Adevar Labs

Prepared for:

DeFi Carrot

November 28, 2025

Carrot Lend Vault Audit Report

Version: after-fix report

Table of Contents

1. Executive Summary

About Clend Vaults

Clend Vaults is a decentralized asset management protocol on Solana that abstracts yield generation. Users deposit assets into a central tokenized vault and receive shares representing their stake in the protocol's Net Asset Value. A designated vault manager actively deploys this aggregated capital into the Clend lending protocol, performing deposits, borrows, and swaps to execute yield strategies. The vault's share price automatically reflects the performance of these underlying managed positions, offering depositors passive exposure to complex strategies.

Audit Summary

The following table summarizes the distribution of identified vulnerabilities by risk level:

Risk Level	Count	Fixed	Acknowledged
Critical	1	1	0
High	2	2	0
Medium	5	5	0
Low	5	5	0

Number of Enhancement Opportunities: 5

Overall Security Posture

This section describes the security posture before and after the fix review. The fix review is the stage of the audit where Adevar Labs checked the fixes implemented by the Carrot Lend development team for the issues found during the audit.

After-fix Review

The Carrot Lend (Clend) team has effectively reviewed and fixed all proposed issues.

Before-fix Review

The Carrot Lend (Clend) team has built a protocol with a clear and logical architecture, interfacing with the Clend lending protocol and Jupiter Aggregator V6. The code demonstrates a good grasp of the Anchor framework and performing Cross-Program Invocations (CPIs) to external protocols.

However, the current implementation contains several vulnerabilities related to accounting logic and oracle handling that could lead to vault insolvency or frozen funds. The protocol's core equity calculation is susceptible to underflows and misinterprets the state of Clend accounts, leading to an overstatement of vault equity. Furthermore, the protocol's reliance on user-supplied oracles without sufficient validation creates a direct path for reserve draining.

Additionally, a trust assumption is placed on the vault manager, who has unilateral control over all capital deployment and strategy execution.

Key Findings Highlight

Cross-Asset Oracle Confusion allows users to supply a mismatched oracle during issue or redeem, leading to mis-pricing of shares and a direct reserve drain.

Unsigned Arithmetic in Equity Calculation (**fold_clend_accounts_equity**) can panic on underflow if liabilities are processed before sufficient assets, freezing vault operations.

Incorrect Balance Classification Logic (mistaking **is_liab**) causes some liabilities from the Clend protocol to be incorrectly counted as assets, leading to an overstatement of vault equity and potential insolvency.

Negative Oracle Price Acceptance in core math functions risks accounting corruption, as the code uses the absolute value of a price instead of reverting.

Launch Recommendations

After-fix Review

All "Mandatory" and "Strongly recommended" issues have been successfully addressed. As the team prepares for launch, we recommend having monitoring solutions in place that will track manager-executed transactions to detect any anomalies. Additionally, we recommend monitoring vault NAV for any anomalies.

Before-fix Review

I. Mandatory before launch:

- Fix the "Critical" and "High" severity issues:
- Enforce a strict 1:1 binding between the asset_mint and its configured oracle in issue and redeem handlers to prevent oracle confusion.
- Correct the **fold_clend_accounts_equity** function to use signed arithmetic or separate accumulators to prevent panics from underflow.
- Mitigate the "First Dust Depositor" attack by making the initial share mint proportional to the first deposit's value, rather than a fixed 100 shares.

II. Strongly recommended:

- Fix the "Medium" severity issues:
- Add a check in **calc_usd_amount** and **calc_token_amount** to revert the transaction if the oracle price is zero or negative.
- Implement slippage protection (e.g., min_shares_out, min_token_out arguments) in the issue and redeem functions to protect users from price volatility.
- Align the **is_liab** balance classification logic with the Clend protocol's implementation to prevent equity overstatement.
- Ensure **origination_fee** is cast to I80F48 before computing to prevent truncation of the fractional part.
- Fix the incorrect calculation in **borrowed_amount_with_fee**.

III. Post-Launch Monitoring:

- Implement robust monitoring and alerting for all manager-executed transactions (deposits, borrows, swaps).
- Actively monitor vault NAV calculations for anomalies to detect any accounting drift.

IV. Future Considerations:

- Explore pathways to decentralize the manager role, such as transitioning to a multi-sig or a DAO-controlled strategy.

Audit Scope

- **Repository:** https://github.com/DeFi-Carrot/clend-vaults
- **Before-Fix Commit Hash:** **c836328c28f1579e971e4509d4ec67f8324c5750**
- **After-Fix Commit Hash:** **c13f4daad8259da7db3a0db0a4f3ae4fdb45a726**
- **Files/Modules in Scope:**
  - protocol/
- **Exclusions:**
  - instructions/add_asset.rs
  - instructions/add_clend_account.rs
  - instructions/init_vault.rs
  - instructions/update_*.rs
  - state/clend.rs
  - state/errors.rs
  - state/events.rs

Note: During the engagement, the Carrot Lend team requested the inclusion of additional changes (commit **5a282681ad74d16ad539683da2036dd6b2c5467a**), which were integrated into this report.

2. Assumptions and Considerations

Audit Assumption

Assumption 1: The vault managers will perform at most 1 swap and/or borrow operation per issue/redeem flow.

Centralization Risks

Assumption 2: The vault managers are a trusted role and will be an account owned by the Carrot Lend protocol.

Trust Assumptions

Assumption 3: While anyone can create a vault, only vaults managed by Carrot Lend will be processed by the off-chain backend.

3. Severity Definitions

Each issue identified in this report is assigned a severity level based on two dimensions: **Impact** and **Likelihood**. These dimensions help project our team's understanding of 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.

