Level Money Contracts

Security Review

Cantina Managed review by:

Milotruck, Security Researcher

Phaze, Security Researcher

tchkvsky, Junior Security Researcher

Delvir0, Junior Security Researcher

July 3, 2024

Contents

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

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

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

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

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

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

2 Security Review Summary

Level is the first delta-neutral synthetic dollar with first-loss protection.

From Jun 24th to Jun 27th the Cantina team conducted a review of contracts on commit hash e920c068. The team identified a total of **12** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 6
- Low Risk: 3
- Gas Optimizations: 1
- Informational: 2

3 Findings

3.1 Medium Risk

3.1.1 Inability to unstake after cooldown period removal

Severity: Medium Risk

Context: StakedlvlUSD.sol#L484-L529, StakedlvlUSD.sol#L442-L482, StakedlvlUSD.sol#L531-L543

Description: Users that have initiated the cooldown process will be unable to unstake when the cooldown period is reduced or removed.

When a user initiates the cooldown process for their shares by calling `cooldownAssets` or `cooldownShares`, the cooldown end time is stored for the user. However, if the cooldown duration is reduced or removed (through `setCooldownDuration`) after a user has initiated the cooldown process, the user will not be able to unstake immediately as their cooldown end time will not be adjusted accordingly. This is because the protocol stores the end time during cooldown initiation and does not use a relative time calculation.

This issue can become problematic in emergency situations where the protocol might need to allow immediate unstaking by setting the cooldown duration to zero. In such scenarios, users who had previously initiated a cooldown would be unable to access their funds, while users who hadn't would be able to unstake immediately.

The following proof of concept demonstrates the scenario:

```
function testCoolDownPeriodReduction() public {
    // Set cooldown duration to be 7 days
    vm.prank(owner);
    stakedlvlUSD.setCooldownDuration(7 days);

    // Alice and Bob both deposit 100 ether assets
    uint256 amount = 100 ether;
    _mintApproveDeposit(alice, amount);
    _mintApproveDeposit(bob, amount);

    uint256 shares = stakedlvlUSD.balanceOf(alice);

    // Alice initiates share cooldown process in anticipation of unstaking
    vm.startPrank(alice);
    stakedlvlUSD.cooldownShares(shares, alice);
    vm.stopPrank();

    // An issue related to the protocol's solvency
    // forces immediate action to allow all users to
    // unstake by removing the cooldown duration
    vm.prank(owner);
    stakedlvlUSD.setCooldownDuration(0);

    // Alice's attempt to unstake or redeem shares is blocked
    vm.startPrank(alice);
    vm.expectRevert(IStakedlvlUSDCooldown.InvalidCooldown.selector);
    stakedlvlUSD.unstake(alice);
    // Alice's shares are in the silo
    vm.expectRevert("ERC4626: redeem more than max");
    stakedlvlUSD.redeem(shares, alice, alice);
    vm.stopPrank();

    // Bob is able to redeem his shares
    shares = stakedlvlUSD.balanceOf(bob);
    vm.prank(bob);
    stakedlvlUSD.redeem(shares, bob, bob);
}
```

Recommendation: Instead of storing the cooldown end time, consider storing the cooldown start time instead.

```diff
  struct UserCooldown {
-     uint104 cooldownEnd;
+     uint104 cooldownStart;
      uint256 underlyingShares;
  }
```

This requires modifying the `cooldownAssets` and the `cooldownShares` functions to note the cooldown start time.

```
- cooldowns[owner].cooldownEnd =
-     uint104(block.timestamp) +
-     cooldownDuration;
+ cooldowns[owner].cooldownStart = uint104(block.timestamp);
```

When a user calls `unstake`, the cooldown end time will be computed given the user's cooldown start time and the current cooldown duration.

```
  function unstake(address receiver) external {
      UserCooldown storage userCooldown = cooldowns[msg.sender];
      uint256 shares = userCooldown.underlyingShares;
+     uint256 cooldownEnd = userCooldown.cooldownStart + cooldownDuration;
+     if (block.timestamp >= cooldownEnd) {
+         userCooldown.cooldownStart = 0;
-     if (block.timestamp >= userCooldown.cooldownEnd) {
-         userCooldown.cooldownEnd = 0;
          userCooldown.underlyingShares = 0;

          // ...

      } else {
          revert InvalidCooldown();
      }
  }
```

