transfer BEFORE start_ts - current time is ~now, start_ts is +1 day.
    let transfer_amount: u64 = 10_000_000;
    let result =
        TransferDelegation::new(&mut litesvm, &bob, alice.pubkey(), mint,
        ↪  delegation_pda)
            .amount(transfer_amount)
            .recurring();

    // Bug: transfer succeeds even though the delegation period hasn't started.
    result.assert_ok();
    assert_eq!(
        get_ata_balance(&litesvm, &bob_ata),
        transfer_amount,
        "Delegatee was able to transfer before start_ts"
    );
}
```

Recommendation: It is recommended to add an explicit check that `current_ts` is at or past `current_period_start_ts` before proceeding with the transfer validation.

```
if current_ts < *current_period_start_ts {
    return Err(MultiDelegatorError::DelegationNotStarted.into());
}
```

Solana Foundation: Fixed in PR 6.

Cantina Managed: Verified fix.

3.1.2 Malicious merchant can drain subscriber funds by deleting and recreating a plan with an inflated per-period amount

Severity: High Risk

Context: delete_plan.rs#L59, subscribe.rs#L150-L152, transfer_subscription.rs#L60, transfer_subscription.rs#L75, subscription_delegation.rs#L23-L38

Description: The `transfer_subscription` instruction reads the per-period spending limit (`amount_per_period`) from the **current on-chain Plan account data** at the time of transfer, rather than from a value snapshotted into the `SubscriptionDelegation` at subscribe time:

```
// transfer_subscription.rs:60
amount_per_period = plan.data.amount;
```

The Plan PDA is derived deterministically from seeds `["plan", owner, plan_id]`. When a merchant deletes an expired plan and recreates it with the **same `plan_id`**, the new plan occupies the **exact same PDA address**. Existing `SubscriptionDelegation` accounts (which store the plan PDA as their `delegatee`) continue to reference the address, and all validation checks pass:

- **PDA address match** (`transfer_subscription.rs`): `subscription.header.delegatee == plan_pda.address()` passes because the address is unchanged.
- **Program ownership** (`transfer_subscription.rs`): The new plan is owned by the program.
- **Mint check** (`transfer_subscription.rs`): The merchant recreates the plan with the same mint.
- **Puller authorization** (`transfer_subscription.rs`): The merchant is still the plan owner.

The only field that changed is the `plan.data.amount` and is the one read without any cross-check against the subscriber's original agreement. The merchant sets `amount = u64::MAX` on the new plan and drains the subscriber's entire token balance in a single transfer.

Proof of Concept: The following test is added to `subscribe.rs` and passes on the current codebase, confirming the vulnerability:

```
#[test]
fn ghost_subscription_attack_via_plan_recreation() {
    let (mut litesvm, alice) = setup();
    let merchant = Keypair::new();
    litesvm.airdrop(&merchant.pubkey(), 10_000_000_000).unwrap();

    let mint = init_mint(
        &mut litesvm,
        TOKEN_PROGRAM_ID,
        MINT_DECIMALS,
        1_000_000_000,
        Some(alice.pubkey()),
        &[],
    );
    let alice_ata = init_ata(&mut litesvm, mint, alice.pubkey(), 100_000_000);
    let merchant_ata = init_ata(&mut litesvm, mint, merchant.pubkey(), 0);

    initialize_multidelegate_action(&mut litesvm, &alice, mint)
        .0
        .assert_ok();

    // Step 1: Merchant creates plan and Alice agrees to 1_000 tokens/period max.
    let original_amount = 1_000u64;
    let end_ts = current_ts() + days(2) as i64;
    let (res, plan_pda) = CreatePlan::new(&mut litesvm, &merchant, mint)
        .plan_id(1)
        .amount(original_amount)
        .period_hours(1)
        .end_ts(end_ts)
        .execute();
    res.assert_ok();

    // Step 2: Alice subscribes, trusting the 1_000/period limit.
    let (_, plan_bump) = get_plan_pda(&merchant.pubkey(), 1);
    Subscribe::new(
        &mut litesvm,
        &alice,
        merchant.pubkey(),
        plan_pda,
        1,
        plan_bump,
        mint,
    )
    .execute()
    .assert_ok();

    let (subscription_pda, _) = get_subscription_pda(&plan_pda, &alice.pubkey());

    // Step 3: Confirm the original limit IS enforced and pulling 1_001 fails.
    TransferSubscription::new(
        &mut litesvm,
        &merchant,
        alice.pubkey(),
        mint,
        subscription_pda,
        plan_pda,
    )
    .amount(original_amount + 1)
    .execute()
    .assert_err(MultiDelegatorError::AmountExceedsPeriodLimit);

    // Step 4: Time passes, plan expires. Merchant deletes it.
    move_clock_forward(&mut litesvm, days(3));
```

```
    DeletePlan::new(&mut litesvm, &merchant, plan_pda)
        .execute()
        .assert_ok();

    // Step 5: Merchant recreates plan with SAME plan_id but 50,000x the amount.
    let malicious_amount = 50_000_000u64;
    let new_end_ts = current_ts() + days(60) as i64;
    let (res, new_plan_pda) = CreatePlan::new(&mut litesvm, &merchant, mint)
        .plan_id(1)
        .amount(malicious_amount)
        .period_hours(1)
        .end_ts(new_end_ts)
        .execute();
    res.assert_ok();

    // The plan PDA is deterministic so same seeds produce same address.
    assert_eq!(plan_pda, new_plan_pda, "recreated plan must have same PDA");