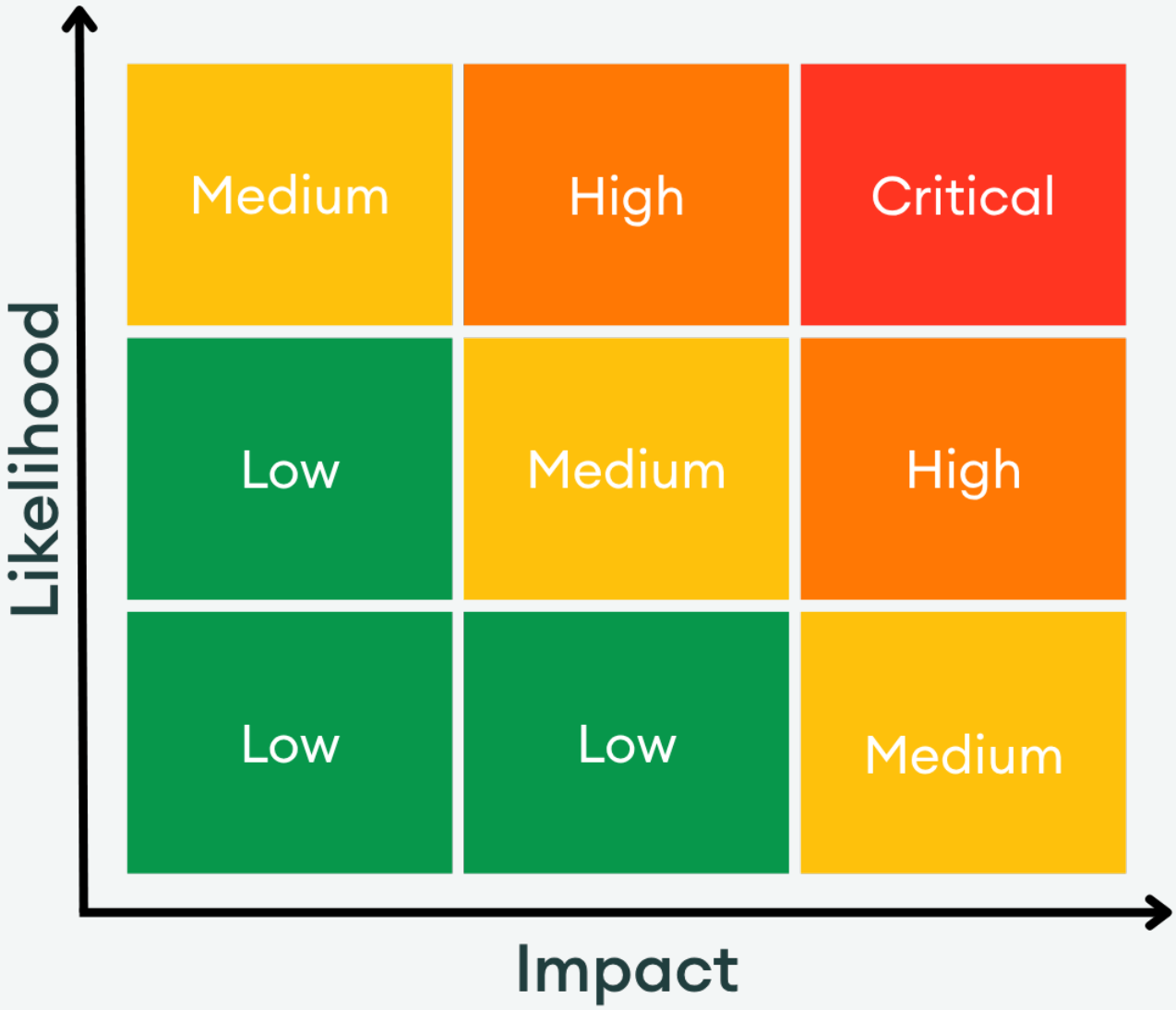
- **High Likelihood**: The vulnerability is trivially exploitable. This means it can be exploited by a wide range of actors without privileged access rights, with minimal capital requirements and low financial risks.
- **Medium Likelihood**: This type of vulnerability can be found and exploited with moderate effort. It might require a significant capital investment, but with manageable financial risk.
- **Low Likelihood**: Exploitation of these vulnerabilities is often technically unfeasible 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 substansial 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.



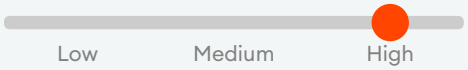
Severity Matrix

4. Findings

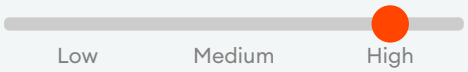
C01: Cross-asset oracle confusion enables share mint mispricing and reserve drain

Status: Resolved

Impact:



Likelihood:



Severity: Critical

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/instructions/issue.rs#L51-L59

>_ issue.rs RUST

```
49:         });
50:
51:         let oracle_pks = vault.get_asset_oracles();
52:         let oracles = load_oracles(&oracle_pks, &ctx.remaining_accounts)?;
53:
54:         let oracle = find_by_key(
55:             oracles.iter(),
56:             &ctx.accounts.asset_oracle.key(),
57:             |o| &o.pk,
58:             ClendVaultsError::OracleNotFound,
59:         )?;
60:
61:         let shares_supply = ctx.accounts.shares.supply;
```

Description:

In both **issue** and **redeem** flows, the asset oracle used to value deposits and withdrawals is selected based on the **asset_oracle** account passed by the user. There is no enforced binding between the supplied **asset_mint** and the oracle expected for that specific asset.

The handler performs the following steps:
See:
- src/instructions/issue.rs#L51-L59
- src/instructions/redeem.rs#L77-L85

RUST

```
let oracle_pks = vault.get_asset_oracles(); // get list of oracles from active vault assets
    let oracles = load_oracles(&oracle_pks, &ctx.remaining_accounts)?; // return list of
    oracles matched with provided remaining_accounts

    // iterate on the list, return the one that pubkey matches with user-provided asset_oracle
    let oracle = find_by_key(
        oracles.iter(),
```

... continued RUST

```
        &ctx.accounts.asset_oracle.key(),
        |o| &o.pk,
        ClendVaultsError::OracleNotFound,
    )?;
```

Here, the supplied **asset_oracle** is simply matched against *any* oracle registered in the vault. The protocol does not assert that the oracle corresponds to the **asset_mint** used in the deposit or redemption. Therefore, a user may choose an oracle belonging to a different asset in the vault that has a more favorable price.

This results in the protocol using incorrect price data when calculating the token contribution or redemption amounts. A user can mint more shares than they should by pricing a deposit using a higher-valued oracle or redeem more tokens than allowed by pricing withdrawals at a lower oracle value.

Recommendation:

Enforce a 1:1 binding between each **asset_mint** and its configured oracle.

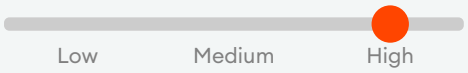
Developer Response:

Added a check enforcing a binding between the asset_mint and its oracle.

