Pashov Audit Group

Jupiter Security Review

December 10th 2025 - December 17th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Jupiter Stablecoin

Jupiter Stablecoin is an oracle-backed stablecoin system on Solana that enables operators to manage vaults, benefactors, and risk parameters while allowing users to mint and redeem against on-chain collateral. It combines multi-oracle price validation, configurable fees, and rate limits to maintain peg stability.

5. Executive Summary

A time-boxed security review of the **jup-ag/jup-stablecoin** repository was done by Pashov Audit Group, during which **Lukasz**, **ctrus**, **JoVi**, **Koolex** engaged to review **Jupiter Stablecoin**. A total of **12** issues were uncovered.

Protocol Summary	
Project Name	Jupiter Stablecoin
Protocol Type	Stablecoin
Timeline	December 10th 2025 - December 17th 2025

Review commit hash:
- 69d64b83330dee4ef04b8791a028ef40beaf8fce (jup-ag/jup-stablecoin)

Fixes review commit hash:
- 57c2a28b2da30f3ffe1102ce751c1caa09d0c994 (jup-ag/jup-stablecoin)

Scope

`oracle.rs` `lib.rs` `error.rs` `admin.rs` `benefactor.rs` `custodian.rs` `init.rs` `mod.rs` `operator.rs` `user.rs` `vault.rs` `benefactor.rs` `common.rs` `config.rs`

6. Findings

Findings count

Severity	Amount
Low	12
Total findings	**12**

Summary of findings

ID	Title	Severity	Status
[L-01]	Lack of last admin deletion protection	Low	**Resolved**
[L-02]	Operator role can't be cleared once granted.	Low	**Resolved**
[L-03]	No dispersion guard across oracle feeds	Low	**Resolved**
[L-04]	Doves price positivity not enforced early	Low	**Resolved**
[L-05]	Stablecoin can be self-collateralized	Low	**Resolved**
[L-06]	Missing `#[repr(C)]` on nested types in Zero-Copy accounts	Low	**Resolved**
[L-07]	Switchboard on-demand oracle staleness check ineffective after Solana network restart	Low	**Resolved**
[L-08]	Strict default max price limit may lead to temporary DoS	Low	Acknowledged
[L-09]	Global mint is redeemable against any vault	Low	Acknowledged
[L-10]	Switchboard staleness check becomes inaccurate during network congestion	Low	**Resolved**
[L-11]	Token-2022 mints with transfer fee extension cause accounting mismatches	Low	**Resolved**
[L-12]	Inconsistent fee application in mint/redeem oracle vs one-to-one comparison	Low	Acknowledged

Low findings

[L-01] Lack of last admin deletion protection

In `instructions/operator.rs`, `delete_operator` allows any enabled `Admin` to close any `Operator` account passed in `deleted_operator`, including their own operator account and the initial upgrade-authority operator. Nothing prevent deleting the last remaining Admin operator. While highly unlikely to happen, the current design is less fault proof and theoretically allows existence of the state where there is no admins at all.

Recommendation

Consider blocking self-deletion with e.g. `require!(operator.key() != deleted_operator.key())`.

[L-02] Operator role can't be cleared once granted.

Description

The `Operator` struct in `operator.rs` implements a `clear_role` function that allows removing a specific role from an operator's bitmask:

```
impl Operator {
    pub fn set_role(&mut self, role: OperatorRole) {
        self.role |= 1 << role as u64;
    }

    // This function exists but is never called from any instruction
    pub fn clear_role(&mut self, role: OperatorRole) {
        self.role &= !(1 << role as u64);
    }
}
```

However, the `manage_operator` instruction in `operator.rs` only provides `SetRole` action, with no corresponding `ClearRole` action:

```
#[derive(AnchorSerialize, AnchorDeserialize)]
pub enum OperatorManagementAction {
    SetStatus { status: OperatorStatus },
    SetRole { role: OperatorRole },
    // Missing: ClearRole { role: OperatorRole }
}

pub fn manage_operator(
    ctx: Context<ManageOperator>,
    action: OperatorManagementAction,
) -> Result<()> {
    let operator = ctx.accounts.operator.load()?;
    operator.is(OperatorRole::Admin)?;
```

```
    let mut managed_operator = ctx.accounts.managed_operator.load_mut()?;

    match action {
        OperatorManagementAction::SetStatus { status } => {
            managed_operator.status = status;
        },
        OperatorManagementAction::SetRole { role } => {
            managed_operator.set_role(role);  // Can only ADD roles
        },
        // No ClearRole variant exists
    }

    Ok(())
}
```

Which means once a role is granted to an operator, the only way to remove that role is to delete the entire operator account. This can create issues for eg. If an operator has multiple roles and one needs to be revoked, the admin must: - Delete the operator (losing all roles) - Recreate the operator - Re-grant each role individually (except the revoked one)

Recommendation:

Implement `clearRole` varient or if its intended we can remove the `clear-role` unused function.