3.1.2 `FULL_RESTRICTED_STAKER_ROLE` blacklist can be bypassed using approvals

Severity: Medium Risk

Context: StakedlvlUSD.sol#L393-L398, StakedlvlUSD.sol#L455-L459

Description: In `StakedlvlUSD`, when users are blacklisted with `FULL_RESTRICTED_STAKER_ROLE`, they should not be able to withdraw their slvlUSD for lvlUSD. In EIP-4626, when calling `withdraw()` and `redeem()`, the caller can specify `owner` to withdraw shares on the owner's behalf:

```
function redeem(
    uint256 shares,
    address receiver,
    address owner
) public virtual override ensureCooldownOff returns (uint256) {
```

This requires `owner` to give approval to the caller to transfer his shares beforehand. However, `_withdraw()` only checks that the `caller` and `receiver` do not have `FULL_RESTRICTED_STAKER_ROLE`:

```
if (
    hasRole(FULL_RESTRICTED_STAKER_ROLE, caller) ||
    hasRole(FULL_RESTRICTED_STAKER_ROLE, receiver)
) {
    revert OperationNotAllowed();
}
```

Therefore, a user with `FULL_RESTRICTED_STAKER_ROLE` can bypass the blacklist and withdraw his shares by doing the following:

- From the blacklisted address, call `approve()` to give approval to another address.
- From the other address, call `redeem()`/`withdraw()` with:
  - `receiver` as the other address.
  - `owner` as the blacklisted address.

Recommendation: In `_withdraw()`, ensure that the `owner` does not have `FULL_RESTRICTED_STAKER_ROLE` as well:

```
  if (
      hasRole(FULL_RESTRICTED_STAKER_ROLE, caller) ||
+     hasRole(FULL_RESTRICTED_STAKER_ROLE, _owner) ||
      hasRole(FULL_RESTRICTED_STAKER_ROLE, receiver)
  ) {
      revert OperationNotAllowed();
  }
```

3.1.3 `_checkMinShares()` does not account for shares in the silo

Severity: Medium Risk

Context: StakedlvlUSD.sol#L320-L324, StakedlvlUSD.sol#L374-L375

Description: In `StakedlvlUSD`, `_checkMinShares()` ensures that the total supply of slvlUSD is either `0`, or not less than `MIN_SHARES`:

```
function _checkMinShares() internal view {
    uint256 _totalSupply = totalSupply();
    if (_totalSupply > 0 && _totalSupply < MIN_SHARES)
        revert MinSharesViolation();
}
```

When users initiate a withdrawal using `cooldownAssets()/cooldownShares()`, instead of being burned, their shares are transferred into `slvlUSDSilo` and temporarily held there.

However, since `_checkMinShares()` only checks that `totalSupply()` is not less than `MIN_SHARES`, shares in the silo are not considered burned. Most notably, the following call to `_checkMinShares()` in `_escrow()` does not check anything as `_escrow()` transfers slvlUSD, so the total supply of slvlUSD does not change:

```
super._transfer(_owner, receiver, shares);
_checkMinShares();
```

This allows users to initiate withdrawals using `cooldownAssets()/cooldownShares()` that subsequently cannot be executed using `unstake()`. For example:

- Assume that:
  - `MIN_SHARES = 1e18`.
  - Alice has `0.5e18` shares, while Bob has `1e18` shares.
  - `totalSupply = 1.5e18`.
- Bob calls `cooldownShares()` to withdraw `1e18` shares. `_checkMinShares()` in `_escrow()` passes as `totalSupply` remains at `1.5e18`.
- When Bob calls `unstake()`, `_checkMinShares()` reverts as burning his `1e18` shares reduces `totalSupply` to `0.5e18`, which is below `MIN_SHARES`.

Note that even if users pay special care not to go below `MIN_SHARES` when initiating withdrawals, other users could still cause their withdrawal to become un-executable:

- Assume that:
  - Bob has `1e18` shares.
  - `totalSupply = 1e18`.
- Bob calls `cooldownShares()` to withdraw `1e18` shares, which should reduce `totalSupply` to `0`.
- Alice calls `deposit()`, which mints `0.5e18` shares:
  - `totalSupply = 1e18 + 0.5e18 = 1.5e18`
- When Bob calls `unstake()`, `_checkMinShares()` reverts as burning his `1e18` shares ends up reducing `totalSupply` to `0.5e18`.