H01: Unsigned arithmetic in fold_clend_accounts_equity may underflow when liability value exceeds current sum leading to temporary fund freezing of vault

Status: Resolved

Impact:



Likelihood:

Severity: High

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/utils/accounts.rs#L110

>_ accounts.rs RUST

```rust
    enforce_max_leverage: bool,
) -> Result<u128> {
    let mut sum: u128 = 0;

    for clend_account in clend_accounts {
```

Description:

The function **fold_clend_accounts_equity()** iterates through all active clend accounts and accumulates vault equity by adding asset values and subtracting liabilities:

RUST

```rust
let mut sum: u128 = 0;
...
if is_liab {
    sum = sum.checked_sub(usd)?;
    sum = sum.checked_add(e_usd)?;
} else {
    sum = sum.checked_add(usd)?;
    sum = sum.checked_add(e_usd)?;
    acct_assets_usd = acct_assets_usd.checked_add(usd)?;
}
```

The variable **sum** is of type **u128**, meaning it cannot represent negative values.
The logic implicitly assumes that the cumulative sum of assets will always be greater than or equal to the liabilities processed so far. However, this assumption is flawed during iterations, it is possible for a liability entry to exceed the current **sum**.

When this happens, the line:

RUST

```rust
sum = sum.checked_sub(usd)?;
```

attempts to subtract a larger value from a smaller one, producing an **underflow** that triggers a panic and aborts the instruction.

This issue is independent of the iteration order: it can occur at any point in the loop where the

cumulative sum of prior equity is smaller than the next liability value. For example, if one large borrower position follows smaller depositor positions, the temporary equity accumulator can go negative mid-iteration, a state that cannot be represented in u128.

Any vault operation that calls **fold_clend_accounts_equity()** (e.g., share valuation, issue, redeem) may **panic** and fail if total liabilities processed exceed the accumulated asset sum at any iteration step.

This could lead to a temporary freezing of funds until the upgrade to fix the logic as the manager cannot simply unfold this by switching clend accounts' position.

Proof of Concept:

Suppose the vault tracks two clend accounts (values already converted to USD):
Account_A: Balance_1 = 100 (Assets), Balance_2 = 100 (Assets)
Account_B: Balance_1 = 300 (Liabilities), Balance_2 = 1000 (Assets)

Iteration steps:
1. Start: **sum = 0**
2. Account_A → add 100 + 100 → **sum = 200**
3. Account_B (Balance_1) → **sum.checked_sub(300)?**
4. Since **sum = 200 < 300**, this subtraction **underflows** and triggers a panic before Balance_2 is processed.

Recommendation:

Use **signed arithmetic** for the accumulator or separately track total assets and total liabilities before computing net equity.

Developer Response:

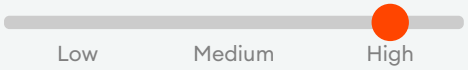
Signed accumulator is now used to prevent underflow.

H02: First dust depositor severely limits vault TVL due to fixed 100-share genesis

Status: Resolved

Impact:

Likelihood:



Severity: High

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/utils/math.rs#L11-L13

>_ math.rs RUST

```
    round_up: bool,
) -> u64 {
    if vault_equity.le(&0) || shares_supply.le(&0) {
        // if vault_equity or shares_supply is 0 or less, just mint 100 shares
        ui_to_amount(100.0, shares_decimals)
    } else {
        // Calculate shares
```

Description:

On vault initialization, the first deposit triggers a special-case rule in **shares_earned** that mints a fixed **100 shares** when **vault_equity == 0** or **shares_supply == 0**:

RUST

```
if vault_equity.le(&0) || shares_supply.le(&0) {
        // if vault_equity or shares_supply is 0 or less, just mint 100 shares
        ui_to_amount(100.0, shares_decimals)
    } else {
}
```

The vault's share mint uses **9 decimals** (according to the test case), so this first deposit mints exactly:

```
100 * 10^9 = 1e11 share base units
```

There is **no minimum deposit requirement**. The first depositor can initialize the vault with the smallest possible token unit, e.g., **1 micro-unit of USDC**,

When this single base-unit deposit mints 1e11 share base units, the initial **price per share (in base units)** becomes:

```
price_per_share = deposited_value / minted_shares
= 1 USDC_base / 1e11 share_base
```

```
... continued
= 1e-11 USDC_base per share_base
```

Each **1 full USDC (1e6 USDC_base)** deposited afterward will mint:

```
shares_minted = 1e6 / 1e-11 = 1e17 share_base units
```

The SPL Token program limits total supply to **u64::MAX = 1.8446744e19** base units.
Once the vault's total share supply exceeds this value, any **mint_to()** CPI in **issue_handler** will **fail and revert** the transaction.

We can now estimate the **maximum vault capacity** (TVL cap):

```
max_total_shares = 1.8446744e19 share_base
price_per_share  = 1e-11 USDC_base/share_base
max_TVL          = 1.8446744e19 * 1e-11 = 1.8446744e8 USDC_base
```

Converting **USDC_base** (1e6 = 1 USDC):

```
1.8446744e8 / 1e6 = 184.46744 USDC
```

So the vault's **maximum total deposit capacity - 184.46 USDC**.

Beyond that point, additional deposits revert permanently because of the SPL mint total-supply limit.

As a result, a user can first deposit to the vault with a single smallest USDC base unit and permanently cap the vault's total deposit capacity to around **184 USDC**.

After reaching this implicit limit, all further deposits revert, effectively **bricking the vault**.

This issue mirrors OZ's ERC4626 **decimals_offset**, where a larger offset limits the total supply. However, it's even more restrictive on Solana because the SPL token standard uses a u64 (64-bit unsigned integer) for its total supply, imposing a smaller maximum limit than is typically found in EVM environments.

Recommendation:

- Make the **genesis share amount proportional to the first deposit** rather than fixed at 100 shares.

Developer Response:

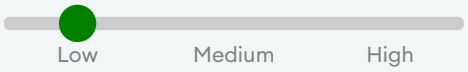
Added a check where first depositor must be the vault manager.

M01: Negative oracle price should abort transaction

Status: Resolved

Impact:

Likelihood:



Severity: Medium

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/utils/math.rs#L57

>_ math.rs RUST

```rust
55:
56:     let token_amount = token_amount as u128;
57:     let price = price.abs() as u128;
58:
59:     // Scale the token amount to the base unit (USD cents)
```

Description:

The **calc_usd_amount** and **calc_token_amount** functions are using the oracle price for the calculations but instead of reverting the transaction on a negative price, the absolute value is taken. Providing a negative price for an asset is generally an indication of a faulty oracle and can put the entire calculation mechanism at risk.

