Level V2 Security Review

Pashov Audit Group

Conducted by: Said, btk, zark, Araj, TheWeb3Mechanic, Silvermist, 0xAbhay

April 9th 2025 - April 18th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check 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. Introduction

A time-boxed security review of the **Level-Money/contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Level V2

Level is the protocol behind Level USD (lvlUSD), a stablecoin backed by restaked dollar tokens. Users can mint and stake lvlUSD to earn a yield from AVSs on restaking protocols, as well as blue-chip on-chain lending protocols. lvlUSD is a stablecoin fully backed by USDC and USDT, which can be minted permissionlessly and generates yield through lending protocols. Users can stake lvlUSD to receive slvlUSD, which appreciates in value as yield is distributed, and both tokens are freely usable across integrated DeFi platforms.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. 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.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 08ffd7e9d81f7017498524e646bf3abba47426a4

fixes review commit hash - 5065d156f72b878db301907509eadb49760275d2

Scope

The following smart contracts were in scope of the audit:

- `DeployLevel`
- `DeploymentUtils`
- `LevelMintingV2`
- `LevelMintingV2Storage`
- `AuthUpgradeable`
- `StrictRolesAuthority`
- `PauserGuard`
- `PauserGuarded`
- `MathLib`
- `OracleLib`
- `StrategyLib`
- `VaultLib`
- `interfaces/`
- `LevelReserveLens`
- `AaveTokenOracle`
- `ERC4626DelayedOracle`
- `ERC4626Oracle`
- `ERC4626OracleFactory`
- `BoringVault`
- `RewardsManager`
- `RewardsManagerStorage`
- `Silo`
- `VaultManager`
- `VaultManagerStorage`

7. Executive Summary

Over the course of the security review, Said, btk, zark, Araj, TheWeb3Mechanic, Silvermist, 0xAbhay engaged with Level to review Level V2. In this period of time a total of **23** issues were uncovered.

Protocol Summary

Protocol Name	Level V2
Repository	https://github.com/Level-Money/contracts
Date	April 9th 2025 - April 18th 2025
Protocol Type	Stablecoin

Findings Count

Severity	**Amount**
High	1
Medium	5
Low	17
Total Findings	**23**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Zero heartBeat can cause reward claim failures	High	Resolved
[M-01]	reward does not account for an under-peg scenario	Medium	Resolved
[M-02]	Unnecessary withdraw can occur during initiateRedeem and reward	Medium	Resolved
[M-03]	Incorrect maxRedeemPerBlock check	Medium	Resolved
[M-04]	Max redeemPerBlock limit not implemented correctly	Medium	Resolved
[M-05]	Missing oracle updates in RewardsManager	Medium	Resolved
[L-01]	initialize() lacks asset and oracle length check	Low	Resolved
[L-02]	Wrong Boring vault setup	Low	Resolved
[L-03]	Morpho low-liquidity vaults at risk of price manipulation	Low	Acknowledged
[L-04]	Testing vaults in morpho are not yet removed from deployment script	Low	Resolved
[L-05]	aUSDC, aUSDT and steakhouse USDC not mintable	Low	Resolved
[L-06]	LevelReserveLens returns incorrect reserves	Low	Resolved
[L-07]	BoringVault can receive ETH but cannot withdraw it	Low	Resolved
[L-08]	Missing role assignment for strategy	Low	Resolved

	removal		
[L-09]	Missing role authorization for beforeTransferHook	Low	Resolved
[L-10]	Unnecessary and dangerous token approvals	Low	Resolved
[L-11]	Inconsistent value tracking in maxRedeemPerBlock across stablecoins	Low	Resolved
[L-12]	Collateral may be stuck in silo if asset is removed	Low	Resolved
[L-13]	Aave incentives will be locked	Low	Acknowledged
[L-14]	RewardsManager cannot handle yield split across multiple assets	Low	Acknowledged
[L-15]	LVLUSD burning causes permanent collateral accounting mismatch	Low	Acknowledged
[L-16]	Vault deposit frontrun	Low	Acknowledged
[L-17]	Default strategies are not updated on strategy removal	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Zero `heartBeat` can cause reward claim failures

Severity

Impact: Medium

Likelihood: High

Description

The `OracleLib` and `RewardsManager` contracts work together to enable reward claiming based on asset prices. A key part of this process involves the `getAssets()` function in `StrategyLib`, which calculates the total assets held by a vault. This function depends on price data fetched via `OracleLib.getPriceAndDecimals()`.