In such scenarios, the user's shares become stuck in the silo as `unstake()` can never be called to withdraw their shares, causing a loss of funds.

Recommendation: In `_checkMinShares()`, the shares held in the silo should be considered burnt:

```
  function _checkMinShares() internal view {
-     uint256 _totalSupply = totalSupply();
+     uint256 _totalSupply = totalSupply() - balanceOf(address(silo));
      if (_totalSupply > 0 && _totalSupply < MIN_SHARES)
          revert MinSharesViolation();
  }
```

3.1.4 Shares cannot be redistributed from users blacklisted during the cooldown period

Severity: Medium Risk

Context: StakedlvlUSD.sol#L500, StakedlvlUSD.sol#L474-L478, StakedlvlUSD.sol#L251-L254

Description: In `StakedlvlUSD`, whenever a user calls `cooldownAssets()`/`cooldownShares()`, his shares are transferred to the silo:

```
_escrow(_msgSender(), address(silo), owner, shares);
```

If the user is given `FULL_RESTRICTED_STAKER_ROLE` during the 7-day cooldown period, when he tries to call `unstake()`, the function will revert in the following block:

```
// withdraw slvlUSD to this contract
silo.withdraw(msg.sender, shares);

// burn slvlUSD from this contract, and send corresponding amount of assets to receiver
super.redeem(shares, receiver, msg.sender);
```

This is because the user's shares cannot be transferred from the silo back to him, so `silo.withdraw()` will revert. However, `redistributeLockedAmount()` only allows the admin to move shares that are held in the blacklisted address:

```
uint256 amountToDistribute = balanceOf(from);
_burn(from, amountToDistribute);
// to address of address(0) enables burning
if (to != address(0)) _mint(to, amountToDistribute);
```

Therefore, if a user is blacklisted during the 7-day cooldown period, his shares are permanently stuck in the silo. For example:

- Bob calls `cooldownShares()` to withdraw all his 100 shares. This transfers his 100 shares to the silo.
- Admin calls `addToDenylist()` and gives Bob `FULL_RESTRICTED_STAKER_ROLE`.
- Now, Bob's shares are permanently stuck in the silo as:
  - When Bob calls `unstake()`, it reverts as described above.
  - When the admin calls `redistributeLockedAmount()` to redistribute Bob's shares, `balanceOf()` returns `0` as his shares are in the silo.

This harms the protocol as a portion of slvlUSD, and by extension, lvlUSD, will no longer be in circulation.

Recommendation: In `redistributeLockedAmount()`, consider redistributing the user's shares from the silo as well:

```
- uint256 amountToDistribute = balanceOf(from);
- _burn(from, amountToDistribute);
+ uint256 amountInEOA = balanceOf(from);
+ _burn(from, amountInEOA);
+
+ uint256 amountInSilo = cooldowns[from].underlyingShares;
+ if (amountInSilo != 0) {
+     _burn(address(silo), amountInSilo);
+     delete cooldowns[from];
+ }
+
+ uint256 amountToDistribute = amountInEOA + amountInSilo;

  // to address of address(0) enables burning
  if (to != address(0)) _mint(to, amountToDistribute);
```

3.1.5 Cooldown period can be bypassed as users gain yield during the withdrawal process

Severity: Medium Risk

Context: StakedlvlUSD.sol#L519-L522, StakedlvlUSD.sol#L468-L479

Description: When `cooldownDuration` is set in `StakedlvlUSD`, withdrawals are a two-step process. Firstly, the user calls `cooldownAssets()/cooldownShares()` to initiate the withdrawal process, which stores the amount of shares (ie. slvlUSD) they would like to withdraw:

```
cooldowns[owner].cooldownEnd =
    uint104(block.timestamp) +
    cooldownDuration;
cooldowns[owner].underlyingShares += shares;
```

After the `cooldownDuration` has passed, the user then calls `unstake()` to execute the withdrawal. This redeems the stored amount of slvlUSD for lvlUSD at the current share price (ie. slvlUSD/lvlUSD conversion rate):

```
UserCooldown storage userCooldown = cooldowns[msg.sender];
uint256 shares = userCooldown.underlyingShares;
if (block.timestamp >= userCooldown.cooldownEnd) {
    // ...

    // burn slvlUSD from this contract, and send corresponding amount of assets to receiver
    super.redeem(shares, receiver, msg.sender);
} else {
```

Since the amount of lvlUSD withdrawn is based on the share price when `unstake()` is called, users are exposed to both gains and losses in `StakedlvlUSD` during the cooldown duration. For example, if lvlUSD rewards are sent to `StakedlvlUSD` after a user calls `cooldownShares()`, the user would still receive a portion of these rewards when he calls `unstake()`.

