

Level A-1

Security Audit

May 1, 2025

Version 1.0.0

Table of Contents

- [Introduction](#)
- [Overall Assessment](#)
- [Specification](#)
- [Source Code](#)
- [Issue Descriptions and Recommendations](#)
- [Security Levels Reference](#)
- [Issue Details](#)
- [Disclaimer](#)

Introduction

This document includes the results of the security audit for Level's smart contract code as found in the section titled 'Source Code'. The security audit was performed by the Macro security team from April 18th to the 28th.

The purpose of this audit is to review the source code of certain Level Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer: While Macro's review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

The following is an aggregation of issues found by the Macro Audit team:

Severity	Count	Acknowledged	Won't Do	Addressed
Medium	7	2	-	5
Low	5	2	-	3
Code Quality	8	-	-	8
Informational	2	-	-	-

Level was quick to respond to these issues.

Specification

Our understanding of the specification was based on the following sources:

- Discussions with the Level team.
- Available documentation in the repository.
- Provided technical documentation.

Trust Model, Assumptions, and Accepted Risks (TMAAR)

Trusted Entities:

- Admin Multisig: Is the roles admin of other roles, can upgrade contracts, pause/unpause contracts, remove user roles, redeemable assets and oracles and disable both minting and redeeming. Has the highest authority of the protocol,

and is trusted to act in the best interest of it. Is a 5/8 timelocked multisig, with 4 internal signers and 4 trusted 3rd party signers, where all signers use cold wallets.

- Operator Multisig: Can pause/unpause contracts, disable minting/redeeming, and manage the vault via the vault manager. Is a 2/5 safe, where signers are internal team members and 4/5 wallets are cold.
- Treasury Multisig: Can call reward() on the reward manager, and is trusted to use assets received to mint more lvlUSD and distribute to stakers. Is a 3/4 safe, where signers are internal team members and all wallets are cold.
- Hexagate GateKeepers: Can pause/unpause contracts and disable minting. Is trusted to monitor the state of lvlUSD and react when needed in the best interest of the protocol. Is a EOA stored in AWS.

Trust Assumptions:

- Users must trust that the above trusted entities will act in the best interest of the protocol.
- Holders of Level are at risk of losing assets if any of the strategies the protocol interacts with experiences unexpected issues, like in the event of a hack or insolvency. The intention is that level money will only interact with proven and stable protocols and keep this risk to a minimum.

Source Code

The following source code was reviewed during the audit:

- **Scope**
 - **Repository:** [Level Money contracts](#)
 - Final Commit Hash: `5065d156f72b878db301907509eadb49760275d2`

Specifically, we audited the following contracts within this repository:

Source Code	SHA256
src/v2/LevelMintingV2.sol	c4021a45a1a843d7668145a288561076ba47198f1699ccc01ff91fa312fab6bc
src/v2/LevelMintingV2Storage.sol	c850e35f95312ca864747ba9eb616ab1af0a485f55d047e7ee5c280085774906
src/v2/auth/AuthUpgradeable.sol	1ad1e7e356a4bb5916401f5904c706ab39cb36756aff3bf322ab34e92d307e53
src/v2/auth/StrictRolesAuthority.sol	fe755fd0453ba4826937e330e48718ea980d50dc187388c06395a13af4ad2d72
src/v2/common/guard/PauserGuard.sol	924663aca6781dcb2f2988c187925d957c428649a3e582eabbbba2b3485226908
src/v2/common/guard/PauserGuarded.sol	6eea005248ca4eeeeee4ad40154a788e0561eba55151b2df055396c2b057ec205
src/v2/common/libraries/MathLib.sol	959dc0878368bc68cac2c41474774fab228e22ca0e80b0d52cdfd183c5841e99
src/v2/common/libraries/OracleLib.sol	407d083edfcf18243e2c5119ef4e50fdb9fd5e2326f2cd31a1363e56a0ef63b5
src/v2/common/libraries/StrategyLib.sol	1857441f2512588aa055acd2dc1f094e3dac54d6f355be42a5edef7ce015d210