    // Step 6: Merchant pulls 50_000_000 from Alice's old subscription.
    //         Alice only ever consented to 1_000/period.
    let alice_balance_before = get_ata_balance(&litesvm, &alice_ata);

    TransferSubscription::new(
        &mut litesvm,
        &merchant,
        alice.pubkey(),
        mint,
        subscription_pda,
        new_plan_pda,
    )
    .amount(malicious_amount)
    .execute()
    .assert_ok();

    // Attack succeeded: merchant stole 50,000x what Alice agreed to.
    assert_eq!(get_ata_balance(&litesvm, &merchant_ata), malicious_amount);
    assert_eq!(
        get_ata_balance(&litesvm, &alice_ata),
        alice_balance_before - malicious_amount,
    );
}
```

Test output:

```
test instructions::subscribe::tests::ghost_subscription_attack_via_plan_recreation ... ok
```

Likelihood: High: The attack requires a **malicious or compromised merchant** - an entity that already has a privileged role in the system. The barrier to exploitation is low.

Impact: High: Every un-revoked subscription to a deleted plan is exploitable. A single merchant with thousands of subscribers could drain all of them in one transaction block.

Recommendation: Any one of the following fixes eliminates the vulnerability. They can be combined for defense in depth.

1. Snapshot the agreed-upon amount into the subscription (recommended). Add an `amount_per_period` field to `SubscriptionDelegation`. Set it at subscribe time from the plan's current `amount`. In `transfer_subscription`, read the limit from the subscription, not the plan:

```
// subscribe.rs: at subscription init:
subscription.amount_per_period = plan.data.amount;

// transfer_subscription.rs: replace line 60:
amount_per_period = subscription.amount_per_period;
```

This decouples the subscriber's consent from the plan's mutable state. Even if the plan is recreated with a higher amount, old subscriptions remain bound to their original terms.

Trade-off: Increases `SubscriptionDelegation` size by 8 bytes (one `u64`). Existing subscriptions would need a migration or the transfer instruction would need to handle both layouts.

2. Prevent `plan_id` reuse. In `create_plan`, check that the plan PDA does not currently exist AND has never existed. The simplest approach is to add a monotonically increasing nonce to the Plan PDA seeds:

```
Seeds: ["plan", owner, plan_id, creation_nonce]
```

Alternatively, reject `create_plan` if a `SubscriptionDelegation` account referencing the PDA already exists though this is harder to enforce on-chain without an index.

Trade-off: Changes the PDA derivation, breaking client-side address computation. Requires a corresponding update to `subscribe`, `transfer_subscription`, and all off-chain tooling.

3. Invalidate ghost subscriptions at transfer time. In `transfer_subscription`, add a check that the subscription's `current_period_start_ts` is not older than the plan's creation timestamp. This requires adding a `created_at` field to `Plan`:

```
// transfer_subscription.rs: after loading both plan and subscription:
if subscription.current_period_start_ts < plan.created_at {
    return Err(MultiDelegatorError::SubscriptionStale.into());
}
```

Trade-off: Increases `Plan` size by 8 bytes. Legitimate long-running subscriptions that span a plan recreation would be correctly rejected.

Regardless of which fix is chosen, consider also:

- **Emitting an event when a plan is deleted**, so off-chain indexers can proactively notify subscribers to revoke.
- **Adding a `plan_generation` counter** to the Plan account that increments on recreation, so clients can detect plan identity changes.

Solana Foundation: Fixed in PR 4.

Cantina Managed: Verified fix. This finding shares the same root cause as finding "Malicious merchant can siphon subscriber funds indefinitely by deleting and recreating a plan with an extended `end_ts`". Both exploit the absence of term snapshotting in `SubscriptionDelegation`: the present finding inflates `amount` for an instant drain, "Malicious merchant can siphon subscriber funds indefinitely by deleting and recreating a plan with an extended `end_ts`" extends `end_ts` for a slow siphon.

The fix resolves both vectors with a single mechanism. The `PlanTerms` struct (`amount`, `period_hours`, `created_at`) is snapshotted into each subscription at subscribe time. At transfer time, `check_plan_terms()` compares the snapshot against the live plan. If a merchant has deleted and recreated the plan (even with the same `plan_id` and mint), the program-set `created_at` timestamp will differ, and the transfer is rejected with `PlanTermsMismatch`.

Of the three recommended fixes in the finding, the implemented approach corresponds to Option A (snapshot agreed terms) combined with elements of Option C (the `created_at` field acts as the staleness check).

3.1.3 Attacker can permanently block PDA creation by over-funding the address beyond the rent-exempt minimum

Severity: High Risk

Context: program.rs#L27

Description: `ProgramAccount::init` (`program.rs:27-74`) handles the case where a PDA has been pre-funded with lamports before the legitimate creator calls the instruction. When the account already has lamports, the code computes how much additional SOL the payer needs to transfer to reach the rent-exempt minimum:

```
// program.rs:47-49
let required_lamports = lamports                    // rent-exempt minimum
    .checked_sub(account.lamports())                // pre-funded amount
    .ok_or(MultiDelegatorError::ArithmeticUnderflow)?;  // FAILS if over-funded
```

If the pre-funded amount **exceeds** the rent-exempt minimum, `checked_sub` returns `None` and the instruction fails with `ArithmeticUnderflow`. Since the PDA address is deterministic and nobody holds its private key, the excess lamports can never be withdrawn. Every subsequent creation attempt hits the same `checked_sub` underflow, making the DoS **permanent**.

The existing test `initialize_multidelegate_with_prefunded_pda` only covers the under-funded case (1,000 lamports, below rent-exempt). The over-funded case was not tested.

Proof of Concept: The following test is added to `initialize_multidelegate.rs` and passes on the current codebase:

```
#[test]
fn overfunded_pda_blocks_creation_permanently() {
    use solana_account::Account;

    let (litesvm, user) = &mut setup();

    let mint = init_mint(
        litesvm,
        TOKEN_PROGRAM_ID,
        MINT_DECIMALS,
        1_000_000_000,
        Some(user.pubkey()),
        &[],
    );
    let _user_ata = init_ata(litesvm, mint, user.pubkey(), 1_000_000);

    let (multi_delegate_pda, _) =
        crate::tests::pda::get_multidelegate_pda(&user.pubkey(), &mint);

    // Attacker pre-funds the PDA with MORE than the rent-exempt minimum.
    // MultiDelegate::LEN = 66 bytes -> rent-exempt ~ 1,350,240 lamports.
    // Attacker sends 10,000,000 lamports (well above rent-exempt).
    litesvm
        .set_account(
            multi_delegate_pda,
            Account {
                lamports: 10_000_000,
                data: vec![],
                owner: solana_pubkey::Pubkey::default(),
                executable: false,
                rent_epoch: 0,
            },
        )
        .unwrap();

    // Legitimate user tries to initialize -- blocked by checked_sub
    // underflow in ProgramAccount::init (program.rs:47-49).
    let (res, _, _) = initialize_multidelegate_action(litesvm, user, mint);
    res.assert_err(MultiDelegatorError::ArithmeticUnderflow);

    // Account was NOT created -- attacker achieved permanent DoS.
    let account = litesvm.get_account(&multi_delegate_pda).unwrap();
    assert!(
        account.data.is_empty(),
        "MultiDelegate should not have been initialized"
    );
}
```

Test output:

```
test instructions::initialize_multidelegate::tests::overfunded_pda_blocks_creation_perma
↪  nently ...
↪  ok
```

Likelihood: Medium: The attacker only needs to transfer slightly more than the rent-exempt minimum (~0.002 SOL for most account sizes). This is economically negligible.

Impact: High: A user's `MultiDelegate` PDA has exactly one valid address per (user, mint) pair. If bricked, the user **cannot use the multi-delegator program at all** for that token. There is no alternative PDA, no nonce to rotate, and no recovery path. The user's ability to create any delegation (fixed, recurring, or subscription) for that mint is permanently destroyed.

Recommendation: Replace `checked_sub` with `saturating_sub` so that over-funded PDAs are handled gracefully.

Solana Foundation: Fixed in PR 5.

Cantina Managed: Verified fix.

3.1.4 Malicious merchant can siphon subscriber funds indefinitely by deleting and recreating a plan with an extended `end_ts`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: This is the converse of "Attacker can permanently block PDA creation by over-funding the address beyond the rent-exempt minimum". Where that finding inflates `amount` to drain funds instantly, this attack extends `end_ts` to siphon funds **perpetually** beyond the subscriber's agreed timeline.

The `transfer_subscription` instruction reads the plan's expiry timestamp from the **current on-chain Plan account** at transfer time:

```
// transfer_subscription.rs:48-51
plan_end_ts = plan.data.end_ts;
if plan_end_ts != 0 && current_ts > plan_end_ts {
    return Err(MultiDelegatorError::PlanExpired.into());
}
```

The `SubscriptionDelegation` does not store the `end_ts` the subscriber agreed to. When a merchant deletes an expired plan and recreates it with the same `plan_id` but a later `end_ts`, the expiry check reads the new value. Transfers that were correctly blocked by `PlanExpired` now succeed again.

The merchant doesn't need to change the amount or period and the same modest pull rate continues operating, but for months or years beyond what the subscriber consented to. This makes the attack harder to detect than "Attacker can permanently block PDA creation by over-funding the address beyond the rent-exempt minimum" which causes an instant drain: from the subscriber's perspective, the same small amounts keep leaving their account long after the subscription should have ended.

Proof of Concept: The following test is added to `subscribe.rs` and passes on the current codebase:

```
#[test]
fn ghost_subscription_attack_extended_timeline() {
    let (mut litesvm, alice) = setup();
    let merchant = Keypair::new();
    litesvm.airdrop(&merchant.pubkey(), 10_000_000_000).unwrap();

    let mint = init_mint(
        &mut litesvm,
        TOKEN_PROGRAM_ID,
        MINT_DECIMALS,
        1_000_000_000,
        Some(alice.pubkey()),
        &[],
    );
    let alice_ata = init_ata(&mut litesvm, mint, alice.pubkey(), 100_000_000);
    let merchant_ata = init_ata(&mut litesvm, mint, merchant.pubkey(), 0);
```

```
initialize_multidelegate_action(&mut litesvm, &alice, mint)
    .0
    .assert_ok();

// Step 1: Plan with modest amount, short lifetime (2 days).
let amount = 1_000u64;
let original_end_ts = current_ts() + days(2) as i64;
let (res, plan_pda) = CreatePlan::new(&mut litesvm, &merchant, mint)
    .plan_id(1)
    .amount(amount)
    .period_hours(1)
    .end_ts(original_end_ts)
    .execute();
res.assert_ok();

// Step 2: Alice subscribes — she agrees to 1_000/hr for 2 days.
let (_, plan_bump) = get_plan_pda(&merchant.pubkey(), 1);
Subscribe::new(
    &mut litesvm,
    &alice,
    merchant.pubkey(),
    plan_pda,
    1,
    plan_bump,
    mint,
)
.execute()
.assert_ok();

let (subscription_pda, _) = get_subscription_pda(&plan_pda, &alice.pubkey());