However, this allows users to bypass the cooldown duration. After depositing, users can instantly call `cooldownShares()/cooldownAssets()` to initiate the 7-day cooldown period, wait for the cooldown period to end, and then delay calling `unstake()` until after new rewards are distributed and vested. Furthermore, whenever the admin calls `freeze()` to socialize losses, users can front-run the admin to call `unstake()` and withdraw their assets, thereby avoiding losses.

This effectively allows users to bypass the cooldown mechanism while gaining positive exposure and giving them the option to evade negative exposure. Furthermore, if a large portion of users do this, the protocol will have no time to wind down their offchain positions to service withdrawals.

The following proof of concept demonstrates how users gain rewards during the cooldown period:

```
function testCoolDownSharesAndUnstake() public {
    // set cooldown duration to be 7 days
    vm.prank(owner);
    stakedlvlUSD.setCooldownDuration(7 days);

    uint256 amount = 100 ether;
    _mintApproveDeposit(alice, amount);

    uint256 shares = stakedlvlUSD.balanceOf(alice);

    assertEq(stakedlvlUSD.balanceOf(alice), shares);
    assertEq(stakedlvlUSD.balanceOf(alice), 100 ether);

    // initiate share cooldown process in anticipation of unstaking
    vm.startPrank(alice);
    stakedlvlUSD.cooldownShares(shares, alice);
    vm.stopPrank();

    // Alice is able to unstake any time now after cooldown period
    skip(7 days);

    // Alice waits until the next reward is transferred in
    lvlUSDToken.mint(rewarder, 100 ether);
    vm.startPrank(rewarder);
    lvlUSDToken.approve(address(stakedlvlUSD), 100 ether);
    stakedlvlUSD.transferInRewards(100 ether);
    vm.stopPrank();
```

```
    assertEq(lvlUSDToken.balanceOf(address(stakedlvlUSD)), 200 ether);

    // Alice waits until next reward's vesting period ends
    skip(8 hours);

    // Alice unstakes and receives all rewards
    vm.prank(alice);
    stakedlvlUSD.unstake(alice);

    assertApproxEqAbs(lvlUSDToken.balanceOf(alice), 200 ether, 10);
}
```

Recommendation: The amount of lvlUSD withdrawn should be determined using the minimum share price at the time `cooldownShares()/cooldownAssets()` is called and at the time `unstake()` is called. This can be done as follows:

1. Add a `expectedAssets` field to the `UserCooldown` struct:

```
struct UserCooldown {
    uint104 cooldownEnd;
    uint256 underlyingShares;
+     uint256 expectedAssets;
}
```

2. In `cooldownShares()/cooldownAssets()`, store the expected amount of assets to withdraw in `expectedAssets`:

```
cooldowns[owner].cooldownEnd =
    uint104(block.timestamp) +
    cooldownDuration;
cooldowns[owner].underlyingShares += shares;
+ cooldowns[owner].expectedAssets += assets;
```

3. In `unstake()`, the amount of assets withdrawn is the minimum of `expectedAssets` and `previewRedeem(shares)`:

```
// burn slvlUSD from this contract, and send corresponding amount of assets to receiver
- super.redeem(shares, receiver, msg.sender);
+ uint256 assets = previewRedeem(shares);
+ if (userCooldown.expectedAssets < assets) assets = userCooldown.expectedAssets;
+ _withdraw(msg.sender, receiver, msg.sender, assets, shares);
```

Note that during the cooldown period, the user's assets can still be used to socialize losses but do not earn any yield. As such, consider documenting that users should call `unstake()` as soon as possible to minimize any potential losses.

3.1.6 `getFreezableAmount` does not account for already frozen assets

Severity: Medium Risk

Context: StakedlvlUSD.sol#L166-L188

Description: The `getFreezableAmount` function incorrectly calculates the permitted freezable assets by not accounting for already frozen assets. This oversight leads to a progressive reduction in the freezable amount when the `freeze` function is called multiple times.