However, the current implementation passes a `heartBeat` value of 0 when calling `getPriceAndDecimals()`:

```
(
            int256assetsForOneShare,
            uint256decimals
        ) = OracleLib.getPriceAndDecimals(address(config.oracle
```

The `heartBeat` parameter is used to determine whether price data is still fresh. Passing 0 effectively means "no tolerance" for stale data. As a result, if the current `block.timestamp` is even slightly greater than the `updatedAt` timestamp of the price, the function will revert — causing the reward claim to fail.

Proof of Concept

- A user calls the `reward()` function on the `RewardsManager` contract.
- `reward()` triggers `getAccruedYield()`, which internally calls `getAssets()` from `StrategyLib`.
- `getAssets()` invokes `OracleLib.getPriceAndDecimals()` with a `heartbeat` of 0.
- If `block.timestamp > updatedAt + 0`, the call reverts, preventing the user from claiming rewards.
- This clearly shows how the current setup can block users from receiving rewards, especially when price feeds are not updated frequently.

Recommendations

Update the `getPriceAndDecimals()` call in `getAssets()` to use a non-zero `heartbeat`. A value of at least 3600 seconds (1 hour) is recommended.

8.2. Medium Findings

[M-01] `reward` does not account for an under-peg scenario

Severity

Impact: High

Likelihood: Low

Description

When `reward` is called, it calculates the accrued yield as the total asset balances in the vault and across strategies, minus the minted shares in the vault.

```
function getAccruedYield(address[] calldata assets) public view returns
      (uint256 accrued) {
        uint256 total;

        for (uint256 i = 0; i < assets.length; i++) {
            address asset = assets[i];

            StrategyConfig[] memory strategies = allStrategies[asset];

            uint256 totalForAsset = vault._getTotalAssets(strategies, asset);
 >>>        total += totalForAsset.convertDecimalsDown(ERC20(asset).decimals
   (), vault.decimals());
        }
        uint256 vaultShares = vault.balanceOf(address(vault));
        accrued = total - vaultShares;

        return accrued;
    }
```

However, this doesn't account for scenarios where the underlying asset is currently under-peg (price < $1). When the asset is under-peg, its actual value is lower than the calculated asset amount. This can result in an incorrect calculation of the accrued yield allocated to the treasury, potentially leading to lvlUSD being insufficiently collateralized in the event of a sharp price drop.

Recommendations

Consider each asset price when calculating the total value.

[M-02] Unnecessary withdraw can occur during `initiateRedeem` and `reward`

Severity

Impact: Medium

Likelihood: Medium

Description

When users trigger `initiateRedeem`, the operation always attempts to withdraw collateral from the default strategies without first checking whether the current collateral balance in the boring vault is sufficient to cover the redemption request.

```
function initiateRedeem
      (address asset, uint256 lvlUsdAmount, uint256 expectedAmount)
        external
        requiresAuth
        notPaused
        returns (uint256, uint256)
    {
        if (!redeemableAssets[asset]) revert UnsupportedAsset();
        if
          (!isBaseCollateral[asset]) revert RedemptionAssetMustBeBaseCollateral();
        if (lvlUsdAmount == 0) revert InvalidAmount();

        uint256 collateralAmount = computeRedeem(asset, lvlUsdAmount);
        if
          (collateralAmount < expectedAmount) revert MinimumCollateralAmountNotMet();

        pendingRedemption[msg.sender][asset] += collateralAmount;
        userCooldown[msg.sender][asset] = block.timestamp;

        // note preventing amounts that would fail by definition at complete
        // redeem due to max per block
        if
          (pendingRedemption[msg.sender][asset] > maxRedeemPerBlock) revert ExceedsMax

        lvlusd.burnFrom(msg.sender, lvlUsdAmount);

        // Don't block redemptions if withdraw default fails
>>>     try vaultManager.withdrawDefault(asset, collateralAmount) {
            emit WithdrawDefaultSucceeded(msg.sender, asset, collateralAmount);
        } catch {
            emit WithdrawDefaultFailed(msg.sender, asset, collateralAmount);
        }

        vaultManager.vault().exit(
            address(silo), ERC20(asset), collateralAmount, address
              (vaultManager.vault()), lvlUsdAmount
        );

        emit RedeemInitiated(msg.sender, asset, collateralAmount, lvlUsdAmount);

        return (lvlUsdAmount, collateralAmount);
    }
```

This could lead to unnecessary withdrawals from strategies, even when there is enough collateral in the vault to cover the redemption request. This results in suboptimal yield and asset management.