Source Code	SHA256
src/v2/common/libraries/VaultLib.sol	73b1652096e4bf4ee47699c2923052ce59b9beff0b4b705ea1bdb3cd2a0aafbdc
src/v2/interfaces/AggregatorV3Interface.sol	f713c8782afe6e66f8ef7e8caf08c4226d1af1ab9928dcf998e73d70427b966c
src/v2/interfaces/aave/DataTypes.sol	98bed7b7ba50b12eb66c4fddd6f7877fa00867601db8612d9d6b6983ae341615
src/v2/interfaces/aave/IEACAggregatorProxy.sol	352cab2ff6d1921094904dac59c21fbb74c2b2e8b0b719b86db6b7e00f2ce6d0
src/v2/interfaces/aave/IPool.sol	d359240fec5f2d131eb1bb43322d881bf194db4fdfd2535990d29899c8a28479
src/v2/interfaces/aave/IPoolAddressesProvider.sol	6596af3544778a94f9c01a3d78baf53dcefb221ef2546c99edc5f845f8d1d8d1
src/v2/interfaces/level/IBoringVault.sol	cef86c4a389461583f6dd879f288dde55983bb1d9a4fd1e389689fdea72af20d
src/v2/interfaces/level/IERC4626Oracle.sol	568600b596c7f10892491e7087b22accf96a2a9db829199f2a60f253cbf19924
src/v2/interfaces/level/ILevelMintingV2.sol	e98e14b9417bd90130dc3d439d56336a88d77f939c2faf8c3b9be29eb2f2d7fc
src/v2/interfaces/level/IPause.sol	823da0726d6860c7a83e0094f3527d2fc34aa0dc17180a086586dd8741a8e0a2
src/v2/interfaces/level/IPauserAuthority.sol	00f201c6bdf2338968d8d06e366220a6f069fd612fea1d6ebdf227aa17c8bb64
src/v2/interfaces/level/IRewardsManager.sol	9a9d6ed54391e0af20c0984cc89c147feb6915e6970a58c2e936f5665f85922
src/v2/interfaces/level/IVaultManager.sol	d4691784fa172fbfae072f9d553ccfb4ef51dda2db325d2447e2e0f5fd8579d8

Source Code	SHA256
src/v2/interfaces/level/IlvlUSD.sol	5db26afef84f4e09fd1ff8fd8e76fa6cba905cadecb6a437006fbb471112d060
src/v2/interfaces/morpho/IMetaMorpho.sol	ace96ee95cd6b68442712a256e24f903932b26d278094b04655611748049dae6
src/v2/interfaces/morpho/IMetaMorphoV1_1.sol	cfbcb076c7d307ddbe0ea7430474a9148af830836449c5b0bed79ef1259b41532
src/v2/interfaces/morpho/IMorpho.sol	e8600e7cb5620deddd002b17b014deaa74aadadd030cd56b2c21b56e53d09b11
src/v2/interfaces/morpho/PendingLib.sol	ca53d7a1bd1c9fe0a9cf092f197cbb8b02192faf773060fc8b2dac1d432c909a
src/v2/lens/LevelReserveLens.sol	e93f82b6f01cef4710b1811bdf22b00b15cbc7b5332cc7a9bc6249825b6604
src/v2/oracles/AaveTokenOracle.sol	81571e3a2989e3e5098548219e9e8cb92951a3c2a7366752a8f9888cc4433a58
src/v2/oracles/ERC4626DelayedOracle.sol	549d137ecc2014f91d8b8734bc55a8f06a47e4a2414174ee277a46f8c10ecd62
src/v2/oracles/ERC4626Oracle.sol	c7c82cdfc071aad172bbf40c2fffb8a0d2eee98bc0a03bd8cec754d9091cb45
src/v2/oracles/ERC4626OracleFactory.sol	7f3f5917bec713649201c428caa5b9d0dab4c00a98c913209e2633bb22e3a4e9
src/v2/usd/BoringVault.sol	a08f6f69d92cd6f90526ff766c86e7998a0878a125943bebeed7ff2f51e12b63
src/v2/usd/RewardsManager.sol	95af9b753ce148cb0583b281d250f0ed3c038ff1209cc690d677d4b6c035853a
src/v2/usd/RewardsManagerStorage.sol	c9b6e5cce9f36e2874f1cde4f0390cfa3d45552055750dcd2fe281ebef788354

