Pashov Audit Group

Palm USD Security Review

December 26th 2025 - December 30th 2025

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Risk Classification 3
4. About Palm USD Rust 4
5. Executive Summary 4
6. Findings 5
Low findings 6
[L-01] Redundant validation checks waste compute units across multiple functions 6
[L-02] Protocol can remain open without any active owner for 24 hours 7
[L-03] Upgrade authority check is unnecessary and wastes compute units 8
[L-04] Protocol can not be paused 9
[L-05] Lack of flexible account sizing for future upgrades 10
[L-06] Missing toolchain version in `Anchor.toml` 10
[L-07] Unused code 10
[L-08] `has_role` does not account for role activation time 10
[L-09] Freeze authority is not managed 11
[L-10] Missing mint validation allows arbitrary minting via PDA authority 11
[L-11] Owner functions can be bricked permanently 12
[L-12] Unrestricted minting enables infinite token supply & inflation 13
[L-13] Token metadata script embeds gateway tokens in on-chain URL 14

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 Palm USD Rust

Palm USD is a multi-chain USD-pegged stablecoin implemented across Ethereum, EVM-compatible chains, Tron, and Solana. It supports cross-chain bridge capabilities through authorized contracts (like LayerZero) and includes controlled minting/burning operations managed by operators and authorized contracts, with versions ranging from V1 to V3.

5. Executive Summary

A time-boxed security review of the **palmfidev/PUSD-core-contract** repository was done by Pashov Audit Group, during which **ctrus**, **Nyx**, **JoVi**, **Lukasz** engaged to review **Palm USD Rust**. A total of **13** issues were uncovered.

Protocol Summary	
Project Name	Palm USD Rust
Protocol Type	Crosschain stablecoin
Timeline	December 26th 2025 - December 30th 2025

Review commit hash:
- e7e37482c0b0bb3d46df36ac165bd0e8c22e1775 (palmfidev/PUSD-core-contract)

Fixes review commit hash:
- 49c049b0d6c7e597b7a4b51d854b750372a6ba9d (palmfidev/PUSD-core-contract)

Scope

`constants.rs` `errors.rs` `lib.rs` `modifiers.rs` `state.rs`

6. Findings

Findings count

Severity	Amount
Low	13
Total findings	**13**

Summary of findings

ID	Title	Severity	Status
[L-01]	Redundant validation checks waste compute units across multiple functions	Low	**Resolved**
[L-02]	Protocol can remain open without any active owner for 24 hours	Low	**Resolved**
[L-03]	Upgrade authority check is unnecessary and wastes compute units	Low	**Resolved**
[L-04]	Protocol can not be paused	Low	Acknowledged
[L-05]	Lack of flexible account sizing for future upgrades	Low	**Resolved**
[L-06]	Missing toolchain version in `Anchor.toml`	Low	**Resolved**
[L-07]	Unused code	Low	**Resolved**
[L-08]	`has_role` does not account for role activation time	Low	**Resolved**
[L-09]	Freeze authority is not managed	Low	Acknowledged
[L-10]	Missing mint validation allows arbitrary minting via PDA authority	Low	**Resolved**
[L-11]	Owner functions can be bricked permanently	Low	**Resolved**
[L-12]	Unrestricted minting enables infinite token supply & inflation	Low	Acknowledged
[L-13]	Token metadata script embeds gateway tokens in on-chain URL	Low	**Resolved**

Low findings

[L-01] Redundant validation checks waste compute units across multiple functions

Multiple functions contain redundant validation checks that are already enforced by Anchor's account constraints at the instruction struct level. These checks waste compute units (approximately 100-500 CUs per redundant check) on every function invocation.

Instance 1: Redundant `require_initialized!` Checks

Affected Functions - `add_role()` - `remove_role()` - `transfer_role()`

Redundant Code

```
// In add_role()
pub fn add_role(ctx: Context<AddRole>, user: Pubkey, role: Role) -> Result<()> {
    require_initialized!(ctx.accounts.program_state);  // REDUNDANT
    // ...
}

// In remove_role()
pub fn remove_role(ctx: Context<RemoveRole>) -> Result<()> {
    require_initialized!(ctx.accounts.program_state);  // REDUNDANT
    // ...
}

// In transfer_role()
pub fn transfer_role(ctx: Context<TransferRole>, user: Pubkey, new_role: Role) -> Result<()> {
    require_initialized!(ctx.accounts.program_state);  // REDUNDANT
    // ...
}
```

Why It's Redundant

The account constraint already validates `program_state` exists and has correct bump:

```
#[derive(Accounts)]
pub struct AddRole<'info> {
    #[account(seeds = [b"program_state"], bump = program_state.bump)]
    pub program_state: Account<'info, ProgramState>,  // Must exist to pass
    // ...
}
```

If `program_state` account doesn't exist (not initialized), the PDA derivation fails, and the transaction reverts **before** the function body executes. Checking `is_initialized` after this is redundant.