The same case also applies when `RewardsManager.reward` is called.

Recommendations

Only trigger `vaultManager.withdrawDefault` when the collateral balance in the vault is insufficient to cover the redemption and `reward` request.

[M-03] Incorrect `maxRedeemPerBlock` check

Severity

Impact: Medium

Likelihood: Medium

Description

`maxRedeemPerBlock` is intended to limit the maximum redemption per block. However, it is currently being checked against `pendingRedemption`.

```
function initiateRedeem
      (address asset, uint256 lvlUsdAmount, uint256 expectedAmount)
        external
        requiresAuth
        notPaused
        returns (uint256, uint256)
    {
        if (!redeemableAssets[asset]) revert UnsupportedAsset();
        if
          (!isBaseCollateral[asset]) revert RedemptionAssetMustBeBaseCollateral();
        if (lvlUsdAmount == 0) revert InvalidAmount();

        uint256 collateralAmount = computeRedeem(asset, lvlUsdAmount);
        if
          (collateralAmount < expectedAmount) revert MinimumCollateralAmountNotMet();

        pendingRedemption[msg.sender][asset] += collateralAmount;
        userCooldown[msg.sender][asset] = block.timestamp;

        // note preventing amounts that would fail by definition at complete
        // redeem due to max per block
>>>     if
  (pendingRedemption[msg.sender][asset] > maxRedeemPerBlock) revert ExceedsMaxBlockLim

        lvlusd.burnFrom(msg.sender, lvlUsdAmount);

        // Don't block redemptions if withdraw default fails
        try vaultManager.withdrawDefault(asset, collateralAmount) {
            emit WithdrawDefaultSucceeded(msg.sender, asset, collateralAmount);
        } catch {
            emit WithdrawDefaultFailed(msg.sender, asset, collateralAmount);
        }

        vaultManager.vault().exit(
            address(silo), ERC20(asset), collateralAmount, address
              (vaultManager.vault()), lvlUsdAmount
        );

        emit RedeemInitiated(msg.sender, asset, collateralAmount, lvlUsdAmount);

        return (lvlUsdAmount, collateralAmount);
    }
```

This will cause the check to not work as intended. Users can easily bypass it by transferring the lvlUSD to another account and redeeming it.

Recommendations

Either move the `maxRedeemPerBlock` check to `completeRedeem`, where `redeemedPerBlock` is increased or move the `redeemedPerBlock` increment to `initiateRedeem` and perform the check against `maxRedeemPerBlock` there.

```
function initiateRedeem
      (address asset, uint256 lvlUsdAmount, uint256 expectedAmount)
        external
        requiresAuth
        notPaused
        returns (uint256, uint256)
    {
        if (!redeemableAssets[asset]) revert UnsupportedAsset();
        if
          (!isBaseCollateral[asset]) revert RedemptionAssetMustBeBaseCollateral();
        if (lvlUsdAmount == 0) revert InvalidAmount();

        uint256 collateralAmount = computeRedeem(asset, lvlUsdAmount);
        if
          (collateralAmount < expectedAmount) revert MinimumCollateralAmountNotMet();

        pendingRedemption[msg.sender][asset] += collateralAmount;
        userCooldown[msg.sender][asset] = block.timestamp;

+      redeemedPerBlock[block.number] += collateralAmount;

        // note preventing amounts that would fail by definition at complete
        // redeem due to max per block
-        if
- (pendingRedemption[msg.sender][asset] > maxRedeemPerBlock) revert ExceedsMaxBlockLim
+        if
+ (redeemedPerBlock[block.number] > maxRedeemPerBlock) revert ExceedsMaxBlockLimit();

        // ...
    }
```

[M-04] Max `redeemPerBlock` limit not implemented correctly

Severity

Impact: Medium

Likelihood: Medium

Description

The `LevelMintingV2` contract is designed to control the total amount of assets that can be redeemed in a single block using the `maxRedeemPerBlock` parameter. This mechanism aims to maintain system stability and prevent

liquidity shortages. However, the current implementation fails to enforce this limit correctly.