[L-03] No dispersion guard across oracle feeds

The oracle set accepts the minimum price from provided feeds as long as each is individually fresh and in-range, without checking how far feeds are from each other. A single low-but-stale or errant feed that still passes individual checks can dominate even if others are much higher. Enforce a max spread (or median + deviation) and reject aggregation when dispersion exceeds the threshold.

[L-04] Doves price positivity not enforced early

Negative or zero Doves prices only fail later via math overflow; they pass staleness checks and can DoS flow. Reject non-positive prices before use (as with other oracles) to avoid accepting invalid feeds.

[L-05] Stablecoin can be self-collateralized

The `create_vault` instruction allows the stablecoin mint to be used as collateral. Users could deposit existing LP and mint more tokens recursively. Ensure the vault mint can never be the same as the stablecoin by enforcing `vault.mint != config.mint`.

[L-06] Missing `#[repr(C)]` on nested types in Zero-Copy accounts

Description

The codebase uses Anchor's `#[account(zero_copy)]` for several account structures (`Vault`, `Config`, `Benefactor`, `Operator`, `Pool`). While Anchor automatically applies `#[repr(C)]` to the top-level struct, nested types embedded within these accounts are missing the required `#[repr(C)]` attribute.

Zero-copy deserialization works by directly casting raw account bytes to struct references without copying data. This requires deterministic memory layout, which Rust only guarantees with `#[repr(C)]`.

Affected Code

Enums without `#[repr(C)]` or `#[repr(u8)]`:

```
// vault.rs
pub enum VaultStatus { Enabled, Disabled }
pub enum OracleType { Empty(..), Pyth(..), Doves(..), SwitchboardOnDemand(..) }

// operator.rs
pub enum OperatorStatus { Enabled, Disabled }

// benefactor.rs
pub enum BenefactorStatus { Active, Disabled }

// pool.rs
pub enum PoolStatus { Active, Paused, Disabled }
```

Structs without `#[repr(C)]`:

```
// common.rs
pub struct PeriodLimit { /* 6 fields */ }

// vault.rs
pub struct PythV2Oracle { /* 4 fields */ }
pub struct SwitchboardOnDemandOracle { /* 4 fields */ }
pub struct DovesOracle { /* 4 fields */ }
pub struct EmptyOracle { /* 4 fields */ }
```

Recommendation

Add explicit representation attributes to all nested types:

```
// Simple enums - use #[repr(u8)]
#[repr(u8)]
pub enum VaultStatus { Enabled = 0, Disabled = 1 }

// Enums with data - #[repr(C, u8)]
```

```
#[repr(C, u8)]
pub enum OracleType { ... }

// Structs - use #[repr(C)]
#[repr(C)]
pub struct PeriodLimit { ... }
```

References

- Anchor Zero Copy Documentation
- bytemuck Pod trait requirements
- Rust Reference - Type Layout

[L-07] Switchboard on-demand oracle staleness check ineffective after Solana network restart

Severity

Impact: Medium

Likelihood: Low

Description

Solana mainnet has experienced multiple outages requiring coordinated restarts, with some lasting several hours. During these restarts, validators restore state from a previous slot snapshot. The slot number continues sequentially from the snapshot slot, and oracle account data is also restored from the snapshot.

Switchboard On-Demand's `get_value` function uses slot-based staleness checking. Per the official documentation:

> "Returns the median value of the submissions in the last `max_staleness` slots."

Note: JupStable uses 0.9.3 and the docs refers to 0.1.14. but in this case, it is irrelevant

JupStable calls this function in `oracle.rs`:

```
let price = price_feed
    .get_value(
        clock.slot,
        stalesness_threshold * 1000 / clock::DEFAULT_MS_PER_SLOT,  // converts 300 seconds →
~750 slots
        1,
        true,
    )
```

After a network restart: - `clock.slot` continues from the snapshot (e.g., slot 100,000,000) - The oracle's last update slot is also from the snapshot (e.g., slot 99,999,500) - The slot difference (500 slots) is within the ~750 slot threshold - The staleness check passes despite hours of real-world time having elapsed

The existing price bounds (`min_oracle_price_usd: 5000` , `max_oracle_price_usd: 10000` , representing $0.50-$1.00) limit exploitation to scenarios where the price remained within this range during the outage but has since moved.