Instance 2: Redundant `constraint` Checks for Owner role in context structs Redundant Code

```
#[derive(Accounts)]
pub struct AddRole<'info> {
    #[account(
        seeds = [b"user_role", owner.key().as_ref()],
        bump = owner_role.bump,
        constraint = owner_role.role == Role::Owner @ PusdError::Unauthorized  // Redundant
    )]
    pub owner_role: Account<'info, UserRole>,
    // ...
}
```

Affected Functions - `add_role()` - `remove_role()` - `transfer_role()`

```
// In add_role()
pub fn add_role(ctx: Context<AddRole>, user: Pubkey, role: Role) -> Result<()> {
    // ...
    require_role!(ctx.accounts.owner_role, Role::Owner);  // already checks role + activation
time has passed, so contraints are redundant
    // ...
}

// In remove_role()
pub fn remove_role(ctx: Context<RemoveRole>) -> Result<()> {
    // ...
    require_role!(ctx.accounts.owner_role, Role::Owner);  // already checks role + activation
time has passed, so contraints are redundant
    // ...
}

// In transfer_role()
pub fn transfer_role(ctx: Context<TransferRole>, user: Pubkey, new_role: Role) -> Result<()> {
    // ...
    require_role!(ctx.accounts.owner_role, Role::Owner);  // already checks role + activation
time has passed, so contraints are redundant
    // ...
}
```

[L-02] Protocol can remain open without any active owner for 24 hours

The `remove_role()` function allows an Owner to remove any role, including their own. Since PDAs are derived from user addresses ([seeds = [b"user_role", user.as_ref()]]), an Owner can pass their own role account as `user_role` and successfully close it. This creates a critical governance gap where the protocol can be left without any active Owner for up to 24 hours(If owner had nominated another owner prior) or no owner at all if there was only one owner and owner did'nt nominate any other address as owner before removing himself.

```
pub fn remove_role(ctx: Context<RemoveRole>) -> Result<()> {
    require_initialized!(ctx.accounts.program_state);
    require_role!(ctx.accounts.owner_role, Role::Owner);

    // NO CHECK preventing owner from removing themselves
```

```
    // owner_role and user_role can be the SAME account

    msg!(
        "Removing role {:?} for user: {}",
        ctx.accounts.user_role.role,
        ctx.accounts.user_role.user
    );

    // Account closed - if this was the owner's own role, they just removed themselves
    Ok(())
}

#[derive(Accounts)]
pub struct RemoveRole<'info> {
    #[account(
        seeds = [b"user_role", owner.key().as_ref()],
        bump = owner_role.bump,
        constraint = owner_role.role == Role::Owner @ PusdError::Unauthorized
    )]
    pub owner_role: Account<'info, UserRole>,  // Used for authorization

    #[account(mut)]
    pub owner: Signer<'info>,

    #[account(
        mut,
        close = owner,
        seeds = [b"user_role", user_role.user.as_ref()],  // Can match owner's PDA!
        bump = user_role.bump
    )]
    pub user_role: Account<'info, UserRole>,  // Role being removed
}
```

Impact - If Owner removes self after nominating a successor, protocol has no active admin for 24 hours. - If sole Owner removes self without a successor, protocol is permanently bricked. Note: similarly in `transfer-role`, make sure at least one owner is left after transferring the role as `Operator` role is activated immediately, is sole owner transfers the role to operator, no active owner remains.

Recommendations Prevent the owner from removing themselves in `remove-role`.

[L-03] Upgrade authority check is unnecessary and wastes compute units

In the `[require_upgrade_authority!]` macro, the check `[upgrade_authority_option != 1]` is redundant. The BPF Upgradeable Loader's ProgramData account layout ensures that if `[upgrade_authority]` is None, the account data would be shorter than 45 bytes, which is already caught by the preceding length check.

```
#[macro_export]
macro_rules! require_upgrade_authority {
    ($program_data:expr, $payer:expr) => {{
        let data = $program_data.try_borrow_data()?;
```

```
        // CHECK 1: Already catches None case, if the upgrade authority owould have been `None`
the length would coome out to be <45, so follow up check is redundant
        if data.len() < 45 {
            return Err(PusdError::InvalidProgramData.into());
        }

        let upgrade_authority_option = data[4 + 8];

        // CHECK 2: REDUNDANT - If option was None (0), data.len() < 45 would have failed
        if upgrade_authority_option != 1 {
            return Err(PusdError::OnlyUpgradeAuthority.into());
        }
    }};
}
```

Layout of `ProgramData`:

Offset 0 (size 4): Account type discriminator

Offset 4 (size 8): Slot (last upgrade)

Offset 12 (size 1): Option discriminant (0 = None, 1 = Some)

Offset 13 (size 32): Upgrade authority pubkey (if Some)

Recommendation

Remove the check as the length check already makes sure the upgrade authority is some.