Recommendation:

Add an explicit check for negative prices and revert if that is the case. For example:

RUST

```rust
require!(price > 0, ClendVaultsError::InvalidOraclePrice);
```

Developer Response:

Added checks that will abort on negative oracle price.

M02: Lack of slippage protection in Issue and Redeem flows enables first-depositor share-price manipulation and value loss

Status: Resolved

Impact:

Likelihood:

Severity: Medium

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/instructions/redeem.rs#L14

>_ redeem.rs RUST

```rust
use clend_protocol_cpi::program::Clend;

pub fn redeem_handler<'a, 'b, 'c: 'info, 'info>(
    ctx: Context<'a, 'b, 'c, 'info, Redeem<'info>>,
    args: RedeemArgs,
```

Description:

The vault's **issue_handler** and **redeem_handler** functions do not implement any **slippage check** between the expected and actual share price at the time of execution.

As a result, users interact with a volatile pricing function, derived from total vault equity in USD, without any bound on price movement between transaction construction and on-chain execution.

Recommendation:

Implement **slippage and sanity checks** on both deposit (issue) and redemption flows:

For issue:
- Revert if **shares_owed == 0**:

```rust
require!(shares_owed > 0, ClendVaultsError::ZeroSharesMint);
```

- Allow users to specify a **min_shares_out** argument and revert if fewer are minted.
- In order to completely mitigate the first-depositor attack, the protocol must ensure that the vault manager performs the initial deposit and owns the bootstrap shares, treating them as non-redeemable dead shares to establish a starting price.

For redeem:
- Accept a **min_token_out** argument and revert if the returned assets amount is below the user's tolerance threshold.

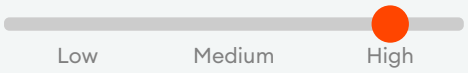
Developer Response:

For both handlers, slippage parameters are now enforced.

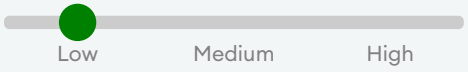
M03: Mismatch in balance classification logic (is_liab) allows liabilities to be treated as assets leading to overstated equity calculation

Status: Resolved

Impact:



Likelihood:



Severity: Medium

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/utils/accounts.rs#L119-L120

>_ accounts.rs RUST

```
117:             let bank = decode_clend_bank_lite(bal.bank_pk, remaining)?;
118:
119:             // amount in native
120:             let is_liab = bal.asset_shares.is_zero();
121:             let amt: u64 = if is_liab {
122:                 bal.get_liability_amount(bank.liability_share_value)?
```

Description:

The vault reproduces clend's account-side inference logic (deciding whether a balance represents **assets** or **liabilities**) but simplifies it incorrectly.
In the original clend code, this logic is implemented in **Balance::get_side()** and uses a fixed-point comparison:

RUST

```rust
pub const EMPTY_BALANCE_THRESHOLD: I80F48 = I80F48!(1);
pub fn get_side(&self) -> Option<BalanceSide> {
    let asset_shares = I80F48::from(self.asset_shares);
    let liability_shares = I80F48::from(self.liability_shares);
    assert!(
        asset_shares < EMPTY_BALANCE_THRESHOLD || liability_shares < EMPTY_BALANCE_THRESHOLD
    );

    if liability_shares >= EMPTY_BALANCE_THRESHOLD {
        Some(BalanceSide::Liabilities)
    } else if asset_shares >= EMPTY_BALANCE_THRESHOLD {
        Some(BalanceSide::Assets)
    } else {
        None
    }
    ...

}
```

The **I80F48!(1)** constant defines a threshold of one full token in **fixed-point decimal format (I80F48)**.
That means a small dust balance of less than 1.0 (e.g. 0.5, 0.01) is interpreted as an empty quantity.

In the vault's own code (e.g., within **fold_clend_accounts_equity()**), the side detection is simplified to:

```rust
let is_liab = bal.asset_shares.is_zero();
```

This check treats **only exact zero** as "no asset", ignoring fractional dust below 1 token.

As a result, if an account has:
- **asset_shares = 0.5** (non-zero but < 1 token), and
- **liability_shares = 100**,

then clend's **get_side()** classifies the balance as **Liabilities**,
but the vault's logic misclassifies it as **Assets**, since **is_zero()** returns false.

This discrepancy leads to an accounting error when aggregating vault equity:
- Liabilities are **omitted** from subtraction.
- The vault's computed **equity_usd** and **acct_assets_usd** are **overstated** (no liability).

Critically, if the vault manager borrows from clend that results in this state (with a small **asset_shares** dust balance and large liabilities), the borrowed amount will be **added to vault equity** (in **fold_reserve_equity()**) instead of **counted as debt**.

This causes the vault's share price to inflate and subsequent withdrawals by depositors would be from overstated equity, leading to vault insolvency.

Proof of Concept:

The following rust snippet imitates some of the logic of clend protocol borrow instruction to demonstrate that it's possible that after the borrow instruction, there will be some dust balance left in asset_share. Consequently, the vault will interpret this as "Assets" side, omitting liabilities.

```rust
use fixed::types::I80F48;
use fixed_macro::types::I80F48;
fn main() {
    // Let's assume that our current asset_share is a dust amount (< 1 SPL token)
    let current_asset_share = I80F48!(0.5);

    // Current asset_share_value (picked this up from random clend bank account)
    let asset_share_value = I80F48!(1.01121906543246);

    // zero-check as in fold_clend_accounts_equity()
    // assert that it's "FALSE" indicating that it interprets dust amount as "Assets" side
    // a divergence from canonical logic
    assert!(!current_asset_share.is_zero());

    // Suppose that we're borrowing 100 token -> clend would try to deduct the total amount of
    // current asset_amount

    // asset_share -> convert to asset_amount -> covert back to asset_share -> deduct from
    balance

    let asset_amount = current_asset_share.checked_mul(asset_share_value).unwrap();
    println!("asset_amount = {}", asset_amount);
```

... continued RUST

```
    let asset_share = asset_amount.checked_div(asset_share_value).unwrap();
    println!("asset_share = {}", asset_share);

    let result = current_asset_share.checked_add(-asset_share).unwrap();
    println!("remaining_asset_share = {}", result);

    // Even after borrow, there is still dust, so clend vault would still interprets this as
    // "Assets" side, omitting liabilities.
    assert!(!result.is_zero());
}
```

Recommendation:

Align vault-side balance classification with clend's canonical logic to ensure consistent accounting.

Developer Response:

Canonical implementation of **get_side** from the protocol is now replicated in the vault logic.