The issue lies in how the `initiateRedeem()` and `completeRedeem()` functions handle redemptions. While `initiateRedeem()` checks if a user's requested redemption exceeds `maxRedeemPerBlock`, this check is applied on a per-user basis. As a result, each user can redeem up to the maximum allowed amount during their cooldown period—regardless of what other users are redeeming. This effectively means that multiple users can redeem the full limit concurrently in the same block.

Consider the following example:

1. `maxRedeemPerBlock` is set to 250,000.
2. User A calls `initiateRedeem()` with an amount of 250,000.
3. User B does the same, also requesting 250,000.
4. Both transactions are mined in the same block.
5. Total redemptions for the block = 500,000, which doubles the intended limit.

Recommendations

Use the same mechanism that is used during minting.

[M-05] Missing oracle updates in `RewardsManager`

Severity

Impact: Medium

Likelihood: Medium

Description

The `RewardsManager.getAccruedYield()` function calculates rewards based on oracle prices but fails to call `_tryUpdateOracle()` before fetching prices. This behavior is different from `LevelMintingV2`, which explicitly updates oracles before using their prices.

```
function getAccruedYield(address[] calldata assets) public view returns
      (uint256 accrued) {
        uint256 total;

        for (uint256 i = 0; i < assets.length; i++) {
            address asset = assets[i];
            StrategyConfig[] memory strategies = allStrategies[asset];
@>            uint256 totalForAsset = vault._getTotalAssets(strategies, asset);

            // ...
        }

        // ...
    }
```

As shown above, `getAccruedYield` calls `_getTotalAssets` for assets like `USDC` and `USDT` to compute the total assets controlled by `BoringVault`. This computation virtually converts all vault shares from lending strategies into their underlying assets.

```
function _getTotalAssets
      (BoringVault vault, StrategyConfig[] memory strategies, address asset)
        internal
        view
        returns (uint256 total)
    {
        // Initialize to undeployed
        uint256 totalForAsset = ERC20(asset).balanceOf(address(vault));

        for (uint256 j = 0; j < strategies.length; j++) {
            StrategyConfig memory config = strategies[j];
@>            totalForAsset += StrategyLib.getAssets(config, address(vault));
        }

        return totalForAsset;
    }
```

To perform this conversion, `StrategyLib.getAssets` is used. It checks the number of shares held in each strategy and converts them to the underlying asset using oracle prices. However, as shown below, the call to update the oracle (required for share tokens like steakUSDC or any other using `ERC4626DelayedOracle`) is missing.

```
function getAssets(
      StrategyConfigmemoryconfig,
      addressvault
    ) internal view returns (uint256 assets_
        // ...

        uint256 shares = receiptToken.balanceOf(vault);

        uint256 sharesToAssetDecimals =
            shares.mulDivDown(10 ** ERC20(address(
              10**ERC20
            )

@>          (
  int256assetsForOneShare,
  uint256decimals
) = OracleLib.getPriceAndDecimals(address(config.oracle
        assets_ = uint256(assetsForOneShare).mulDivDown
          (sharesToAssetDecimals, 10 ** decimals);
        return assets_;
    }
```

The impact of this vulnerability is that stale prices may be used during each rewards calculation, and it's unclear when `ERC4626DelayedOracle::update` will next be called to assign updated prices.

Recommendations

To resolve this issue, ensure that the oracle is updated before price retrieval, similar to the approach used in `LevelMintingV2.sol`.

8.3. Low Findings

[L-01] `initialize()` lacks asset and oracle length check

In initialize(), oracle is added for the asset through for-loop. However, it doesn't check/require asset.length to be equals oracles.length:

```
function initialize(
        address[] memory _assets,
        address[] memory _oracles,
....
    ) external initializer {
....
        for (uint256 i = 0; i < _assets.length; i++) {
            addOracle(_assets[i], _oracles[i], false);
        }
....

    }
```

Ensure both assets and oracles are of the same length, otherwise, it will be an out-of-bond error.

[L-02] Wrong Boring vault setup

`VaultManager` is a contract intended to be created behind an upgradeable proxy. However, in `VaultManagerStorage`, the boring vault is set in the constructor instead of in `VaultManager`'s initializer function. Consider moving this logic from the `VaultManagerStorage` constructor to the `initialize` function in `VaultManager`.