Currently, the function calculates the freezable amount as a percentage of the total assets in the staking contract. However, when assets are frozen, they are transferred from the staking contract to the `Freezer` contract. This transfer reduces the `totalAssets` value, which is based on the staking contract's current balance (`asset.balanceOf(address(this))`). Consequently, each subsequent calculation of the freezable amount is based on a diminished `totalAssets` value.

This issue effectively limits the ability to freeze the intended percentage of assets over multiple `freeze` function calls.

To illustrate the problem, consider these scenarios:

Scenario 1: *Single large freeze*:

1. Initial state: `totalAssets = 100`, `freezable = 50` (50%), `toFreeze = 50`.

2. After `freeze` call: 50 assets transferred to `Freezer`.
3. Result: `totalAssets = 50`, `freezable = 25` (50%), `alreadyFrozen = 50`.

   The frozen amount now exceeds the calculated freezable amount.

Scenario 2: *Multiple smaller freezes*:

1. Initial state: `totalAssets = 100`, `freezable = 50` (50%), `toFreeze = 33`.
2. After first `freeze` call: 33 assets transferred to `Freezer`.
3. New state: `totalAssets = 67`, `freezable = 33` (50%).
4. Result: No more assets can be frozen, despite only 33% being frozen so far.

The issue potentially prevents the protocol from freezing the intended amount of assets. This could affect the protocol's ability to respond to emergencies of implement governance decisions. The likelihood of this occurring depends on how often the `freeze` function is called.

The following proof of concept demonstrates how the freezable amount is incorrectly reduced when the `freeze` function is called multiple times:

```
function testIncorrectFreezableAmountCalculation() public {
    // Setup
    uint256 initialAmount = 100 ether;
    uint16 freezablePercentage = 5000; // 50%

    // Set freezable percentage
    vm.prank(owner);
    stakedlvlUSD.setFreezablePercentage(freezablePercentage);

    // Mint and deposit initial amount
    _mintApproveDeposit(alice, initialAmount);

    // Verify initial state
    assertEq(stakedlvlUSD.totalAssets(), initialAmount);
    assertEq(stakedlvlUSD.getFreezableAmount(), initialAmount / 2);

    // Freeze half of the freezable amount
    uint256 firstFreezeAmount = stakedlvlUSD.getFreezableAmount() / 2;
    // Grant freezer role
    vm.prank(owner);
    stakedlvlUSD.grantRole(FREEZER_ROLE, freezer);
    vm.prank(freezer);
    stakedlvlUSD.freeze(firstFreezeAmount);

    // Verify state after first freeze
    assertEq(stakedlvlUSD.totalAssets(), initialAmount - firstFreezeAmount);

    // Calculate expected freezable amount
    uint256 expectedFreezableAmount = (initialAmount * freezablePercentage) / 10_000;

    // But the actual freezable amount is less due to the incorrect calculation
    uint256 actualFreezableAmount = stakedlvlUSD.getFreezableAmount();

    // Show that the actual freezable amount is less than expected
    assertLt(actualFreezableAmount, expectedFreezableAmount, "Actual freezable amount should be less than
↪   expected");

    // Attempt to freeze the remaining expected amount (this should fail)
    vm.prank(freezer);
    vm.expectRevert();
    stakedlvlUSD.freeze(expectedFreezableAmount - firstFreezeAmount);
}
```

Recommendation: Modify the `getFreezableAmount` function to include both the staking contract's balance and the Freezer's balance in its calculation:

```
  function getFreezableAmount() public view returns (uint256) {
-     return (totalAssets() * freezablePercentage) / 10_000;
+     uint256 frozenAssets = IERC20(asset()).balanceOf(address(freezer));
+     return (totalAssets() + frozenAssets) * freezablePercentage / 10_000;
  }
```

3.2 Low Risk

3.2.1 `MIN_SHARES` check could DOS deposits/withdrawals

Severity: Low Risk

Context: StakedlvlUSD.sol#L319-L324

Description: In `StakedlvlUSD`, the `_checkMinShares()` function ensures that the total supply of slvlUSD is either `0`, or not lower than `1e18`:

```
/// @notice ensures a small non-zero amount of shares does not remain, exposing to donation attack
function _checkMinShares() internal view {
    uint256 _totalSupply = totalSupply();
    if (_totalSupply > 0 && _totalSupply < MIN_SHARES)
        revert MinSharesViolation();
}
```

