Pashov Audit Group

Elixir
Security Review

August 17th 2025 - August 22nd 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 Elixir

Elixir is a Sui-based project that lets users stake deUSD tokens to earn yield distributed from protocol-generated rewards. It uses smart contracts to handle deposits, cooldown periods, reward vesting, and automated distribution through a dedicated rewards distributor.

5. Executive Summary

A time-boxed security review of the **ElixirProtocol/move-contracts-v2** repository was done by Pashov Audit Group, during which **0xTheBlackPanther**, **Lucasz**, **shaflow** engaged to review **Elixir**. A total of **13** issues were uncovered.

Protocol Summary	
Project Name	Elixir
Protocol Type	Staking vault
Timeline	August 17th 2025 - August 22nd 2025

Review commit hash:

- 594a240937946e77e72d03a73a2e8001580ee78c (ElixirProtocol/move-contracts-v2)

Fixes review commit hash:

- 301af9582aa2ced9de992858aa0f0213eda71dd5 (ElixirProtocol/move-contracts-v2)

Scope

`acl.move` `admin_cap.move` `config.move` `deusd.move` `deusd_lp_staking.move` `deusd_minting.move` `locked_funds.move` `roles.move` `sdeusd.move` `staking_rewards_distributor.move` `clock_utils.move` `cryptography.move` `math_u64.move` `set.move`

6. Findings

Findings count

Severity	Amount
High	1
Medium	2
Low	10
Total findings	**13**

Summary of findings

ID	Title	Severity	Status
[H-01]	Incorrect timestamp comparison leads to cooldown bypass	High	**Resolved**
[M-01]	Inconsistencies in Blacklisting logic	Medium	**Resolved**
[M-02]	Orphaned rewards captured by first staker	Medium	**Resolved**
[L-01]	Incomplete storage cleanup on full withdrawal	Low	**Resolved**
[L-02]	Faulty logic allows duplicate custodian addition	Low	**Resolved**
[L-03]	Ghost members in ACL causing potential storage bloat	Low	**Resolved**
[L-04]	Missing role bounds check in `has_role`	Low	**Resolved**
[L-05]	Unavailable view function	Low	**Resolved**
[L-06]	The `deposit` function lacks asset support check	Low	**Resolved**
[L-07]	Missing the function to transfer the `AdminCap` object	Low	**Resolved**
[L-08]	Soft restriction bypass via token transfers	Low	Acknowledged
[L-09]	Cooldown timer reset causes user withdrawal delays	Low	Acknowledged
[L-10]	Cooldown duration changes apply retroactively	Low	Acknowledged

High findings

[H-01] Incorrect timestamp comparison leads to cooldown bypass

Severity

Impact: Medium

Likelihood: High

Description

In `sdeusd.move`, unstake uses `clock::timestamp_ms(clock)` (milliseconds) to compare against `cooldown_end`, which is set using `clock_utils::timestamp_seconds(clock)` (seconds). As a result, the check:

```
assert!(current_time >= cooldown.cooldown_end || management.cooldown_duration == 0,
EInvalidCooldown);
```

always passes, since current_time (ms) is ~1000× larger than cooldown_end (s).

Code in `update_user_cooldown`:

```
fun update_user_cooldown(
    management: &mut SdeUSDManagement,
    user: address,
    amount: u64,
    clock: &Clock,
) {
    if (management.cooldowns.contains(user)) {
        let cooldown = management.cooldowns.borrow_mut(user);
        cooldown.cooldown_end = clock_utils::timestamp_seconds(clock) +
management.cooldown_duration;
        cooldown.underlying_amount = cooldown.underlying_amount + amount;
    } else {
        management.cooldowns.add(user, UserCooldown {
            cooldown_end: clock_utils::timestamp_seconds(clock) + management.cooldown_duration,
            underlying_amount: amount,
        })
    };
}
```

Later in the `unstake` function:

```
[...]
    let current_time = clock::timestamp_ms(clock);
    assert!(
        current_time >= cooldown.cooldown_end || management.cooldown_duration ==
0,//@audit-high compared to seconds ??
```

```
        EInvalidCooldown
    );
[...]
```

Finding a full description.

Recommendations

Perform the comparison using `clock_utils::timestamp_seconds` in unstake.