Recommendations

Use the `LastRestartSlot` sysvar to detect post-restart scenarios and reject oracle data if its last update slot predates the most recent restart.

[L-08] Strict default max price limit may lead to temporary DoS

The vault is created by copying `..Default::default()` into its on-chain state, and the default Vault sets `max_oracle_price_usd: 10000` which corresponds to a hard ceiling of 1.0000. In normal market conditions, it is normal for a healthy stablecoin to fluctuate between +/- 0.1% or more, however this limit would prevent that. In `programs/jup-stable/src/state/vault.rs` :

```
impl Default for Vault {
    fn default() -> Self {
        Vault {
            mint: Pubkey::default(),
            custodian: Pubkey::default(),
            token_account: Pubkey::default(),
            token_program: Pubkey::default(),
            stalesness_threshold: 300,
            min_oracle_price_usd: 5000,
            max_oracle_price_usd: 10000,
            status: VaultStatus::Disabled,
            _padding1: [0; 7],
            bump: 0,
            decimals: 0,
            _padding2: [0; 6],
            custodian_deposited: [0; 16],
            custodian_withdrawn: [0; 16],
            oracles: [OracleType::Empty(Default::default()); MAX_ORACLES],
            _padding3: [0; 3],
            period_limits: [PeriodLimit::default(); MAX_PERIOD_LIMIT],
            total_minted: [0; 16],
            total_redeemed: [0; 16],
            reserved: [0; 256],
        }
    }
}
```

If the default configuration is applied without a change, user mints and redeems will revert, until an operator updates the bounds.

Recommendations

Require explicit min/max oracle bounds during vault setup or set defaults wide enough for expected premium/discount.

Jupiter comments

The default configuration will obviously not be used in production. The program enforces a configuration update before the vault can be enabled.

[L-09] Global mint is redeemable against any vault

All vaults share one global mint and the protocol doesn't record which vault collateralized a given token unit. If a collateral depegs or is mispriced, a benefactor can mint the stablecoin against that impaired vault, then redeem the same mint from a healthier vault holding a different asset. The bad-collateral vault stays insolvent while the good vault pays out until empty, turning a single-vault loss into cross-vault collateral theft.

Recommendations

Make sure to track liabilities per vault and enforce redeem-from-issuance (or a routing mechanism that debits the originating vault) so the global mint cannot be redeemed against unrelated collateral.

Jupiter comments

This is intentional. The program will only support stablecoins as collateral. Thanks to how we compute minting and redeeming amounts, the price used will be at most 1:1 (minus the benefactor fee). If a collateral depegs, the min_oracle_price / max_oracle_price limits will take effect. We also have monitoring tools in place to pause minting and redeeming in case of exceptional market events.

[L-10] Switchboard staleness check becomes inaccurate during network congestion

In `oracle.rs`, the Switchboard staleness threshold is converted from seconds to slots using a fixed 400ms slot time:

```
let price = price_feed
    .get_value(
        clock.slot,
        stalesness_threshold * 1000 / clock::DEFAULT_MS_PER_SLOT,  // 400ms constant
        1,
        true,
    )
```

During network congestion, actual slot times can exceed 600-1000ms. With `stalesness_threshold = 300` seconds: - Calculated max slots: `300 * 1000 / 400 = 750 slots` - Normal network (400ms/slot): 750 slots ≈ 300 seconds ✓ - Congested network (800ms/slot): 750 slots ≈ 600 seconds (twice as stale as intended)

This allows staler prices than intended to pass validation during periods of network congestion, which often coincide with market volatility.

Recommendations

Add an additional timestamp-based staleness check using the `last_update_timestamp` field available in `PullFeedAccountData`. This ensures the staleness threshold is enforced accurately regardless of network conditions, while maintaining the existing slot-based validation.

[L-11] Token-2022 mints with transfer fee extension cause accounting mismatches

The protocol allows creating vaults for stablecoin collateral via the `create_vault` instruction and explicitly supports Token-2022 by accepting `TokenInterface`. However, the protocol does not validate against the transfer fee extension.

It's worth noting that major stablecoins like USDC and USDT on EVM have transfer fee functionality built into their contracts (currently disabled but can be enabled by the issuer). Given this precedent, it's plausible that Solana stablecoin issuers could adopt similar mechanisms using Token-2022's native transfer fee extension, either at launch or enabled later.