M04: Truncated origination_fee will lose all execution borrow fees

Status: Resolved

Impact:

Likelihood:

Severity: Medium

Location: https://github.com/AdevarLabs/clend-vaults-execution-fee-source/blob/execution-fee-pr/protocol/programs/clend-vaults/src/instructions/clend_account_borrow.rs#L68-L75

>_ clend_account_borrow.rs RUST

```
66:
67:     // calculate borrow outputs
68:     let (borrow_fee, borrow_fee_usd, borrowed_amount_with_fee) = if origination_fee > 0 {
69:         // calculate borrowed amount with fee
70:         let borrowed_amount_with_fee = args
71:             .amount
72:             .checked_mul(origination_fee)
73:             .ok_or(ClendVaultsError::MathOverflow)?
74:             .checked_sub(args.amount)
75:             .ok_or(ClendVaultsError::MathUnderflow)?;
76:
77:         // subtract original borrow amount from borrowed amount with fee to get borrow fee
```

Description:

The clend-vault protocol implements an execution fee which is meant to have the user executing a transaction pay for the incurred borrow and swap fees. In **clend_account_borrow**, the borrow fee is calculated using the **origination_fee** of the corresponding bank of the clend_account.
In the bank, the origination fee is stored as **protocol_origination_fee: I80F48**. The **I80F48** type is meant to represent fractional numbers, containing both an integer and a fractional component. **clend_account_borrow_handler** casts the **I80F48** type to a **u64** type using **checked_to_num**, losing the fractional component. In the case of the **origination_fee**, which will only contain a fractional component, this effectively truncates to zero.

Proof of Concept:

Add this at the end of **clend_account_borrow.rs**:

RUST

```
#[cfg(test)]
mod conversion_tests {
    // Import necessary items from the fixed crate
    use fixed::types::I80F48;
    use fixed_macro::types::I80F48;

    // The 'fixed!' macro allows for easy constant creation of fixed-point numbers.
    // It is important to define this outside of the test function if using an older Rust
```

... continued

```rust
    version
    // but placing it here is fine for modern Rust versions.
    // The syntax I80F48!(...) is provided by the fixed crate.
    const FIXED_0_17: I80F48 = I80F48!(0.17);

    /// Tests the safe conversion of a fixed-point number (0.17) to u64.
    /// This demonstrates that the fractional part is truncated (rounds toward zero).
    #[test]
    fn test_i80f48_fraction_truncation() {
        // 1. Arrange: The fixed-point value 0.17
        let fixed_val = FIXED_0_17;

        // 2. Act: Perform the checked conversion to u64
        // The checked_to_num method truncates the fractional part.
        let result: Option<u64> = fixed_val.checked_to_num::<u64>();

        // 3. Assert: Verify the result
        // Since the integer part of 0.17 is 0, we expect Some(0).
        assert_eq!(result, Some(0), "The fractional part should be truncated, resulting in 0."
    );
    }
}
```

Recommendation:

Update the faulty implementation with this:

```rust
let origination_fee: I80F48 = bank_state
        .protocol_origination_fee
        .into();

// calculate borrow outputs
let (borrow_fee, borrow_fee_usd, borrowed_amount_with_fee) = if origination_fee > 0 {
    // calculate fee
    let borrow_fee: u64 = I80F48::from_num(args.amount)
        .checked_mul(origination_fee)
        .ok_or(ClendVaultsError::MathOverflow)?
        .checked_to_num()
        .ok_or(ClendVaultsError::MathOverflow)?;

    // calculate borrowed amount with fee
    let borrowed_amount_with_fee = borrow_fee
        .checked_add(args.amount)   // /!\ this is including the fix for the other issue
        .ok_or(ClendVaultsError::MathOverflow)?;
```

Developer Response:

Calculation of the **borrow_fee** is now done as **I80F48** to ensure no truncation of the fractional value, before being casted to **u64**.
The fix also resolve L05.

M05: borrowed_amount_with_fee subtracts the fees from the amount when it should add them

Status: Resolved

Impact: Low Medium High

Likelihood: Low Medium High

Severity: Medium

Location: https://github.com/AdevarLabs/clend-vaults-execution-fee-source/blob/1fe71b7dcc44903ef1db06c60af56af4e05a38be/protocol/programs/clend-vaults/src/instructions/clend_account_borrow.rs#L70-L74

>_ clend_account_borrow.rs RUST

```
68:     let (borrow_fee, borrow_fee_usd, borrowed_amount_with_fee) = if origination_fee > 0 {
69:         // calculate borrowed amount with fee
70:         let borrowed_amount_with_fee = args
71:             .amount
72:             .checked_mul(origination_fee)
73:             .ok_or(ClendVaultsError::MathOverflow)?
74:             .checked_sub(args.amount)
75:             .ok_or(ClendVaultsError::MathUnderflow)?;
76:
```

Description:

The calculation for **borrowed_amount_with_fee** subtracts the borrow amount from the fee instead of adding them together. This results in a value that is smaller than the actual borrowed amount.
Subsequently, when attempting to calculate **borrow_fee** right after by subtracting the original amount, the transaction will most likely revert with a **MathUnderflow** error since **borrowed_amount_with_fee < args.amount**.

Recommendation:

Change the operation to add the fee to the borrowed amount:

DIFF

```
let borrowed_amount_with_fee = args
     .amount
     .checked_mul(origination_fee)
     .ok_or(ClendVaultsError::MathOverflow)?
-    .checked_sub(args.amount)
+    .checked_add(args.amount)
     .ok_or(ClendVaultsError::MathUnderflow)?;
```

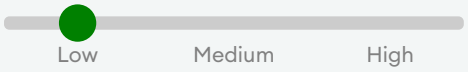
Developer Response:

Fixed according to our recommendations, see M05.

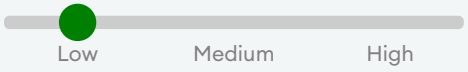
L01: Management fee timestamp updated even when computed fee rounds to zero

Status: Resolved

Impact:



Likelihood:



Severity: Low

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/state/clend_vaults.rs#L654-L659

>_ clend_vaults.rs — RUST

```
652:             / SECONDS_IN_YEAR as u128;
653:
654:         // update timestamp
655:         self.management_fee_last_update = current_time;
656:
657:         if fee_usd_cents == 0 {
658:             return 0;
659:         }
660:
661:         // convert usd cents to shares ui based on NAV
```

Description:

The management fee accrual logic updates the **management_fee_last_update** field before checking whether the computed fee amount is non-zero:

RUST

```
self.management_fee_last_update = current_time;

if fee_usd_cents == 0 {
     return 0;
}
```

If **fee_usd_cents** rounds down to zero, the function still resets the timestamp and returns early. This effectively forfeits the elapsed time window that produced a sub-cent result instead of carrying it forward to the next accrual.