[L-04] Protocol can not be paused

We currently don't have any emergency pause functionality, leaving no mechanism to halt critical operations (minting, role management) during security incidents, exploits, or market emergencies.

The `[ProgramState]` only tracks initialization status with no pause flag:

```
#[account]
pub struct ProgramState {
    pub is_initialized: bool, // 1 byte
    pub bump: u8,             // 1 byte
    // Missing: pub is_paused: bool
}
```

Rest of the functionalities don't have any pause checks. This missing pause mechanism exposes the protocol to unmitigated risk during security incidents.

Recommendation Add pausing functionality & store pause status in `ProgramState`

[L-05] Lack of flexible account sizing for future upgrades

The protocol currently uses hardcoded account sizes without proper checks to ensure accounts have sufficient space for future updates. This lack of flexibility may complicate future upgrades to account structures.

Recommendation

Add padding space to the Program Derived Accounts (PDAs) to allow for future protocol updates. This will give the protocol the flexibility to handle changes without the need for complex migrations or reallocations later on.

[L-06] Missing toolchain version in `Anchor.toml`

The Anchor.toml file does not specify the anchor_version or solana_version under the [toolchain] section. This can lead to compatibility issues when building or deploying the program, especially if different team members or CI/CD pipelines use different versions of Anchor or Solana.

[L-07] Unused code

The codebase contains several unused code artifacts that reduce maintainability:

- Unused `UpgradeableData` struct,
- Error `RoleNotActiveYet`,
- Error `GrantRoleFailed`.

The `UpgradeableData` struct seems to be intended for parsing the BPF Loader Upgradeable program data account, but the `require_upgrade_authority!` macro manually parses raw bytes instead.

Two error variants, `RoleNotActiveYet` and `GrantRoleFailed` are defined but never returned by any code path, with `RoleNotActivated` being used instead of the former.

Recommendation Remove unused code or implement it in the corresponding functions.

[L-08] `has_role` does not account for role activation time

The `has_role` instruction checks only whether a `UserRole` account stores a given role enum, but does not verify whether the role is currently active based on `role_active_time`.

For offchain integrations or external programs that rely on `has_role` as an authorization signal, this function may report a role as present even if it is still within the activation delay window. If such integrations treat a true return value as proof of active authorization, they may prematurely bypass intended timelocks.

Recommendation Account for the activation time before returning `true`.

[L-09] Freeze authority is not managed

The program transfers only mint authority to the PDA, but does not set or revoke freeze authority on the mint. If the freeze authority remains with the deployer, that key can freeze user token accounts despite role changes.

Recommendation

Manage Freeze Authority as well, or explicitly document that is skipped on purpose.

[L-10] Missing mint validation allows arbitrary minting via PDA authority

The PUSD program never stores or validates which Token-2022 mint it controls. The `mint` account in both `MintByContract` and `MintByOperator` contexts is an unchecked AccountInfo with no constraint verifying it matches an expected canonical mint address.

```
#[derive(Accounts)]
pub struct MintByContract<'info> {
    // ...

    /// The Token-2022 mint account
    /// CHECK: Validated by Token-2022 program  // <--  Only validates it's A mint
    #[account(mut)]
    pub mint: AccountInfo<'info>,  // <-- No canonical address check!

    // ...
}

#[derive(Accounts)]
pub struct MintByOperator<'info> {
    // ...

    /// CHECK: Validated by Token-2022 program
    #[account(mut)]
    pub mint: AccountInfo<'info>,  // <-- Same issue

    // ...
}
```

The program's mint authority PDA ([seeds = [b"mint_authority"]]) is deterministic and publicly derivable. Any authorized external actor can:

- Create a new Token-2022 mint.
- Set that mint's authority to the PUSD program's mint authority PDA.
- Call `mint()` or `mint_by_operator()` with this fake mint.
- The CPI to Token-2022 will succeed because the program's PDA is indeed the authority for this attacker-controlled mint.