// Step 3: Merchant pulls legitimately in the first period.
TransferSubscription::new(
    &mut litesvm,
    &merchant,
    alice.pubkey(),
    mint,
    subscription_pda,
    plan_pda,
)
.amount(amount)
.execute()
.assert_ok();

assert_eq!(get_ata_balance(&litesvm, &merchant_ata), amount);

// Step 4: Plan expires. Transfers are blocked.
move_clock_forward(&mut litesvm, days(3));

TransferSubscription::new(
    &mut litesvm,
    &merchant,
    alice.pubkey(),
    mint,
    subscription_pda,
    plan_pda,
)
.amount(amount)
.execute()
.assert_err(MultiDelegatorError::PlanExpired);

// Step 5: Merchant deletes expired plan, recreates with same
//         plan_id and amount but end_ts extended by a year.
DeletePlan::new(&mut litesvm, &merchant, plan_pda)
    .execute()
```

```
        .assert_ok();

    let extended_end_ts = current_ts() + days(365) as i64;
    let (res, new_plan_pda) = CreatePlan::new(&mut litesvm, &merchant, mint)
        .plan_id(1)
        .amount(amount)
        .period_hours(1)
        .end_ts(extended_end_ts)
        .execute();
    res.assert_ok();
    assert_eq!(plan_pda, new_plan_pda);

    // Step 6: Merchant resumes pulling — Alice's 2-day agreement is
    //         now silently extended to 365+ days.
    let balance_before = get_ata_balance(&litesvm, &alice_ata);

    TransferSubscription::new(
        &mut litesvm,
        &merchant,
        alice.pubkey(),
        mint,
        subscription_pda,
        new_plan_pda,
    )
    .amount(amount)
    .execute()
    .assert_ok();

    assert_eq!(
        get_ata_balance(&litesvm, &alice_ata),
        balance_before - amount,
        "Merchant siphoned tokens beyond the original 2-day agreement",
    );

    // Step 7: Advance another 30 days — still pulling from Alice.
    move_clock_forward(&mut litesvm, days(30));

    TransferSubscription::new(
        &mut litesvm,
        &merchant,
        alice.pubkey(),
        mint,
        subscription_pda,
        new_plan_pda,
    )
    .amount(amount)
    .execute()
    .assert_ok();

    // Merchant has now pulled well beyond Alice's original 2-day window.
    assert!(
        get_ata_balance(&litesvm, &merchant_ata) > amount,
        "Merchant continues extracting tokens months after the original plan expired",
    );
}
```

Test output: `test instructions::subscribe::tests::ghost_subscription_attack_extended_-timeline ... ok.`

Recommendation: The same fixes from "Attacker can permanently block PDA creation by over-funding the address beyond the rent-exempt minimum" address this issue as both share the same root cause. The recommended fix (Option A from the fix suggested) snapshots `amount_per_period`, `period_hours`, **and** `end_ts` into the `SubscriptionDelegation` at subscribe time:

```
// subscribe.rs — at subscription init:
subscription.plan_end_ts = plan.data.end_ts;
```

```
// transfer_subscription.rs – replace line 48:
plan_end_ts = subscription.plan_end_ts;
```

With the expiry stored in the subscription, a recreated plan with an extended `end_ts` has no effect on existing subscriptions as they expire according to the original terms.

Solana Foundation: Fixed in PR 4.

Cantina Managed: The fix correctly mitigates the delete-and-recreate attack vector by snapshotting `PlanTerms` (including the program-controlled `created_at` fingerprint) into each `SubscriptionDelegation` at subscribe time. Transfers and cancellations validate the snapshot against the live plan via `check_plan_terms()`, blocking ghost plans.

The parameter `end_ts` is intentionally not snapshotted and remains read from the live Plan account. This means merchants can extend a plan's lifetime via `update_plan` without triggering `PlanTermsMismatch`. This is expected behavior and subscribers retain the ability to cancel at any time if they disagree with the updated timeline. The ghost-account vector (delete + recreate with extended `end_ts`) is fully blocked by the `created_at` fingerprint, which differs for every plan lifecycle.

3.2 Medium Risk

3.2.1 Fixed delegation creation rejects zero expiry despite being a valid ”no expiry” sentinel

Severity: Medium Risk

Context: create_fixed_delegation.rs#L28-L30

Description: The `CreateFixedDelegationData::validate()` method in `create_fixed_delegation.rs` does not account for the `expiry_ts = 0` sentinel value that represents a delegation with no expiry:

```
if self.expiry_ts < current_time.saturating_sub(TIME_DRIFT_ALLOWED_SECS) {
    return Err(MultiDelegatorError::FixedDelegationExpiryInPast);
}
```

The transfer-time validation in `transfer_validation.rs` correctly treats 0 as ”never expires”:

```
if expiry_ts != 0 && current_ts > expiry_ts {
    return Err(MultiDelegatorError::DelegationExpired.into());
}
```

This inconsistency means that on mainnet, any attempt to create a fixed delegation with `expiry_ts = 0` will always fail since `0 < current_time - 120` will always be true. Due to this, users will not be able to create permanent (non-expiring) fixed delegations in production, which is a supported and documented feature of the protocol.

Proof of Concept: The following test case demonstrates the issue:

```
#[test]
fn create_fixed_delegation_with_no_expiry() {
  let (litesvm, user) = &mut setup();
  let payer = user;
  let amount: u64 = 100_000_000;
  let expiry_ts: i64 = 0;
  let nonce: u64 = 0;

  let mint = init_mint(
     litesvm,
     TOKEN_PROGRAM_ID,
      MINT_DECIMALS,
      1_000_000_000,
      Some(payer.pubkey()),
      &[],
   );
   let _user_ata = init_ata(litesvm, mint, payer.pubkey(), 1_000_000);
```