However, under the current configuration this is unlikely to occur. **vault_equity** is represented with 9-decimal precision, and the accrual tick is set to 60 seconds, a deliberately chosen interval that ensures the computed fee remains above one cent for reasonable management fee rates (e.g. 1-2% per year).

Even at a much lower illustrative rate such as 0.06% per year, a 60-second accrual on a 1 USD vault still yields 114 in **fee_usd_cents**, meaning **fee_usd_cents** will not be zero.

Therefore, the behavior is mostly theoretical under current assumptions, but may surface if parameters are changed.

Recommendation:

For robustness against future configuration changes, update **management_fee_last_update** only when a non-zero fee is realized

Developer Response:

Ordering was fixed so that management fee only gets updated if there is a non-zero fee.

L02: Two-step rounding-up in Issue fee conversion results in gradual value leak from shareholders

Status: Resolved

Impact: Low Medium High

Likelihood: Low Medium High

Severity: Low

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/state/clend_vaults.rs#L693-L709

>_ clend_vaults.rs RUST

```rust
691:         let fee_amount_in_shares = if fee_amount_in_asset_units > 0 {
692:             // 2. Convert the fee (in asset units) to its USD value.
693:             let fee_usd = calc_usd_amount(
694:                 fee_amount_in_asset_units,
695:                 asset_decimals,
696:                 asset_price,
697:                 asset_price_expo,
698:                 true,
699:             )
700:             .ok_or(ClendVaultsError::MathOverflow)?;
701:
702:             // 3. Convert that USD value into the equivalent number of vault shares.
703:             let fee_in_shares = shares_earned(
704:                 fee_usd,
705:                 adjusted_shares_supply,
706:                 shares_decimals,
707:                 vault_equity,
708:                 true,
709:             );
710:
711:             // 4. Add the calculated shares amount to the accumulator.
```

Description:

In the **issue_handler** flow, the vault charges an **issue fee** on the depositor's asset amount before minting shares.
However, the conversion path for this fee applies **two consecutive rounding-up operations**, which causes the minted fee shares to represent **slightly more value** than the actual fee deducted.

Formally, when a user deposits **X** units of the asset:

1. The vault splits the deposit into:

* **y**; user's deposit amount
* **k**; fee portion (**y + k = X**)
2. The fee amount **k** is converted to USD, **rounded up once.**
3. The USD value is then converted to vault shares (**z**), **rounded up again.**

If we back-convert the final minted fee shares (**z**) into the equivalent asset amount (**k'**), the relationship becomes:

k' > k > y + k' > X

This means the protocol receives **more value in shares** than the depositor actually paid in fees.

Because total vault equity only increases by the actual deposit **X**, while the protocol fee shares correspond to a slightly larger value (**k'**), the surplus portion is implicitly drawn from the **existing vault equity**, diluting all prior shareholders.

Recommendation:

Charge the issue fee in shares after pricing, like redemption fee. This will ensure that the fee is paid from depoistor as intended.

Developer Response:

Issue fee is now charged in shares after pricing.

L03: Missing Verification-level check accepts partially verified Pyth oracle data

Status: Resolved

Impact: Low Medium High

Likelihood: Low Medium High

Severity: Low

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/utils/price.rs#L104-L124

>_ price.rs RUST

```
102:     }
103:
104:     pub fn get_price(&self) -> Result<(i128, i32, i64)> {
105:         let max_confidence_threshold: u64 = 5_000_000_000; // TODO: not sure what to s
         et here
106:         let ema_conf = self.price_update_v2.price_message.ema_conf;
107:         let oracle_pk = self.pk;
108:
109:         // if confidence interval provided by pyth is outside our acceptable tolerance
         , error
110:         if ema_conf > max_confidence_threshold {
111:             msg!(
112:                 "oracle {:?} price confidence is outside tolerance, ema_conf: {}, max_
         confidence_threshold: {}",
113:                 oracle_pk,
114:                 ema_conf,
115:                 max_confidence_threshold
116:             );
117:             return Err(ClendVaultsError::PriceConfidenceOutsideTolerance.into());
118:         }
119:
120:         Ok((
121:             self.price_update_v2.price_message.ema_price.into(),
122:             self.price_update_v2.price_message.exponent,
123:             self.price_update_v2.price_message.publish_time,
124:         ))
125:     }
126: }
```

Description:

The **PythOracle::get_price()** function accepts and returns oracle price data without enforcing any validation on the **verification_level** field contained in **PriceUpdateV2**.

RUST

```
pub fn get_price(&self) -> Result<(i128, i32, i64)> {
let max_confidence_threshold: u64 = 5_000_000_000;
let ema_conf = self.price_update_v2.price_message.ema_conf;
```

... continued

```rust
  if ema_conf > max_confidence_threshold {
      return Err(ClendVaultsError::PriceConfidenceOutsideTolerance.into());
  }
  Ok((
      self.price_update_v2.price_message.ema_price.into(),
      self.price_update_v2.price_message.exponent,
      self.price_update_v2.price_message.publish_time,
  ))
}
```

The **PriceUpdateV2** structure includes a **verification_level** field that distinguishes between fully verified and partially verified oracle messages:

```rust
pub enum VerificationLevel {
    Partial { num_signatures: u8 },
    Full,
}
```

Since the handler does not check this field, **any validly deserialized message** will be accepted, regardless of whether it was fully verified or not.

Fortunately, in current Pyth deployments, publishers enforce a minimum quorum (typically 5 signatures) even for **Partial** updates, which means the immediate risk of accepting an unverified price is mitigated by off-chain rules.

However, vaults rely on accurate valuation, not execution speed. Accepting a partially verified price therefore risks minting or redeeming shares against inaccurate valuations. Since vaults can safely trade latency for correctness, they should only accept fully verified oracle data.

If flakiness is a concern, **Partial**-level verified message can be allowed but the minimum signature threshold should be enforced explicitly on the integration side.

Since Pyth uses a Wormhole guardian set and relies on a super-majority of guardians to validate cross-chain updates, the protocol should also enforce this in order to inherit the same security guarantee of these cross-chain updates (from PythNet).
According to Wormhole docs, this threshold should be at least 13 of 19 signatures of Wormhole Guardian.