This function is called whenever users deposit or withdraw lvlUSD, since the total supply of slvlUSD changes. However, when the `StakedlvlUSD` contract is first deployed, an attacker can donate lvlUSD to the contract to severely inflate how much `1e18` slvlUSD corresponds to, causing deposits to be DOSed. For example:

- The `StakedlvlUSD` contract holds no lvlUSD and no slvlUSD has been minted.
- Attacker transfers `1e18` lvlUSD directly to the contract.
- Now, the amount of lvlUSD needed to mint `1e18` slvlUSD is:

  ```
  assets = shares * (totalAssets + 1) / (totalSupply + 1) =  1e18 * (1e18 + 1) / (0 + 1) = 1e36
  ```

- Since lvlUSD is pegged to USD, `1e18` USD is needed for the first deposit, which is not possible.

Additionally, the `MIN_SHARES` check could cause withdrawals to be DOSed when multiple users hold a small amount of slvlUSD. For example:

- Assume that:
  - Alice and Bob hold `0.5e18` slvlUSD each.
  - They are the last two stakers remaining (ie. `totalSupply = 1e18`).
- If either of them attempt to withdraw any shares, `totalSupply` is decreased to lower than `1e18`.
- As such, both of them will never be able to withdraw any shares.

An attacker could also front-run calls to `redeem()`/`withdraw()` to force them to revert due to the `MIN_SHARES` check:

- Assume that:
  - Alice and Bob hold `1e18` slvlUSD each.
  - They are the last two stakers remaining (ie. `totalSupply = 2e18`).
- Bob calls `redeem()` to withdraw `1e18` shares.
- Alice front-runs Bob and calls `redeem()` to withdraw `1` share:
  - `totalSupply = 2e18 - 1`.
- Bob's call to `redeem()` is executed afterwards:
  - `totalSupply = 2e18 - 1 - 1e18 = 1e18 - 1`.
  - Since `totalSupply` is lower than `1e18`, the `MIN_SHARES` check reverts.

Recommendation: The team has stated that they will perform a first deposit on deployment, which eliminates the risk of an attacker donating lvlUSD to DOS deposits.

However, to prevent withdrawals from ever being DOSed due to the `MIN_SHARES` check, this first deposit should never be withdrawn. This ensures that `totalSupply` will never go below `1e18` and the `MIN_SHARES` check will never fail.

3.2.2 LvlUSD from rewards might be permanently stuck in the `StakedlvlUSD` contract

Severity: Low Risk

Context: StakedlvlUSD.sol#L277-L288, StakedlvlUSD.sol#L234-L235

Description: In the `StakedlvlUSD` contract, lvlUSD transferred in as rewards vest over a period of 8 hours:

```
function getUnvestedAmount() public view returns (uint256) {
    uint256 timeSinceLastDistribution = block.timestamp -
        lastDistributionTimestamp;

    if (timeSinceLastDistribution >= VESTING_PERIOD) {
        return 0;
    }

    return
        ((VESTING_PERIOD - timeSinceLastDistribution) * vestingAmount) /
        VESTING_PERIOD;
}
```

However, even if all users happen to withdraw from `StakedlvlUSD` during the vesting period of 8 hours, the remaining rewards will continue vesting. These rewards will be allocated to the 1 virtual share in OpenZeppelin's ERC4626 implementation, causing a scenario where the `StakedlvlUSD` contract holds lvlUSD, but they cannot be withdrawn by anyone.

Additionally, `rescueTokens()` cannot be used to rescue the remaining lvlUSD in the contract due to the `address(token) == asset()` check:

```
if (address(token) == asset()) revert InvalidToken();
IERC20(token).safeTransfer(to, amount);
```

Therefore, if all users withdraw while rewards are still vesting, the remaining lvlUSD rewards will be permanently stuck in the `StakedlvlUSD` contract.

Recommendation: Modify `rescueTokens()` to allow lvlUSD to be withdrawn by the admin when there are no stakers:

```
- if (address(token) == asset()) revert InvalidToken();
+ if (address(token) == asset() && totalSupply() != 0) revert InvalidToken();
  IERC20(token).safeTransfer(to, amount);
```

Note that this recommendation is not full-proof; an attacker can always front-run a call to `rescueTokens()` to perform a deposit, causing `totalSupply()` to no longer be `0`.