```
    initialize_multidelegate_action(litesvm, payer, mint).0.assert_ok();

    let delegatee = Pubkey::new_unique();

    let (res, delegation_pda) = CreateDelegation::new(litesvm, payer, mint,
    ↪  delegatee).nonce(nonce).fixed(amount, expiry_ts);
     res.assert_err(MultiDelegatorError::FixedDelegationExpiryInPast);
}
```

Recommendation: Add an explicit check for the zero sentinel before the expiry-in-past validation:

```
if self.expiry_ts != 0 && self.expiry_ts <
↪  current_time.saturating_sub(TIME_DRIFT_ALLOWED_SECS) {
    return Err(MultiDelegatorError::FixedDelegationExpiryInPast);
}
```

Solana Foundation: Fixed in PR 8.

Cantina Managed: Verified fix.

3.2.2 Recurring delegation creation rejects zero expiry despite being a valid "no expiry" sentinel

Severity: Medium Risk

Context: create_recurring_delegation.rs#L55-L57

Description: The `CreateRecurringDelegationData::validate()` method in `create_recurring_delegation.rs` unconditionally rejects `expiry_ts = 0`:

```
if self.start_ts >= self.expiry_ts {
   return Err(MultiDelegatorError::RecurringDelegationStartTimeGreaterThanExpiry);
}
```

On mainnet, `start_ts` is a real Unix timestamp. When `expiry_ts = 0`, the condition `start_ts >= 0` is always true, so creation is rejected. However, the transfer-time validation in `transfer_validation.rs` explicitly treats 0 as "never expires":

```
if expiry_ts != 0 && current_ts > expiry_ts {
    return Err(MultiDelegatorError::DelegationExpired.into());
}
```

This is the same class of inconsistency as the fixed delegation zero-expiry issue. The creation path rejects the sentinel value that the transfer path is designed to accept. Users cannot create perpetual recurring delegations.

Proof of Concept: The following test demonstrates the existence of the issue:

```
#[test]
fn create_recurring_delegation_with_zero_expiry() {
   let (litesvm, user) = &mut setup();
   let payer = user;
   let amount_per_period: u64 = 50_000_000;
   let period_length_s: u64 = 86400;
   let start_ts: i64 = current_ts();
   let expiry_ts: i64 = 0; // No expiry sentinel
   let nonce: u64 = 0;

   let mint = init_mint(
      litesvm,
      TOKEN_PROGRAM_ID,
       MINT_DECIMALS,
       1_000_000_000,
        Some(payer.pubkey()),
        &[],
    );
    let _user_ata = init_ata(litesvm, mint, payer.pubkey(), 1_000_000);
```

```
    initialize_multidelegate_action(litesvm, payer, mint).0.assert_ok();

    let delegatee = Pubkey::new_unique();

    let (res, _delegation_pda) = CreateDelegation::new(litesvm, payer, mint, delegatee)
       .nonce(nonce)
       .recurring(amount_per_period, period_length_s, start_ts, expiry_ts);

    res.assert_err(MultiDelegatorError::RecurringDelegationStartTimeGreaterThanExpiry);
}
```

Recommendation: Add a carve-out for the zero sentinel before the ordering check:

```
if self.expiry_ts != 0 && self.start_ts >= self.expiry_ts {
    return Err(MultiDelegatorError::RecurringDelegationStartTimeGreaterThanExpiry);
}
```

Additionally, add a test that creates a recurring delegation with `expiry_ts = 0` using a realistic mainnet clock and verifies both creation and transfer succeed.

Solana Foundation: Fixed in PR 7.

Cantina Managed: Verified fix.

3.3 Low Risk

3.3.1 Cancellation sets `expires_at_ts` beyond plan's `end_ts`, trapping subscriber rent

Severity: Low Risk

Context: cancel_subscription.rs#L56, cancel_subscription.rs#L63-L67

Description: When a subscriber cancels, `cancel_subscription.rs` computes `expires_at_ts` as the end of the current billing period:

```
// cancel_subscription.rs:63-67
expires_at_ts = periods_elapsed
    .checked_add(1)
    .and_then(|p| p.checked_mul(period_length_s))
    .and_then(|offset| period_start.checked_add(offset))
    .ok_or::<ProgramError>(MultiDelegatorError::ArithmeticOverflow.into())?;
```

This value is never capped at `plan.data.end_ts`. When the current billing period extends past the plan's expiry, `expires_at_ts` overshoots `end_ts`, creating a **dead zone** where the subscriber's rent (~0.002 SOL) is locked in the `SubscriptionDelegation` PDA until `expires_at_ts` is finally reached.

Recommendation: Cap `expires_at_ts` at `plan.data.end_ts` when the plan has a finite end date. Add one line after the period-boundary computation:

```
// cancel_subscription.rs - AFTER line 67 (the existing computation)
// Cap at plan end_ts so the subscriber can revoke as soon as the plan expires.
if plan.data.end_ts != 0 && expires_at_ts > plan.data.end_ts {
    expires_at_ts = plan.data.end_ts;
}
```

This ensures `expires_at_ts <= plan.data.end_ts` whenever the plan has a finite lifetime. The subscriber can then revoke immediately once the plan expires, with no dead zone. The fix preserves the existing behavior for plans with `end_ts = 0` (no expiry) and for cancellations where the period end falls before the plan end.

Solana Foundation: Fixed in PR 10.

Cantina Managed: Verified fix. The fix adds a `.min(plan.data.end_ts)` cap to the `expires_at_ts` computation inside the period-boundary calculation chain, gated behind `plan.data.end_ts != 0` to preserve no-expiry behavior.

3.3.2 Merchant can slash `end_ts` to near-immediate via `update_plan`, bypassing the period-length guard enforced at creation

Severity: Low Risk

Context: update_plan.rs#L43-L50

Description: `create_plan` enforces that `end_ts` must be at least one full billing period into the future, ensuring subscribers get at least one complete period of service:

```
// create_plan.rs:73-77 (PlanData::validate)
if self.end_ts != 0 {
    let period_secs = (self.period_hours as i64) * 3600;
    if current_time + period_secs > self.end_ts {
        return Err(MultiDelegatorError::InvalidEndTs);
    }
}
```

`update_plan` validates `end_ts` with a weaker check that only requires the timestamp to be in the future:

```
// update_plan.rs:43-49 (UpdatePlanData::validate)
pub fn validate(&self, current_time: i64) -> Result<(), MultiDelegatorError> {
    PlanStatus::try_from(self.status).map_err(|_|
    ↪  MultiDelegatorError::InvalidPlanStatus)?;
    if self.end_ts != 0 && self.end_ts <= current_time {
        return Err(MultiDelegatorError::InvalidEndTs);
    }
    Ok(())
}
```

There is **no check** that `end_ts >= current_time + period_hours * 3600`. A merchant with a 720-hour (30-day) plan can call `update_plan` with `end_ts = current_ts + 1`, instantly expiring the plan on the next block.

Proof of Concept: The following test is added to `update_plan.rs` and passes on the current codebase:

```
#[test]
fn update_plan_end_ts_bypasses_period_guard() {
    let (mut litesvm, owner) = setup();
    let merchant = &owner;
    let alice = Keypair::new();
    litesvm.airdrop(&alice.pubkey(), 10_000_000_000).unwrap();

    let mint = init_mint(
        &mut litesvm,
        TOKEN_PROGRAM_ID,
        MINT_DECIMALS,
        1_000_000_000,
        Some(alice.pubkey()),
        &[],
    );
    let _alice_ata = init_ata(&mut litesvm, mint, alice.pubkey(), 100_000_000);
    let _merchant_ata = init_ata(&mut litesvm, mint, merchant.pubkey(), 0);

    initialize_multidelegate_action(&mut litesvm, &alice, mint)
        .0
        .assert_ok();

    // Create plan: 720-hour (30-day) periods, no end date.
    let (res, plan_pda) = CreatePlan::new(&mut litesvm, merchant, mint)
        .plan_id(1)
        .amount(1_000)
        .period_hours(720)
        .execute();
    res.assert_ok();

    // Alice subscribes, trusting the 30-day period.
```

```
let (_, plan_bump) = get_plan_pda(&merchant.pubkey(), 1);
Subscribe::new(
    &mut litesvm,
    &alice,
    merchant.pubkey(),
    plan_pda,
    1,
    plan_bump,
    mint,
)
.execute()
.assert_ok();

let (subscription_pda, _) = get_subscription_pda(&plan_pda, &alice.pubkey());

// Merchant pulls within the agreed limit — works fine.
TransferSubscription::new(
    &mut litesvm,
    merchant,
    alice.pubkey(),
    mint,
    subscription_pda,
    plan_pda,
)
.amount(500)
.execute()
.assert_ok();

// ---- THE ATTACK ----
// create_plan would reject end_ts = current_ts + 1 for a 720h plan.
// update_plan only checks end_ts > current_time — no period guard.
let near_immediate_end = current_ts() + 2;
let res = UpdatePlan::new(&mut litesvm, merchant, plan_pda)
    .end_ts(near_immediate_end)
    .execute();
res.assert_ok(); // <-- SHOULD be rejected but isn't.

// Verify plan expires almost immediately.
let account = litesvm.get_account(&plan_pda).unwrap();
let plan = Plan::load(&account.data).unwrap();
let plan_end = plan.data.end_ts;
let period_secs = plan.data.period_hours as i64 * 3600;
assert!(
    plan_end < current_ts() + period_secs,
    "end_ts is less than one period into the future",
);

// Advance past the slashed end_ts.
move_clock_forward(&mut litesvm, 3);

// Transfers fail — plan expired.
TransferSubscription::new(
    &mut litesvm,
    merchant,
    alice.pubkey(),
    mint,
    subscription_pda,
    plan_pda,
)
.amount(100)
.execute()
.assert_err(MultiDelegatorError::PlanExpired);

// Alice cancels. expires_at_ts is based on the 30-day period.
CancelSubscription::new(&mut litesvm, &alice, plan_pda, subscription_pda)
    .execute()
```

```
        .assert_ok();

    // Rent trap: expires_at_ts is ~30 days out despite the plan
    // having expired seconds ago.
    let sub_account = litesvm.get_account(&subscription_pda).unwrap();
    let sub = SubscriptionDelegation::load(&sub_account.data).unwrap();
    let expires_at = sub.expires_at_ts;
    let gap = expires_at - near_immediate_end;
    assert!(gap > days(29) as i64);