Recommendation:

Add explicit verification-level enforcement before accepting oracle price data:

```rust
match self.price_update_v2.verification_level {
    VerificationLevel::Full => {},
    _ => Err()
}
```

Developer Response:

Pyth oracle price now checked for full verification level before used.

L04: Missing verification for Token 2022 Extensions

Status: Resolved

Impact: Low Medium High

Likelihood: Low Medium High

Severity: Low

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/instructions/issue.rs#L102-L108

>_ issue.rs RUST

```
100:         .ok_or(ClendVaultsError::MathOverflow)?;
101:
102:         let shares_owed = shares_earned(
103:             deposit_usd,
104:             adjusted_shares_supply,
105:             ctx.accounts.shares.decimals,
106:             vault_equity,
107:             false,
108:         );
109:
110:         shares_owed
```

Description:

The Carrot Lend vaults enable users to deposit assets that are compatible with SPL Token and SPL Token 2022 mints, selected by the vault's manager. Currently, in the **issue** flow, there is no mechanism accounting for tokens that have a **TransferFee** extension, supported by the SPL Token 2022 standard. Such tokens will cause the protocol to mint an excess number of shares in addition to incorrect **management_fee** and **issue_fee** calculations.

The **TransferHook** extension should also be taken into consideration and may affect protocol accounting. The mint authority can specify logic to execute after each transfer which may impact actual transferred amounts.

Information about SPL Token-2022 extensions and associated security risks can be found here: https://neodyme.io/en/blog/token-2022/

Recommendation:

Mint shares based on the actual amount of tokens received, not the amount the user intends to send. For example:

RUST

```
let before_balance = ctx.accounts.vault_asset_reserve.amount;

let transfer_checked_accounts = TransferChecked {
      from: ctx.accounts.user_asset_account.to_account_info(),
      mint: ctx.accounts.asset_mint.to_account_info(),
      authority: ctx.accounts.user.to_account_info(),
      to: ctx.accounts.vault_asset_reserve.to_account_info(),
```

... continued RUST

```
  };

let ctx = CpiContext::new(ctx.accounts.asset_token_program.to_account_info(), transfer_checked
    _accounts);
transfer_checked(ctx, args.amount, ctx.accounts.asset_mint.decimals)?;

ctx.accounts.vault_asset_reserve.reload(); // update the account in memory

let after_balance = ctx.accounts.vault_asset_reserve.amount;
let received = after_balance.checked_sub(before_balance).ok_or(ErrorCode::UnderflowError)?;

//Perform subsequent calculation on the received amount
```

Also, restrict any problematic Token 2022 extensions the protocol doesn't intend to support during the **add_asset** flow, for example:

RUST

```
let extensions = get_mint_extensions(&ctx.account.mint)?;

if extensions.contains(&ExtensionType::TransferHook) {
    return err!(ExtError::InvalidMint);
}
```

Some SPL Token 2022 mints contain many extensions but the extensions are not used. This arises from the fact that extensions cannot be added after the mint has been created. We recommend treating these tokens as if the extensions were active unless the authority which can modify this has been revoked.

Developer Response:

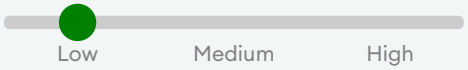
Whitelist implemented for supported extensions which doesn't include TransferFee and TransferHook extensions.

L05: Missing state validation in issue_start allows session state to be overwritten when invoked multiple times

Status: Resolved

Impact:

Likelihood:



Severity: Low

Location: https://github.com/AdevarLabs/clend-vaults-execution-fee-source/blob/1fe71b7dcc44903ef1db06c60af56af4e05a38be/protocol/programs/clend-vaults/src/instructions/issue_start.rs#L120-L122

>_ issue_start.rs RUST

```
118:
119:     // update user session state
120:     let mut user_session = ctx.accounts.user_session.load_mut()?;
121:     user_session.is_issue()?;
122:     user_session.op.set_issue_state(deposit_usd, shares_owed)?;
123:
124:     let transfer_checked_accounts = TransferChecked {
```

Description:

The **issue_start** instruction lacks validation to prevent multiple invocations within a single user session.
When a user calls **issue_start** twice during the same session (between **user_session_start** and **user_session_end**), the second call overwrites the deposit information from the first call.

When **issue_start** is called multiple times within a single session:
1. Each invocation transfers tokens from the user to the vault reserve (line 131-138)
2. Each invocation calculates deposit value and shares independently
3. Each invocation overwrites the session state instead of rejecting or accumulating

The validation only checks that the operation type is **ISSUE**, not whether the state has already been populated.

One impact is that multiple calls in the same transaction to **issue_start** will transfer tokens from the caller, but only register the last deposit. Beyond this immediate issue, the lack of state validation could enable other issues as the session state becomes decoupled from actual vault balance changes.

This same pattern exists in **swap** and **borrow** where **swap_cost** and **borrow_cost** are directly assigned without validation.

Recommendation:

Add validation in **set_issue_state** to prevent state overwrite.

Developer Response:

User Session is now validated by also ensuring that **self.issue_deposit_usd == 0 && self.issue_shares_owed_pre_exec == 0**.

5. Enhancement Opportunities

E01: Checking for zero amount Issuances/Redemptions will prevent transactions with zero economic value

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/instructions/issue.rs#L20

>_ issue.rs RUST

```rust
18: pub fn issue_handler<'a, 'b, 'c: 'info, 'info>(
19:     ctx: Context<'a, 'b, 'c, 'info, Issue<'info>>,
20:     args: IssueArgs,
21: ) -> Result<()> {
22:     let (vault_fee_config, asset_fee_config) = {
```

Description:

The current issue and redeem handlers do not implement a zero/minimum **amount** check. This presents an opportunity for malicious users to spam the protocol with transactions that offer zero economic value. Additionally, there is an opportunity for locking the management fee in the protocol with recurring dust transactions. However, we have presented another issue whose fix would already prevent this, see: **Management fee timestamp updated even when computed fee rounds to zero**.

Potential Benefit:

Prevent transactions offering zero economic value to the protocol, cluttering event logs and valid user transactions.

Recommendation:

Implement a zero or minimum value check on the amount argument in the issue and redeem handlers. For example:

RUST

```rust
require!(args.amount > 0, ClendVaultsError::InvalidAmount);
```

Developer Response:

Added zero amount checks in issue and redeem.