Medium findings

[M-01] Inconsistencies in Blacklisting logic

Severity

Impact: Medium

Likelihood: Medium

Description

The implementation of full/soft restrictions in `sdeusd.move` shows some inconsistencies that may allow bypasses of it. The full details are described below.

Finding a full description. Full restrictions include adding user to an `sdeUSD` deny list, which prevents him from operating those coins via the native mechanism. There is also a soft restriction which is meant to prevent access to some of protocol features.

- In functions `deposit` and `mint`, there is only a check against `is_soft_restricted_staker`. Then sdeUSD is transferred to the `receiver`. However, if a restricted user points out another `receiver` address, they may still be able to stake, as the sdeUSD will be transferred to a non-blocked address.
- For the deposit/mint functions, newly minted sdeUSD can still be sent to an address that is fully blacklisted in the current epoch. This is because the restriction on receiving tokens only takes effect in the following epoch.
- Similar case might also happen with `unstake`, if the user becomes restricted after the cooldown is initiated, as the last check for full restrictions happens in `withdraw_to_silo`, then that user will still be able to unstake the funds and exit the protocol
- Finally, since full restriction implements all features of a soft restriction + extra restrictions, then `is_soft_restricted_staker` could potentially invoke `is_full_restricted_staker` inside, as all full restricted stakers are also soft restricted at the same time (but not all soft restricted are full restricted).

Recommendations

Consider implementing additional restriction checks according to above points.

[M-02] Orphaned rewards captured by first staker

Severity

Impact: High

Likelihood: Low

Description

In contract `sdeusd.move` func `transfer_in_rewards()` and `convert_to_shares()` when rewards are distributed while no sdeUSD holders exist, the first subsequent staker captures all orphaned rewards at 1:1 conversion rate.

If you check the **vulnerable logic** it doesn't check if there are active stakers.

```
// transfer_in_rewards() - No check for active stakers
public fun transfer_in_rewards(...) {
    // Missing: assert!(total_supply(management) > 0, ENoActiveStakers);
    update_vesting_amount(management, amount, clock);
}

// convert_to_shares() - 1:1 ratio when no existing stakers
if (total_supply == 0 || total_assets == 0) {
    assets  // First staker gets 1:1 regardless of unvested rewards
}
```

Let's take an example **how it occurs** 1. Protocol has zero sdeUSD holders (total_supply = 0). 2. 100 deUSD rewards distributed via `transfer_in_rewards()` . 3. Alice stakes 1000 deUSD during the vesting period → gets 1000 shares (1:1 ratio). 4. Vesting completes → Alice's 1000 shares now represent 1100 deUSD total assets. 5. Alice redeems for 100 deUSD profit (orphaned community rewards).

No validation prevents reward distribution to an empty staker pool, combined with 1:1 conversion, ignoring unvested assets.

Impact

First staker after empty periods captures all orphaned community rewards through timing manipulation.

Recommendation

Prevent reward distribution when no active stakers exist.

```
public fun transfer_in_rewards(...) {
    assert!(total_supply(management) > 0, ENoActiveStakers);
    update_vesting_amount(management, amount, clock);
    // ... rest of function
}
```

Low findings

[L-01] Incomplete storage cleanup on full withdrawal

In the `locked_funds.move` function `update_user_collateral_coin_types()` when users withdraw all tokens of a specific type, the tracking list is updated, but the underlying `BalanceStore<T>` dynamic field entry is not removed from storage.

This is **how it occurs**: 1. User deposits ETH → creates `BalanceStore<ETH>` dynamic field in deposit calling `get_or_create_balance_store_mut`. 2. When the user withdraws all ETH → balance becomes 0. 3. `update_user_collateral_coin_types()` removes ETH from the tracking list. 4. `BalanceStore<ETH>` with zero balance remains in storage permanently.

The **root cause** is missing cleanup logic, only tracking is updated, and actual storage persists.

One concrete impact of this is **gradual storage bloat**, as empty balance entries accumulate over time without cleanup.

As recommendation, add a dynamic field removal when the balance reaches zero. In the `update_user_collateral_coin_types` else part, add the code below:

```
} else {
    // Current: remove from tracking
    coin_types.remove(&coin_type);

    // ADD: remove empty storage
    let balance_key = get_balance_store_key<T>(owner);
    if (df::exists_(&management.id, balance_key)) {
        df::remove<vector<u8>, BalanceStore<T>>(&mut management.id, balance_key);
    }
}
```

[L-02] Faulty logic allows duplicate custodian addition

In `deusd_minting.move` function `add_custodian_address_internal()`, the incorrect assertion logic fails to prevent duplicate custodian addresses from being added.