[L-03] Morpho low-liquidity vaults at risk of price manipulation

Morpho vaults or any ERC-4626 vaults are inherently prone to price manipulation, especially in the case of new or low-liquidity vaults. For instance, In an empty vault with no protection mechanisms:

- The attacker deposits a minimal amount (e.g., 1 wei of the token) and receives 1 share.
- The attacker then transfers a large amount directly to the vault (e.g., 100M tokens).
- This creates an exchange rate where 1 shares ≈ 100M tokens.
- A user depositing 1 token would receive 0 shares due to the inflated exchange rate.

Failing to check the price per share when calculating total value or during deposit/withdrawal operations could cause issues. Generally, slippage checks on the expected returned assets or shares are used to ensure no loss occurs when the ERC-4626 price is manipulated.

Level team comments:

Acknowledged. When depositing into vaults, we plan on adopting the following policy:

- at least $100k in initial underlying deposits
- we can be no more than 25% of the vault's size

[L-04] Testing vaults in morpho are not yet removed from deployment script

Inside `DeployLevel.s.sol`, there is a `_setupMorphoVaultsForTests` function which sets up and registers a list of strategies for testing purposes, but it is currently still being called inside the script. Consider removing the `_setupMorphoVaultsForTests` call inside the `run` function.

[L-05] `aUSDC`, `aUSDT` and steakhouse USDC not mintable

Based on the documentation, it is stated that users will be able to mint with aUSDC, aUSDT, and Steakhouse USDC. However, currently in `DeployLevel.s.sol`, these assets are not configured as mintable assets.

[L-06] `LevelReserveLens` returns incorrect reserves

The `_getReserves()` function in the `LevelReserveLens` contract currently only includes reserves from the v1 vaults, completely ignoring collateral held in the v2 vaults. As a result, it underestimates the total reserves backing lvlUSD, returning v1Reserves + 0—with the v2 collateral effectively hardcoded to zero:

```
function _getReserves(
      IERC20Metadatacollateral,
      addresswaCollateralAddress,
      addresssymbioticVault
    )
        internal
        view
        override
        returns (uint256)
    {
        uint256 v1Reserves = super._getReserves
          (collateral, waCollateralAddress, symbioticVault);

        uint256 boringVaultValue = 0;

        return v1Reserves + boringVaultValue;
    }
```

This misrepresentation can lead to inaccurate reserve data for any consumers relying on this function, potentially affecting financial decisions.

Add a public function to the `VaultManager` contract that returns the total assets held for a given collateral token. Then, update `_getReserves()` in LevelReserveLens to use this new function instead of the hardcoded zero for v2.

[L-07] `BoringVault` can receive ETH but cannot withdraw it

The `BoringVault` contract has `receive` function that is `payable`:

```
receive() external payable {}
```

If someone sends a transaction with `msg.value != 0` then the ETH will be stuck in the contract forever without a way for anyone to withdraw it.

Remove the `receive` function since the ETH balance is not used in the contract anyway.

[L-08] Missing role assignment for strategy removal

The `removeAssetStrategy` function in `VaultManager` requires auth but no role has been granted permission to call it. This could prevent removing problematic strategies in emergencies. Consider adding the role capability :

```
_setRoleCapabilityIfNotExists(
    STRATEGIST_ROLE,
    address(config.levelContracts.vaultManager),
    bytes4(abi.encodeWithSignature("removeAssetStrategy(address,address)"))
);
```

[L-09] Missing role authorization for `beforeTransferHook`

The `setBeforeTransferHook` function in BoringVault requires auth but no role has been granted permission to call it. This means the hook functionality is effectively disabled as no one can set it. Consider adding the necessary role capability :

```
_setRoleCapabilityIfNotExists(
            VAULT_MANAGER_ROLE,
            address(config.levelContracts.boringVault),
            bytes4(abi.encodeWithSignature("setBeforeTransferHook(address)"))
        );
```

[L-10] Unnecessary and dangerous token approvals

The `VaultLib` library contains unnecessary token approvals in both `_withdrawFromAave` and `_withdrawFromMorpho` functions:

```
function _withdrawFromAave
      (BoringVault vault, StrategyConfig memory _config, uint256 amount)
        internal
        returns (uint256 withdrawn)
    {
        address aaveV3 = _getAaveV3Pool();

@>        vault.increaseAllowance(address
  (_config.receiptToken), aaveV3, amount);

        // ...
     }

    function _withdrawFromMorpho
      (BoringVault vault, StrategyConfig memory _config, uint256 amount)
        internal
        returns (uint256 withdrawn)
    {
        IERC4626 morphoVault = IERC4626(_config.withdrawContract);

        uint256 sharesToRedeem = morphoVault.previewWithdraw(amount);

        if (sharesToRedeem == 0) {
            revert("VaultManager: amount must be greater than 0");
        }

@>        vault.increaseAllowance(address
  (_config.receiptToken), _config.withdrawContract, sharesToRedeem);

        // ...
    }
```