    // Revoke fails — subscriber is stuck for ~30 days.
    RevokeSubscription::new(&mut litesvm, &alice, subscription_pda)
        .execute()
        .assert_err(MultiDelegatorError::SubscriptionNotCancelled);
}
```

Recommendation: Apply the same period-length guard from `create_plan` to `update_plan`.

Solana Foundation: Fixed in PR 9.

Cantina Managed: Verified fix. The fix extracts the one-period-ahead guard from `create_plan` into a shared `validate_plan_end_ts()` function in `state/common.rs` and applies it in both `create_plan` and `update_plan`. This eliminates the inconsistency.

3.3.3 Delegation created with `expiry_ts` in the past (within drift window) succeeds but is immediately expired and unusable

Severity: Low Risk

Context: transfer_validation.rs#L20-L22, transfer_validation.rs#L51-L53

Description: There is an asymmetry between how `expiry_ts` is validated at delegation creation time versus transfer time.

Creation (`create_fixed_delegation.rs`) applies a `TIME_DRIFT_ALLOWED_SECS` (120s) tolerance when checking if the expiry is in the past:

```
if self.expiry_ts != 0 && self.expiry_ts <
↪   current_time.saturating_sub(TIME_DRIFT_ALLOWED_SECS) {
    return Err(MultiDelegatorError::FixedDelegationExpiryInPast);
}
```

This allows an `expiry_ts` up to 120 seconds behind the current clock to pass validation.

Transfer (`transfer_validation.rs`) performs a strict comparison with **no** drift tolerance:

```
if expiry_ts != 0 && current_ts > expiry_ts {
    return Err(MultiDelegatorError::DelegationExpired);
}
```

This means a delegation created with `expiry_ts = now - 60s` will:

1. **Pass** creation validation (60s < 120s drift window).
2. **Immediately fail** every transfer attempt (`now > now - 60s`).

The delegation PDA is allocated and rent is paid, but the delegation is dead on arrival and it can never be used for a transfer. The same asymmetry exists for `RecurringDelegation` via `create_recurring_delegation.rs:47` which applies the same drift tolerance to `start_ts`.

Recommendation: Apply the `TIME_DRIFT_ALLOWED_SECS` tolerance consistently on both the creation and transfer sides. The drift window exists to account for cross-validator clock skew and the same reasoning applies at transfer time. Update the expiry check in `transfer_validation.rs` to match the tolerance already used at creation:

```
// transfer_validation.rs — apply the same drift tolerance used at creation
if expiry_ts != 0 && current_ts > expiry_ts.saturating_add(TIME_DRIFT_ALLOWED_SECS) {
```

```
    return Err(MultiDelegatorError::DelegationExpired);
}
```

The same change should be applied to the recurring transfer expiry check in `validate_recurring_transfer`.

Solana Foundation: Fixed in PR 13.

Cantina Managed: Verified fix. The fix applies `TIME_DRIFT_ALLOWED_SECS` (120s) tolerance symmetrically to both the creation and transfer paths.

3.4 Informational

3.4.1 PDA account creation propagates generic system program error instead of descriptive program error

Severity: Informational

Context: delegation.rs#L98, plan.rs#L66

Description: The `ProgramAccount::init` function is used across multiple instructions to create PDA accounts via CPI to the System Program. When the PDA already exists (e.g., duplicate `nonce` for `delegations` or duplicate `plan_id` for `plans`), the System Program returns a generic `Custom(0)` error ("Allocate: account already in use").

This raw error is propagated directly to callers without any descriptive context, as shown in the code snippets below.

In `create_delegation_account` (used by `create_fixed_delegation` and `create_recurring_delegation`):

```
ProgramAccount::init::<()>(accounts.payer, accounts.delegation_account, &seeds, space)?;
```

In `create_plan_account` (used by `create_plan`):

```
ProgramAccount::init::<()>(accounts.merchant, accounts.plan_pda, &seeds, Plan::LEN)?;
```

A pre-check on the target account's data length before the CPI would allow returning a descriptive program-specific error.

Proof of Concept: The following integration test creates a fixed delegation with nonce 0, then attempts to create another fixed delegation with the same nonce. The second creation fails with a generic System Program error instead of a descriptive program error.

```
#[test]
fn duplicate_nonce_error_is_not_descriptive() {
    let (litesvm, user) = &mut setup();
    let payer = user;

    let mint = init_mint(
        litesvm,
        TOKEN_PROGRAM_ID,
        MINT_DECIMALS,
        1_000_000_000,
        Some(payer.pubkey()),
        &[],
    );
    let _user_ata = init_ata(litesvm, mint, payer.pubkey(), 1_000_000);

    initialize_multidelegate_action(litesvm, payer, mint)
        .0
        .assert_ok();

    let delegatee = Pubkey::new_unique();

    // First creation succeeds.
    let (res, _) = CreateDelegation::new(litesvm, payer, mint, delegatee)
```

```
        .nonce(0)
        .fixed(100, current_ts() + 1000);
    res.assert_ok();

    // Second creation with same nonce fails.
    let (res2, _) = CreateDelegation::new(litesvm, payer, mint, delegatee)
        .nonce(0)
        .fixed(200, current_ts() + 2000);

    match res2 {
        Ok(_) => panic!("Expected duplicate nonce to fail"),
        Err(failed_tx) => {
            // The error is NOT a descriptive MultiDelegatorError,
            // it's a generic System Program error (Allocate on existing account).
            println!("Error: {:?}", failed_tx.err);
        }
    }
}
```

Recommendation: It is recommended to add a pre-check before each `ProgramAccount::init` call to verify that the target account does not already exist, and return a descriptive error if it does. For delegations in `delegation.rs`:

```
if accounts.delegation_account.data_len() > 0 {
    return Err(MultiDelegatorError::DelegationAlreadyExists.into());
}

ProgramAccount::init::<()>(accounts.payer, accounts.delegation_account, &seeds, space)?;
```

For plans in `plan.rs`:

```
if accounts.plan_pda.data_len() > 0 {
    return Err(MultiDelegatorError::PlanAlreadyExists.into());
}