If there is some reward mechanism offchain based on amount of pusd minted(or if protocols implement this in future) this can be exploited by minting fake tokens and receiving reward pusd based on fake mint's amount.

Recommendations Store canonical mint in `programState` and verify it via anchor constraints in instruction contexts.

[L-11] Owner functions can be bricked permanently

During `initialize()`, if `owner_address` and `operator_address` are the same, both role accounts resolve to the same PDA (derived from [[b"user_role", address.as_ref()]]. The second `_grant_role()` call for `Operator` overwrites the first `Owner` role assignment, leaving the protocol with no Owner and permanently disabling all administrative functions.

Inside `initialize()`

```
pub fn initialize(
    // Only validates non-zero, NOT that they differ
    require_valid_address!(owner_address);
    require_valid_address!(operator_address);
    // MISSING: require!(owner_address != operator_address, PusdError::SameAddress);

    // First call: Sets PDA to Owner role
    _grant_role(
        Role::Owner,
    )?;

    // Second call: If same address, OVERWRITES the same PDA to Operator!
    _grant_role(
        Role::Operator,      // <-- Owner role is now lost
    )?;
    // ...
}
```

Since we are using `init-if-needed` (as it does not revert if the account already exists, and overwrites state silently) and not making sure explicitly that both pubkeys are different, this issue can manifest.

Impact

- All administrative functions are disabled, like `add_role`, `remove_role`, `transfer-role`.. new operators/authorized contracts can't be added, roles can not be transferred, protocol is stuck with no owner role to manage access control.

Recommendations

Add explicit validation that addresses differ:

```
pub fn initialize(
    ctx: Context<Initialize>,
    owner_address: Pubkey,
    operator_address: Pubkey,
```

```
) -> Result<()> {
    require_upgrade_authority!(ctx.accounts.program_data, ctx.accounts.payer.key());

    let program_state = &mut ctx.accounts.program_state;
    require_not_initialized!(program_state);

    require_valid_address!(owner_address);
    require_valid_address!(operator_address);

    // ADD: Ensure owner and operator are different addresses
    require!(
        owner_address != operator_address,
        PusdError::OwnerOperatorSameAddress
    );

    // ... rest of function
}
```

[L-12] Unrestricted minting enables infinite token supply & inflation

The `mint()` and `mint_by_operator()` functions lack any supply controls - no per-transaction limits, daily caps, total supply ceiling, or rate limiting. Combined with the absence of an emergency pause mechanism, a compromised or malicious AuthorizedContract/Operator can mint unlimited tokens in a single transaction, causing irreversible hyperinflation and complete stablecoin depegging.

```
pub fn mint(ctx: Context<MintByContract>, amount: u64) -> Result<()> {
    require_role!(ctx.accounts.contract_role, Role::AuthorizedContract);

    require!(
        ctx.accounts.recipient.key() != Pubkey::default(),
        PusdError::RecipientIsZeroAddress
    );

    // Only checks amount > 0, NO upper bound!
    require!(amount > 0, PusdError::AmountIsZero);

    // Can mint u64::MAX (18,446,744,073,709,551,615) in single call
    _mint(/* ... */, amount)?;

    Ok(())
}
```

Impact

- No `AuthorizedContract` can be trusted: Even legitimate DeFi protocols can be compromised.
- Once one contract is exploited, entire PUSD ecosystem collapses.
- Other protocols integrating PUSD suffer collateral damage.
- Stablecoin price crashes to effectively $0.
- All PUSD holders lose 100% of value.

Example scenario:

- AuthorizedContract calls mint(u64::MAX)
  └── 18,446,744,073,709,551,615 PUSD created
- Attacker swaps PUSD → SOL on Raydium
  └── Crashes PUSD price, extracts SOL
- Attacker swaps PUSD → USDC on Orca
  └── Further dumps, extracts USDC

PUSD price has gone to zero, and attacker extracts all profit from usdc/sol.

Recommendations

Implement comprehensive minting controls: daily limit, weekly limit, max supply etc. Both PUSD specific and `AuthorizedContract` specific

[L-13] Token metadata script embeds gateway tokens in on-chain URL

The token creation script constructs the metadata URL by embedding a Pinata gateway token. On-chain metadata is public, so any embedded token becomes permanently exposed. If the token grants billing, access control, or upload privileges, exposure can lead to unauthorized use, cost abuse, or content tampering.

Recommendations

Avoid embedding secrets in on-chain metadata URLs. Use public gateways or an authenticated proxy that keeps credentials off-chain, and add script checks that reject URLs containing credential-like query parameters.