These approvals are unnecessary because:

- Aave V3's `withdraw` function doesn't require the approval of `aTokens`.
- Morpho's `withdraw` function burns shares directly from the caller's balance without requiring approval.

Consider removing these unnecessary token approvals from both withdrawal functions.

[L-11] Inconsistent value tracking in `maxRedeemPerBlock` across stablecoins

The `maxRedeemPerBlock` limit tracks collateral amounts (USDC/USDT) rather than the actual `lvlUSD` value being redeemed. Since stablecoins can deviate from their peg (e.g., USDC at $0.99 and USDT at $1.01), this creates inconsistencies in the actual `lvlUSD` value being limited. A user redeeming through USDT when it's above peg could redeem 1-2% more `lvlUSD` value

compared to someone using USDC when it's below peg, while both would be counted the same against the `maxRedeemPerBlock` limit.

```
uint256 collateralAmount = computeRedeem(asset, lvlUsdAmount);
        if
          (collateralAmount < expectedAmount) revert MinimumCollateralAmountNotMet();

@>        pendingRedemption[msg.sender][asset] += collateralAmount;
```

Consider tracking and limiting the `lvlUSD` value being redeemed instead of the raw collateral amount to ensure consistent value limits across different collateral types.

[L-12] Collateral may be stuck in silo if asset is removed

When user completeRedeem(), it requires the asset to be `redeemableAssets`. However, admin can remove any asset from redeemableAssets mapping through `removeRedeemableAsset()`. This will revert completeRedeem() and collateral will `stuck` in a silo contract.

```
function completeRedeem(
    addressasset,
    addressbeneficiary
  ) external notPaused returns (uint256 collateralAmount
        if (!redeemableAssets[asset]) revert UnsupportedAsset();
....

    }
```

1. Suppose a user-initiated redemption of 100e18 lvlUSD for USDC/T, which is baseCollateral as well as redeemableAssets. This will transfer the collateral from boringVault to the silo contract.
2. Admin removed the asset from redeemableAssets before user could complete his redeem.
3. When user call completeRedeem(), it'll revert because that asset was no longer a valid redeemableAssets.
4. As a result, the collateral of user will stick forever in the silo contract.

Recommendations:

Remove the requirement of redeemableAssets in completeRedeem() because collateral has been already removed from boringVault.

[L-13] Aave incentives will be locked

Aave provides **Incentives** (e.g., staking rewards or liquidity mining rewards, seeing here: https://aave.com/docs/primitives/incentives) to users who supply assets to the protocol. These incentives are typically distributed in the form of additional tokens (e.g., AAVE or other governance tokens) and can be claimed by users who interact with Aave's incentive mechanisms.

However, in the current implementation of the `LevelMintingV2, RewardManager, VaultManager` contract, there is no functionality to claim these incentives. This is a **missing feature** that could prevent levelV2 from accessing the full benefits of supplying assets to Aave.

The `vaultManager` contract supplies collateral tokens to Aave.

```
function _depositToAave
      (BoringVault vault, StrategyConfig memory _config, uint256 amount)
        internal
        returns (uint256 deposited)
    {
        address aaveV3 = _getAaveV3Pool();
        vault.increaseAllowance(address
          (_config.baseCollateral), aaveV3, amount);

        uint256 balanceBefore = ERC20(_config.baseCollateral).balanceOf(address
          (vault));
        vault.manage(
            address(aaveV3),
            abi.encodeWithSignature(
                "supply(address,uint256,address,uint16)", address
                  (_config.baseCollateral), amount, address(vault), 0
            ),
            0
        );
        uint256 balanceAfter = ERC20(_config.baseCollateral).balanceOf(address
          (vault));

        uint256 deposited_ = balanceBefore - balanceAfter;
        emit DepositToAave(address(vault), address
          (_config.baseCollateral), amount, deposited_);

        return deposited_;
    }
```

However, it does not provide a method for rewardManger/operator to claim the incentives that Aave distributes to suppliers. In Eth mainnet, the aave rewards contract
is: https://etherscan.io/address/0x8164Cc65827dcFe994AB23944CBC90e0aa80bFcb

Currently, this contract is still available for reward claiming.