ProgramAccount::init::<()>(accounts.merchant, accounts.plan_pda, &seeds, Plan::LEN)?;
```

Solana Foundation: Fixed in PR 12.

Cantina Managed: Verified fix.

3.4.2 Sponsor cannot independently reclaim rent from delegation accounts

Severity: Informational

Context: revoke_delegation.rs#L54, revoke_delegation.rs#L92

Description: When a third-party sponsor funds the creation of a delegation account (fixed or recurring), the `payer` field in the delegation header is set to the sponsor's address. On revocation, `resolve_destination()` in `revoke_delegation.rs` correctly routes the rent refund back to the sponsor rather than the delegator:

```
if payer_bytes == accounts.authority.address().as_ref() {
   Ok(accounts.authority)
} else {
   let receiver = accounts.receiver.ok_or(...)?;
   // ... validates receiver matches payer
   Ok(receiver)
}
```

However, the revoke instruction requires the authority to be the delegator:

```
if delegator_bytes != accounts.authority.address().as_ref() {
   return Err(MultiDelegatorError::Unauthorized.into());
}
```

This means the sponsor has no independent path to trigger closure and reclaim their rent. The sponsor is entirely dependent on the delegator to revoke. This applies to both fixed and recurring delegations created via the optional sponsor flow in `create_delegation_account()`.

Recommendation: Acknowledge as a known trust assumption and document that sponsors accept the risk of indefinite rent lock when funding delegation accounts on behalf of others.

Alternatively, consider allowing the sponsor to reclaim rent under specific conditions, such as after the delegation has expired (`expiry_ts != 0 && current_ts > expiry_ts`), where the delegation is no longer functional and closure has no effect on either party's authorization.

Solana Foundation: Fixed in PR 21.

Cantina Managed: Verified fix. The sponsor expiry check uses a strict comparison (`expiry_ts > current_ts`) without the drift tolerance. This is fine since the drift tolerance is a leniency for delegatees performing transfers, not for rent reclamation. Being strict here is the safer default.

3.4.3 Unchecked arithmetic in subscription transfer period length calculation

Severity: Informational

Context: transfer_subscription.rs#L61

Description: In `transfer_subscription.rs`, the plan's period length is converted from hours to seconds using unchecked multiplication:

```
period_length_s = plan.data.period_hours * 3600;
```

While `period_hours` is validated at plan creation time to be <= 8760 (`MAX_PLAN_PERIOD_HOURS`), making overflow impossible under current constraints (8760 × 3600 = 31,536,000, well within `u64` range), this relies on an invariant enforced elsewhere rather than being locally defensive.

Recommendation: Replace the unchecked multiplication with `checked_mul` for consistency and defense in depth:

```
period_length_s = plan.data.period_hours
  .checked_mul(3600)
  .ok_or::<ProgramError>(MultiDelegatorError::ArithmeticOverflow.into())?;
```

Solana Foundation: Fixed in PR 14.

Cantina Managed: Verified fix.

3.4.4 Re-initializing `MultiDelegate` revives orphaned delegation accounts

Severity: Informational

Context: close_multidelegate.rs#L39

Description: The `MultiDelegate` PDA is derived deterministically from `["MultiDelegate", user, token_mint]` with no nonce or generation counter. When a user closes their `MultiDelegate` via `close_multidelegate()`, any outstanding delegation PDAs (fixed, recurring, or subscription) are not invalidated and they continue to exist on-chain with headers referencing the now-closed `MultiDelegate` address.

If the user later re-initializes a `MultiDelegate` for the same mint, the new PDA is created at the same address as the original. All previously orphaned delegations become functional again because `transfer_with_delegate()` loads the `MultiDelegate` solely by address to obtain the signing bump, and the re-created account satisfies that lookup.

The sequence:

1. Alice creates `MultiDelegate` for USDC, creates fixed delegation granting Bob 1000 USDC.
2. Alice closes `MultiDelegate` (delegations remain on-chain, transfers fail because `MultiDelegate` is gone).
3. Time passes, Alice forgets about the old delegation to Bob.

4. Alice re-initializes `MultiDelegate` for USDC (same deterministic PDA address).
5. Bob's old delegation is live again and Bob can transfer up to the original remaining allowance.

The `initialize_multidelegate()` instruction is idempotent by design (it checks `data_len() == 0` before creating), so this only occurs when the account was fully closed and then re-created. There is no generation counter, epoch, or nonce on the `MultiDelegate` that delegations could validate against to detect staleness.

Recommendation: Document that closing a `MultiDelegate` does not invalidate outstanding delegations, and that users should revoke all active delegations before closing.

Alternatively, consider adding a monotonic generation counter to the `MultiDelegate` that is incremented on each initialization, and storing the expected generation in delegation headers so that stale delegations fail validation on transfer.

Solana Foundation: Fixed in PR 12.

Cantina Managed: Verified fix.

3.4.5 Test suite tests instructions in isolation

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The program has 173 tests with strong per-instruction coverage: input validation, account constraints, signer/writable checks, and single-instruction error paths are thorough. However, the suite contains **zero cross-instruction lifecycle tests and zero multi-actor adversarial sequences**. Every instruction is tested in isolation.

All major vulnerabilities found in this audit are **composition bugs** and individual instructions behave correctly, but sequencing them reveals exploitable state transitions. Additionally, Token-2022 is validated at initialization (10 extension-blocking tests) but has **zero transfer-path tests** and no test performs a delegation transfer through a Token-2022 mint.

Recommendation: Consider improving the test suite by adding more adversarial sequences and full-lifecycle tests.

Solana Foundation: Fixed in PR 19.

Cantina Managed: Fix verified.