If you check the below code, it uses OR operator

```
assert!(custodian != @0x0 || management.custodian_addresses.contains(custodian),
EInvalidCustodianAddress);
```

The OR condition passes when the custodian already exists, opposite of the intended behavior.

Let's take an example of **how it occurs**: 1. Admin adds custodian @alice successfully. 2. Admin adds @alice again. 3. Assertion evaluates: `(@alice != @0x0) || (contains(@alice))` = `true || true` = `true`. 4. Validation passes when it should reject the duplicate.

It happened because of a wrong logical operator and a missing negation, it should have rejected when the address already exists, not accepted.

The impact of it is that duplicate custodians are accepted instead of a clear error, causing confusion and potential storage waste.

As a recommendation, replace with proper validation logic:

```
assert!(custodian != @0x0, EZeroAddress);
assert!(!management.custodian_addresses.contains(custodian), EDuplicateCustodian);
```

[L-03] Ghost members in ACL causing potential storage bloat

The bug occurs in `remove_role` and `set_roles` functions in acl.move, where members with all roles revoked (roles=0) remain in the `roles_by_member` LinkedTable without automatic removal, leading to accumulation of useless entries via repeated add/clear operations; this bloats storage and increases gas costs/iteration time in `get_members`, degrading performance over time.

As a recommendation, add logic in `remove_role` and `set_roles` to check if roles==0 after updates and invoke `remove_member` to purge empty entries, ensuring efficient table management without impacting existing functionality.

[L-04] Missing role bounds check in `has_role`

The has_role function in `sources/acl.move` lacks role bounds validation, unlike other role functions. This inconsistency could cause runtime aborts if invalid role values (≥128) are passed. The issue may arise in future integrations where external contracts pass user-controlled role parameters or during cross-contract calls with unvalidated inputs.

Impact -> Runtime trxs abort instead of graceful error handling, leading to inconsistent API behavior.

Recommendation -> Add `assert!(role < 128, EInvalidRole);` at the beginning of `has_role` function for consistency with add_role and remove_role.

[L-05] Unavailable view function

The `cooldown_underlying_amount` and `cooldown_end_time` functions take a `UserCooldown` and return its `underlying_amount` and `cooldown_end` fields.

```
/// Get the underlying amount from a cooldown
public fun cooldown_underlying_amount(cooldown: &UserCooldown): u64 {
    cooldown.underlying_amount
}

/// Get the cooldown end time
```

```
public fun cooldown_end_time(cooldown: &UserCooldown): u64 {
    cooldown.cooldown_end
}
```

However, these two view functions are actually unusable because `UserCooldown` is a field within `SdeUSDManagement`, which cannot be directly accessed by external modules. Moreover, the module does not contain any function that returns a reference to `UserCooldown`. Therefore, these two functions cannot be used in practice.

It is recommended to remove these two functions entirely. Alternatively, they can be converted into helper functions for use by the `get_user_cooldown_info` function.

[L-06] The `deposit` function lacks asset support check

The `deposit` function allows anyone to send coins to `management`, mainly intended for the router to inject the tokens required for withdrawals after rebalancing.

```
public fun deposit<T>(
    management: &mut DeUSDMintingManagement,
    global_config: &GlobalConfig,
    coins: Coin<T>,
    ctx: &TxContext,
) {
    assert_package_version_and_initialized(management, global_config);
    let amount = coins.value();
    assert!(amount > 0, EInvalidAmount);

    let balance = get_or_create_balance_store_mut<T>(management);
    balance.join(coin::into_balance(coins));

    event::emit(Deposit {
        depositor: ctx.sender(),
        asset: type_name::get<T>(),
        amount,
    });
}
```

However, it is recommended to perform an `is_support_asset` check here to avoid mistakenly accepting unsupported assets, as these assets would still require `collateral_manager` and `AdminCap` to transfer out.