As a result, levelV2 cannot claim the incentives provided by Aave, resulting in lost rewards.

Recommendations:

Add a function that allows operator/rewardManager to claim incentives from Aave. This involves interacting with Aave's **Incentives Controller** or **Rewards Distributor** contracts.

Level team comments

Acknowledged. Since the owner of the BoringVault can call manage, the rewards won't be lost forever. Aave seems to rarely issue rewards for USDC/T, and Morpho's reward claiming mechanism can be configured by the contract's deployer, so we're okay with deferring adding a function to claim rewards to the next RewardsManager update.

[L-14] `RewardsManager` cannot handle yield split across multiple assets

The `RewardsManager::reward` function cannot handle cases where the total accrued yield is split between multiple assets (`USDC` and `USDT`) and exceeds the balance of any single asset's strategies. This creates a situation where rewards cannot be claimed even though yield exists in the protocol.

```
function reward(address[] calldata assets) external requiresAuth {
        uint256 accrued = getAccruedYield(assets);
        address redemptionAsset = assets[0];

        if (accrued == 0) {
            revert NotEnoughYield();
        }

        uint256 accruedAssets = accrued.convertDecimalsDown(vault.decimals
          (), ERC20(redemptionAsset).decimals());

        vault._withdrawBatch(allStrategies[redemptionAsset], accruedAssets);

@>        vault.exit(treasury, ERC20(redemptionAsset), accruedAssets, address
  (vault), 0);

        emit Rewarded(redemptionAsset, treasury, accruedAssets);
    }
```

For example:

1. BoringVault has 10 shares.
2. USDC strategies have 11 USDC.
3. USDT strategies have 11 USDT.
4. Total accrued yield is (11+11) - 10 = 12.
5. When trying to claim rewards, the function will revert because:
   - It tries to withdraw 12 from USDC strategies (which only have 11).
   - Or try to withdraw 12 from USDT strategies (which only have 11).
   - Neither strategy has enough to cover the full yield amount.

This means that even though there is yield available in the protocol, it cannot be claimed because the reward function doesn't support splitting the withdrawal across multiple assets. As a result, every call on the `reward()` will always revert.

Recommendations:

`reward()` should be modified to support withdrawing yield from multiple assets proportionally. This could be implemented by calculating the share of yield from each asset and making proportional withdrawals from each asset's strategies. Alternatively, you could specify which assets to withdraw from and in what amounts.

Level team comments

Acknowledged. We expect to be calling `reward` at least once a week, so the yield that we can expect to share is roughly `(annualized yield APR / 52) * total USD value of reserves`. Illustrating a couple cases:

Assuming average yield of 4%: `4% / 52` = `0.08%` Assuming a high yield of 20%: `20% / 52` = `0.40%` Assuming an extremely high yield of 100%: `100% / 52` = `1.92%`

So assuming that at least 2% of reserves are stored in the most liquid base collateral, we should be able to cover extremely high yield cases. And since we only have 2 base collateral, we're comfortable assuming that the most liquid base collateral can cover cases where the yield is very high, or we call `reward` less frequently.

[L-15] `LVLUSD` burning causes permanent collateral accounting mismatch

When `LVLUSD` tokens are burned directly (not through the redemption process), the corresponding vault shares in the `BoringVault` are not burned. This creates a mismatch between the total supply of `LVLUSD` tokens and the vault shares representing the collateral backing them.

The issue affects the `RewardsManager`'s yield calculations, which are based on `total - vault.balanceOf(address(vault))`. With unburned vault shares, this calculation becomes inaccurate, leading to incorrect reward distributions since there is not as many `LVLUSD` as it is assuming.

```
contract
   lvlUSD is ERC20Burnable, ERC20Permit, IlvlUSDDefinitions, SingleAdminAccessControl
        // code
}
```

For example:

1. User deposits 100 `USDC` and gets 100 `LVLUSD`.
2. `BoringVault` mints 100 vault shares.
3. User burns 50 `LVLUSD` directly.
4. Result:
   - `LVLUSD` supply: 50
   - Vault shares: 100
   - `USDC` in vault: 100

```
function getAccruedYield(address[] calldata assets) public view returns
      (uint256 accrued) {
        uint256 total;

        for (uint256 i = 0; i < assets.length; i++) {
            // code
        }

        uint256 vaultShares = vault.balanceOf(address(vault));
        accrued = total - vaultShares;

        return accrued;
    }
```

While there's no direct financial incentive to burn `LVLUSD` tokens, there are legitimate use cases where `LVLUSD` might be burned (e.g., protocol upgrades,

token migrations). The mismatch between vault shares and `LVLUSD` supply affects the protocol's accounting and reward distribution mechanisms.