3.2.3 No upper-bound check when setting `freezablePercentage`

Severity: Low Risk

Context: StakedlvlUSD.sol#L545-L555

Description: The `freezablePercentage` variable is initially set to 0 during contract construction. The NATSPEC comment suggests that the value can be anywhere between 0 and 10,000 (representing 0-100%). The `setFreezablePercentage` function is used to update this percentage value, and currently, it only ensures that the value fits in a `uint16`. As it stands, a value of 10,000 or greater can be set, which would result in the entire asset in the vault being frozen when `freeze()` is called. If there's a specific maximum acceptable value for `freezablePercentage`, it should be enforced to ensure that the freezing of funds remains within an intended range.

Recommendation: Add an upper boundary check to the `setFreezablePercentage` function to ensure that `freezablePercentage` does not exceed the maximum acceptable value.

```
+ uint16 constant MAX_FREEZABLE_PERCENTAGE = 5_000; // 50%
+ error MaxFreezablePercentage();

  function setFreezablePercentage(
      uint16 percentage
  ) external onlyRole(DEFAULT_ADMIN_ROLE) {
+     if (percentage > MAX_FREEZABLE_PERCENTAGE) revert MaxFreezablePercentage();
      uint16 previousFreezable = freezablePercentage;
      freezablePercentage = percentage;

      emit FreezablePercentageUpdated(previousFreezable, percentage);
  }
```

3.3 Gas Optimization

3.3.1 Redundant incremental performed in `transferInRewards()` and `transferInFrozenFunds()`

Severity: Gas Optimization

Context: StakedlvlUSD.sol#L133-L137, StakedlvlUSD.sol#L151-L155

Description: For both functions, when `getUnvestedAmount()` or `getUnvestedUnfrozenAmount() !=0`, the function reverts. Meaning this function will only execute when these values are 0.

While these return values are 0, they are being used to increment the following:

- `uint256 newVestingAmount = amount + getUnvestedAmount();`
- `uint256 newUnfreezingAmount = amount + getUnvestedUnfrozenAmount();`

Further, the declarations of the `RewardsReceived` and `FrozenFundsReceived` events make the assumption that `amount` and `newVestingAmount` could be distinct values.

```
/// @notice Event emitted when the rewards are received
event RewardsReceived(uint256 indexed amount, uint256 newVestingAmount);
/// @notice Event emitted when frozen funds are received
event FrozenFundsReceived(uint256 indexed amount, uint256 newVestingAmount);
```

Due to the conditional revert, these will always be the same. The second arguments in the event can therefore be removed.

Recommendation: Consider removing the second arguments to the event definitions as these values will always be the same.

```
  /// @notice Event emitted when the rewards are received
- event RewardsReceived(uint256 indexed amount, uint256 newVestingAmount);
+ event RewardsReceived(uint256 indexed amount);
  /// @notice Event emitted when frozen funds are received
- event FrozenFundsReceived(uint256 indexed amount, uint256 newVestingAmount);
+ event FrozenFundsReceived(uint256 indexed amount);
```

Further, `uint256 newVestingAmount` and `uint256 newUnfreezingAmount` can be replaced for the input value `amount` as they will always be the same amount.

- `transferInRewards()`:

```
  function transferInRewards(uint256 amount) external nonReentrant onlyRole(REWARDER_ROLE)
↪ notZero(amount) {
      if (getUnvestedAmount() > 0) revert StillVesting();
-     uint256 newVestingAmount = amount + getUnvestedAmount();

-     vestingAmount = newVestingAmount;
+     vestingAmount = amount;
      lastDistributionTimestamp = block.timestamp;
      // transfer assets from rewarder to this contract
      IERC20(asset()).safeTransferFrom(msg.sender, address(this), amount);

-     emit RewardsReceived(amount, newVestingAmount);
+     emit RewardsReceived(amount);
  }
```

- `transferInFrozenFunds()`:

```
   function transferInFrozenFunds(uint256 amount) external nonReentrant onlyRole(FREEZER_ROLE)
↪  notZero(amount) {
       if (getUnvestedUnfrozenAmount() > 0) revert StillVesting();
-      uint256 newUnfreezingAmount = amount + getUnvestedUnfrozenAmount();

-      unfreezingAmount = newUnfreezingAmount;
+      unfreezingAmount = amount;
       lastUnfreezingTimestamp = block.timestamp;

       // transfer assets from freezer to this contract
       freezer.withdraw(amount);

-      emit FrozenFundsReceived(amount, newUnfreezingAmount);
+      emit FrozenFundsReceived(amount);
   }
```