[L-07] Missing the function to transfer the `AdminCap` object

The `AdminCap` is the protocol's privilege credential. It is created in the `init` function and transferred to the sender.

```
public struct AdminCap has key {
    id: UID,
}
```

However, the `AdminCap` object only has the `key` ability, which means it cannot be freely transferred via `public_transfer` outside the module. Moreover, there is no function within the module that allows the `AdminCap` holder to transfer ownership. This indicates that the functionality for owner transfer is missing.

It is recommended to add a function to transfer the ownership of the `AdminCap` in the `admin_cap` module.

[L-08] Soft restriction bypass via token transfers

In the `sdeusd.move` function `add_to_blacklist()` function soft-restricted users are only blocked from direct staking but can receive sdeUSD tokens via transfers, bypassing the restriction entirely.

lets check the **vulnerable code**:

```
if (is_full_blacklisting) {
    coin::deny_list_v2_add(deny_list, &mut management.deny_cap, target, ctx);
} else {
    management.soft_restricted_stakers.add(target);  // Only internal tracking
    // Missing: deny_list addition for soft restrictions @audit-poc
}
```

This is **how it occurs**: 1. Alice gets soft-blacklisted (cannot call `deposit()`). 2. Bob stakes deUSD normally and gets sdeUSD shares. 3. Bob transfers sdeUSD tokens to Alice via `transfer::public_transfer()`. 4. Alice successfully increased her stake without triggering restriction checks.

This means soft restrictions only prevent direct protocol interaction, not token receipt.

Impact

Soft-restricted users can bypass staking restrictions through collusion or token gifts.

Recommendation

Add soft-restricted users to the coin separate deny list so that direct transfers are also disabled, this way, another user sending sdeUSD to a blacklisted user will not be allowed.

[L-09] Cooldown timer reset causes user withdrawal delays

In `deusd_lp_staking.move` function `unstake()`, each unstake operation overwrites the cooldown timestamp for all cooling tokens, not just the newly unstaked amount.

If you check this **vulnerable code**, it resets the timer for ALL cooling tokens

```
stake_data.cooldown_start_timestamp = clock_utils::timestamp_seconds(clock);
// ↑ Resets timer for ALL cooling tokens
```

This is **how it occurs**: 1. User unstakes 1000 tokens on Day 0 (withdrawable Day 30). 2. User unstakes 100 more tokens on Day 20 (thinking it's independent). 3. Cooldown resets: all 1100 tokens now withdrawable Day 50 (30 days from Day 20).

The **root cause** here is that a single timestamp is shared across all cooling tokens instead of per-batch tracking.

The concrete impact here is user confusion and unintended withdrawal delays from multiple unstake transactions resetting cooldown periods.

Recommendation

Prevent cooldown reset if tokens are already cooling.

```
// In unstake() function:
if (stake_data.cooling_down_amount == 0) {
    stake_data.cooldown_start_timestamp = clock_utils::timestamp_seconds(clock);
}
```

[L-10] Cooldown duration changes apply retroactively

In `deusd_lp_staking.move` function `withdraw()`, the cooldown duration is read from current parameters at withdrawal time, not locked when user unstakes. Admin can change cooldown periods mid-stream, affecting users who have already begun cooling down.

Let's check this **scenario**: 1. User unstakes 1000 tokens when cooldown = 30 days. 2. User expects withdrawal on Day 30. 3. Admin updates cooldown to 60 days on Day 20. 4. User tries withdrawing on Day 30 → fails because `assert!(current_time >= stake_data.cooldown_start_timestamp + params.cooldown, ECooldownNotOver);` now requires 60 days.

The **root cause** is that `params.cooldown` always reads current global setting, not the duration that was active when the user started cooldown.

One clear and concrete impact is **unpredictable withdrawal times**, users cannot reliably plan when their funds will be available, as the admin can retroactively extend lockup periods.

Recommendation

Recommendation is to lock the cooldown duration at unstaking time:

```
public struct StakeData has store, copy, drop {
    staked_amount: u64,
    cooling_down_amount: u64,
    cooldown_start_timestamp: u64,
    locked_cooldown_duration: u64,  // ADD: Store duration when unstaking
}

// In unstake():
stake_data.locked_cooldown_duration = params.cooldown;
```

```
// In withdraw():
assert!(current_time >= stake_data.cooldown_start_timestamp +
stake_data.locked_cooldown_duration, ECooldownNotOver);
```

Another solution will be to document clearly that cooldown periods can change anytime and apply retroactively, ensuring users understand this risk before unstaking.