Recommendations:

The optimal solution should implement a hook in the `LVLUSD` contract that automatically calls the `BoringVault` to burn the corresponding vault shares whenever `LVLUSD` tokens are burned.

Level team comments

Acknowledged. Since lvlUSD is an immutable contract, we won't be able to affect it's logic. In addition, there is a path to recovery, since the admin timelock can call `vault.exit` to recover the collateral that used to back the burned `lvlUSD`.

[L-16] Vault deposit frontrun

The vault supports multiple strategies (e.g., Aave and Morpho). However, when users mint LVLUSD, funds are initially deposited only into the default strategy (like Aave). Strategy rebalancing (e.g., moving funds from Aave to Morpho) must be done manually by the admin using the following steps:

1. **Withdraw** tokens from the Aave strategy: The admin first calls the `withdraw()` function to pull the aTokens from Aave and convert them back to USDC, moving the tokens to the vault.

2. **Deposit** the USDC into the Morpho strategy: Once the USDC tokens are back in the vault, the admin can then use the `deposit()` function to move those tokens into the Morpho strategy.

The function to withdraw tokens from the Aave strategy is not the same as the deposit function — the admin first **withdraws** tokens, which are then available for rebalancing or depositing into a different strategy.

Here’s an example:

```
// Admin withdraws aTokens from Aave and converts them to USDC in the vault
function withdraw
  (address asset, uint256 amount) external requiresAuth notPaused {
    // Withdraw logic...
}

// Admin then deposits the USDC into Morpho strategy
function deposit(
  addressasset,
  addressstrategy,
  uint256amount
) external requiresAuth notPaused returns (uint256 deposited
    return _deposit(asset, strategy, amount);
}
```

In this way, the admin is able to reallocate funds between Aave and Morpho strategies by first withdrawing from Aave and then depositing into Morpho.

Issue: Front-Running Causes DoS

If a user redeems LVLUSD right before the admin's deposit transaction is mined, the vault sends collateral (e.g., USDC) to the `Silo` contract, **reducing the vault's token balance**.

Example:

- Vault has **exactly 20,000 USDC**.
- Admin tries to move **20,000 USDC** from Aave to Morpho.
- A user redeems **1 LVLUSD**, causing vault to send **1 USDC** to Silo → vault now has **19,999 USDC**.
- The deposit transaction fails due to insufficient funds.
- Even though the 1 USDC is still in the protocol (in Silo), it's inaccessible to the vault at the moment.

This allows any user to front-run and **grief the protocol**, preventing legitimate strategy reallocations.

Extended DoS: Large Holder Exploit

Even if the admin attempts to reallocate only a **partial amount**, such as **10,000 out of 20,000**, a malicious user holding **more than the remaining balance (e.g., 10,001 LVLUSD)** can **force the same DoS**:

- Vault holds 20,000 USDC.
- Admin initiates deposit of 10,000.
- Malicious user redeems 10,001 LVLUSD → vault sends 10,001 USDC to Silo.
- Vault now only has 9,999 USDC.
- Admin's 10,000 USDC deposit **fails**, despite having sufficient liquidity just a moment earlier.

This creates a denial of service (DoS) scenario, where the vault cannot reallocate funds properly due to a front-running attack or a malicious user exploiting the system.

Recommendations:

LevelMintingV2::initiateRedeem function when admin is depositing into a strategy

Level team comments

Great! Acknowledging this (if all actions happen in a multicall using private RPCs, this issue would be mitigated)

[L-17] Default strategies are not updated on strategy removal

When a strategy is removed using `removeAssetStrategy()`, the function removes the strategy from `assetToStrategy` mapping but fails to update the `defaultStrategies` array. This can lead to a situation where the default strategies list contains invalid or removed strategies, potentially causing issues with default deposits and withdrawals.