3.4 Informational

3.4.1 SlvlUSD symbol is passed into `ERC20Permit` instead of its name

Severity: Informational

Context: StakedlvlUSD.sol#L105-L109

Description: In the constructor of the `StakedlvlUSD` contract, the symbol of the slvlUSD token (ie. "slvlUSD") is passed as the `name` parameter to `ERC20Permit`'s constructor:

```
constructor(
    IERC20 _asset,
    address _initialRewarder,
    address _owner
) ERC20("Staked lvlUSD", "slvlUSD") ERC4626(_asset) ERC20Permit("slvlUSD") {
```

However, according to the comments in `ERC20Permit`, the name of the ERC20 token should be passed to `ERC20Permit`:

```
/**
 * @dev Initializes the {EIP712} domain separator using the `name` parameter, and setting `version` to `"1"`.
 *
 * It's a good idea to use the same `name` that is defined as the ERC20 token name.
 */
constructor(string memory name) EIP712(name, "1") {}
```

The `name` parameter in `ERC20Permit` is hashed and used in the domain separator in signatures. As a result, `slvlUSD.name()` will be different from the name stored in the domain separator.

Recommendation: Pass "Staked lvlUSD", which is the name of the token, into `ERC20Permit`:

```
  constructor(
      IERC20 _asset,
      address _initialRewarder,
      address _owner
- ) ERC20("Staked lvlUSD", "slvlUSD") ERC4626(_asset) ERC20Permit("slvlUSD") {
+ ) ERC20("Staked lvlUSD", "slvlUSD") ERC4626(_asset) ERC20Permit("Staked lvlUSD") {
```

3.4.2 Minor improvements

Severity: Informational

Context:

1. Freezer.sol#L10, Slasher.sol#L11, slvlUSDSilo.sol
2. Slasher.sol
3. StakedlvlUSD.sol#L61-L62
4. StakedlvlUSD.sol#L123-L124
5. StakedlvlUSD.sol#L474-L478

6. slvlUSDSilo.sol#L9
7. IStakedlvlUSD.sol#L35-L36

Description/Recommendation:

1. Freezer.sol#L10, Slasher.sol#L11, slvlUSDSilo.sol. The following line and the `SafeERC20` import can be removed as `SafeERC20` is not used:

```
- using SafeERC20 for IERC20;
```

2. Slasher.sol. All functions can be changed from `public` to `external` as they are not used internally.
3. StakedlvlUSD.sol#L61-L62. `silo` and `freezer` can be declared as `immutable` as they are never changed post-deployment:

```
- slvlUSDSilo public silo;
- Freezer public freezer;
+ slvlUSDSilo public immutable silo;
+ Freezer public immutable freezer;
```

4. StakedlvlUSD.sol#L123-L124. Setting `cooldownDuration` and `freezablePercentage` to `0` in the constructor is redundant as they are initialized to `0` by default. Consider removing both lines:

```
- cooldownDuration = 0;
- freezablePercentage = 0;
```

5. StakedlvlUSD.sol#L474-L478. In `unstake()`, shares are withdrawn from the silo to the user instead of the `StakedlvlUSD` contract. The following comments are incorrect:

```
- // withdraw slvlUSD to this contract
+ // withdraw slvlUSD to the user
silo.withdraw(msg.sender, shares);

- // burn slvlUSD from this contract, and send corresponding amount of assets to receiver
+ // burn slvlUSD from the user, and send corresponding amount of assets to receiver
super.redeem(shares, receiver, msg.sender);
```

6. slvlUSDSilo.sol#L9. The contract's title should be changed from `USDeSilo` to `slvlUSDSilo`:

```
- * @title USDeSilo
+ * @title slvlUSDSilo
```

7. IStakedlvlUSD.sol#L35-L36. The natspec incorrectly refers to `USDe` instead of `lvlUSD`:

```
- /// @notice Error emitted when owner attempts to rescue USDe tokens.
+ /// @notice Error emitted when owner attempts to rescue lvlUSD tokens.
  error InvalidToken();
```