E02: Adding Remaining Account length check will prevent out-of-bounds index access

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/instructions/swap.rs#L43

>_ swap.rs RUST

```
41:     // first account is token program
42:     // second is authority (the vault)
43:     let authority = accounts[1].pubkey;
44:     require!(
45:         authority.eq(&ctx.accounts.vault.key()),
```

Description:

Currently, the remaining accounts in the swap instruction are accessed by index directly without an initial array length check.

Potential Benefit:

Out-of-bounds array access will be prevented.

Recommendation:

Add a length check for the remaining accounts array. For example:

RUST

```
require!(accounts.len() >= 4, ClendVaultsError::InvalidAccountsLength);
```

Developer Response:

Length check for the remaining accounts array added.

E03: Redundant searching of Clend accounts and Banks in Emission settlement loop

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/state/clend_vaults.rs#L314-L346

>_ clend_vaults.rs RUST

```rust
        let mut n_settled: u8 = 0;

        // iterate over each clend account
        for clend_account in clend_accounts {
            let clend_account_state =
                decode_clend_account(clend_account.address, remaining_accounts)?;

            // for each balance
            for balance in clend_account_state.balances.iter().filter(|b| b.active) {
                let bank_pk = balance.bank_pk;
                let bank_state = decode_clend_bank_lite(bank_pk, remaining_accounts)?;

                // if emissions remaining, settle
                if bank_state.emissions_remaining > 0 {
                    let clend_account_info = find_by_key(
                        remaining_accounts.iter(),
                        &clend_account.address,
                        |ai| ai.key,
                        ClendVaultsError::ClendAccountNotFound,
                    )?;
                    let bank_account_info = find_by_key(
                        remaining_accounts.iter(),
                        &bank_pk,
                        |ai| ai.key,
                        ClendVaultsError::ClendBankNotFound,
                    )?;

                    // settle emissions
                    settle_clend_emissions(clend_program, clend_account_info, bank_acc
ount_info)?;

                    // increment counter
                    n_settled = n_settled
                        .checked_add(1)
                        .ok_or(ClendVaultsError::MathOverflow)?;
                }
            }
        }
```

Description:

In the emission-settlement section of the vault logic, the handler performs redundant lookups of **clend_account_info** and **bank_account_info** within the same iteration where these accounts were already deserialized by **decode_clend_account** and **decode_clend_bank_lite**:

RUST

```rust
if bank_state.emissions_remaining > 0 {
        let clend_account_info = find_by_key(
            remaining_accounts.iter(),
            &clend_account.address,
```

... continued

```rust
            |ai| ai.key,
            ClendVaultsError::ClendAccountNotFound,
        )?;
        let bank_account_info = find_by_key(
            remaining_accounts.iter(),
            &bank_pk,
            |ai| ai.key,
            ClendVaultsError::ClendBankNotFound,
        )?;

        settle_clend_emissions(clend_program, clend_account_info, bank_account_info)?;
        n_settled = n_settled.checked_add(1).ok_or(ClendVaultsError::MathOverflow)?;
    }
```

Both the Clend account and its corresponding bank have already been loaded and decoded earlier in the flow via:

```rust
let clend_account = decode_clend_account(...)?;
    let bank_state = decode_clend_bank_lite(...)?;

    pub fn decode_clend_account<'info>(
        pk: Pubkey,
        remaining: &'info [AccountInfo<'info>],
    ) -> Result<ParsedClendAccount> {
        let ai = find_account(&pk, remaining).ok_or(ClendVaultsError::ClendAccountNotFound)?;
    // <-- already found the AccountInfo
        let data = ai.data.borrow();
        let parsed = ParsedClendAccount::from_account_data(&data)?;
        drop(data);
        Ok(parsed)
    }
```

Re-performing **find_by_key** on every iteration adds unnecessary iteration overhead. Since these **AccountInfo** references can be retained or passed directly from the decode routines, the repeated lookup is redundant.

Potential Benefit:

- Compute unit savings

Recommendation:

- Refactor **decode_clend_account** and **decode_clend_bank_lite** to return its corresponding account along with the parsed data

E04: Consolidate redundant mint_to CPIs in distribute_fees_handler

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/instructions/distribute_fees.rs#L15

>_ distribute_fees.rs RUST

```
13:         let management_fee_mint_to_accounts = MintTo {
14:             mint: ctx.accounts.shares.to_account_info(),
15:             to: ctx.accounts.shares_destination.to_account_info(),
16:             authority: ctx.accounts.vault.to_account_info(),
17:         };
```

Description:

In **distribute_fees_handler**, the program performs three separate CPI calls to the SPL Token program's **mint_to** instruction, once for each fee category (**performance_fee**, **management_fee**, and **issue_fee**).

All three operations mint tokens to the **same destination account**, which means three CPIs are being executed sequentially to transfer the same mint into the same account.

This design is functionally correct but suboptimal. Each CPI call incurs additional compute units.

Potential Benefit:

Since all fees are minted to the same destination account, these amounts could be summed before performing a single **mint_to** CPI, reducing runtime cost.

Recommendation:

Compute the total fee amount first, then perform **one** CPI call.

Developer Response:

The three separate CPI calls were consolidated into one.

E05: Adding Token Program Account check will prevent faulty CPI Swap invocations

Location: https://github.com/AdevarLabs/clend-vaults/blob/main/clend-vaults-main/protocol/programs/clend-vaults/src/instructions/swap.rs#L41

>_ swap.rs RUST

```
39:
40:     // check accounts
41:     // first account is token program
42:     // second is authority (the vault)
43:     let authority = accounts[1].pubkey;
```

Description:

The vault manager can swap assets within a vault by invoking the Jupiter program with arbitrary swap data and accounts. While there are several checks in place for **authority**, **source_account**, and **destination_account**, the first **remaining_account** is not checked to be the token program.

Potential Benefit:

Adding this check will remove the possibility of making faulty CPI swap invocation because the incorrect token program account was passed.

Recommendation:

Add a check similar to the other accounts verifying that the first remaining account is the token program. For example:
In the context, add

RUST

```
#[derive(Accounts)]
pub struct Swap<'info> {
  // Other accounts
  pub token_program: Program<'info, Token>
}
```

And in the handler

RUST

```
let token_program = accounts[0].pubkey;
    require!(
        token_program.eq(&ctx.accounts.token_program.key()),
        ClendVaultsError::InvalidTokenProgram
    );
```

See the Jupiter Aggregator v6 IDL for more information about the accounts that are expected in the various instructions.

Developer Response:

Added token program account check.

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.