When Token-2022 tokens with the transfer fee extension are transferred using `transfer_checked`, fees are automatically withheld from the recipient's account—the full amount leaves the sender, but the recipient can only access `(amount - fee)`. The protocol records the input amount rather than the actual usable amount received.

In `user.rs`, the `mint` function transfers collateral to the vault:

```
transfer_checked(
    ctx.accounts.deposit_collateral(),
    amount,
    ctx.accounts.vault_mint.decimals,
)?;
```

The protocol then mints LP tokens based on the original `amount`, but the vault only receives `(amount - transfer_fee)` in usable funds. Similarly, in `redeem`, the vault transfers collateral to users based on calculated amounts, but users receive less due to withheld fees.

This creates systematic accounting mismatches: - **Minting:** Users receive LP tokens representing more collateral than actually deposited in usable form - **Redemption:** Users receive less collateral than expected, or vault becomes insolvent when recorded balances exceed actual usable balances

The withheld fees belong to the token's fee authority and are not recoverable by the protocol, leading to gradual undercollateralization of the LP token supply.

Recommendations

Since transfer fees can be enabled on existing tokens after vault creation, validating at vault creation time is insufficient. Instead, measure actual balance changes before and after transfers:

```
pub fn mint(ctx: Context<Mint>, amount: u64, min_amount_out: u64) -> Result<()> {
    require!(amount > 0, JupStableError::ZeroAmount);

    // Capture balance before transfer
    let vault_balance_before = ctx.accounts.vault_token_account.amount;

    transfer_checked(
        ctx.accounts.deposit_collateral(),
        amount,
        ctx.accounts.vault_mint.decimals,
    )?;

    // Reload account and calculate actual received amount
    ctx.accounts.vault_token_account.reload()?;
    let actual_received = ctx.accounts.vault_token_account.amount
        .checked_sub(vault_balance_before)
        .ok_or(JupStableError::MathOverflow)?;

    // Use actual_received for all subsequent calculations
    // ...
}
```

This pattern ensures accurate accounting regardless of whether transfer fees are enabled at vault creation or activated later.

[L-12] Inconsistent fee application in mint/redeem oracle vs one-to-one comparison

Both `mint` and `redeem` calculate two amounts and take the minimum, but fee is applied inconsistently:

- `one_to_one_amount` uses `net_amount` (after fee)
- `oracle_amount` uses `amount` (before fee)

```
// In compute_mint_amount and compute_redeem_amount:
let one_to_one_amount = /* uses net_amount (after fee) */
let oracle_amount = /* uses amount (before fee) */

let result = oracle_amount.min(one_to_one_amount);
```

When oracle amount wins the comparison, protocol collects less fee than intended.

Mint example (oracle < peg):

User (Benefactor) deposits 10,000 USDC with 5% fee and oracle at $0.90:

Current behavior:

- one_to_one_amount = 9,500 JUPUSD (fee applied)
- oracle_amount = 10,000 × 0.90 = 9,000 JUPUSD (no fee)
- mint_amount = 9,000 JUPUSD

Correct behavior:

- oracle_amount = 9,500 × 0.90 = 8,550 JUPUSD
- mint_amount = 8,550 JUPUSD

Impact: User receives 450 extra JUPUSD, protocol loses 450 JUPUSD worth of fee.

Redeem example (oracle > peg):

User redeems 10,000 JUPUSD with 5% fee and oracle at $1.10:

Current behavior:

- one_to_one_amount = 9,500 USDC (fee applied)
- oracle_amount = 10,000 ÷ 1.10 = 9,090 USDC (no fee)
- redeem_amount = 9,090 USDC

Correct behavior:

- oracle_amount = 9,500 ÷ 1.10 = 8,636 USDC
- redeem_amount = 8,636 USDC

Impact: User receives 454 extra USDC, protocol loses 454 USDC worth of fee.

Recommendations

Use `net_amount` for both calculations:

```
let oracle_amount = calculate_mint_amount(
    oracle_price,
    Decimal::new(net_amount.try_into()?, vault_decimals),
```

```
    peg_price,
    lp_decimals,
)?;
```

Jupiter comments

The term fee in the code may be misleading. These "fees" are not really fees, they are used to overcollateralize the protocol. The business logic for computing minting and redeeming amounts is copied from the Ethena smart contract, our partner in building jupUSD. “Fees” are only applied when the oracle price is better than the 1:1 rate minus the benefactor fee rate. This ensures that benefactors cannot mint or redeem at more than 1:1 (minus fees). If the oracle price is below that rate, we use the oracle price and do not "charge" any fees.