Source Code	SHA256
src/v2/usd/Silo.sol	d34a41e84eafbce858815e414e86083a74aba054c4c3f29a4046a81a0d4548f1
src/v2/usd/VaultManager.sol	c1092deb5ef7bb5f8d02cd8f70d2b35539c2984f7eaafe076a99f4b819698974
src/v2/usd/VaultManagerStorage.sol	a4c289ca09a769160a7e9d1a9f395cbb40ffd26dc1191064910517e554259378

Additionally we reviewed these deployment scripts:

Source Code	SHA256
script/v2/DeployLevel.s.sol	bb998a52fb684f29a0976108abd50c4835e45ea2a00501a489d043b6ae19853d
script/v2/DeployTestnet.s.sol	87862d3cc0967c38634c0e5e29942134a218ff0290ede456c3be5da517cfba5f
script/v2/DeploymentUtils.s.sol	0ea8eee346e4ee7cff8bbe5fe250f35c062eda818d88a9aed17192abbc634930
script/v2/lens/UpgradeLevelReserveLens.s.sol	ab4ba98bf3671310f4667fc93a6b709feb d19f5d4bd010f8e427a6423d7e2e03

* File hashes shown above are the final version of the reviewed contracts, corresponding to the final commit hash.

Issue Descriptions and Recommendations

Click on an issue to jump to it, or scroll down to see them all.

M-1 External reward received are stuck in the vault

M-2 Potential precision loss calculating total assets

M-3 Calculated rewards may be inaccurate if underlying depegs

M-4 USDT approvals can revert if current value is non-zero

M-5 Risk of being under collateralized after calling reward()

M-6 Suboptimal yield withdrawal strategy reduces protocol returns

M-7 Burnt lvlUSD not reflected in vault shares

L-1 Minting can be prevented by a griever

L-2 Oracle not updated consistently

L-3 Users who initiate a new redemption while having a completed cooldown must wait again unnecessarily

L-4 Potential overflow in `computeMint()` when using high decimal tokens

L-5 Rewards calculation may be inaccurate due to non-exhaustive asset tracking

I-1 Rebalancing can be grieved

I-2 Rewards may not be claimable without rebalancing assets

Q-1 Public functions with `requiresAuth` modifier can be bypassed in future upgrades

Q-2 Redundant denylist check in mint function leads to unnecessary gas costs

Q-3 Redundant subtraction operation in `completeRedeem()` can be simplified

Q-4 Inefficient implementation of `getAssets` in `VaultLib._getTotalAssets`

Q-5 Redundant calculation of deposited/withdrawn amounts in Vault operations

Q-6 Redundant token existence checks in BoringVault's enter/exit functions add unnecessary gas cost

- Q-7 Missing validation checks in strategy configuration functions
- Q-8 Inconsistent initialization pattern in `PauserGuarded` contract used by non-upgradeable contracts

Security Level Reference

We quantify issues in three parts:

1. The high/medium/low/spec-breaking **impact** of the issue:

- How bad things can get (for a vulnerability)
- The significance of an improvement (for a code quality issue)
- The amount of gas saved (for a gas optimization)

2. The high/medium/low **likelihood** of the issue:

- How likely is the issue to occur (for a vulnerability)

3. The overall critical/high/medium/low **severity** of the issue.

This third part – the severity level – is a summary of how much consideration the client should give to fixing the issue. We assign severity according to the table of guidelines below:

Severity	Description
(C-x) Critical	We recommend the client must fix the issue, no matter what, because not fixing would mean significant funds/assets WILL be lost.
(H-x) High	We recommend the client must address the issue, no matter what, because not fixing would be very bad, or some funds/assets will be lost, or the code's behavior is against the provided spec.
(M-x) Medium	We recommend the client to seriously consider fixing the issue, as the implications of not fixing the issue are severe enough to impact the project significantly, albeit not in an existential manner.
(L-x) Low	<p>The risk is small, unlikely, or may not be relevant to the project in a meaningful way.</p> <p>Whether or not the project wants to develop a fix is up to the goals and needs of the project.</p>
(Q-x) Code Quality	The issue identified does not pose any obvious risk, but fixing could improve overall code quality, on-chain composability, developer ergonomics, or even certain aspects of protocol design.
(I-x) Informational	Warnings and things to keep in mind when operating the protocol. No immediate action required.
(G-x) Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

Issue Details

M-1

External reward received are stuck in the vault

TOPIC	STATUS	IMPACT	LIKELIHOOD
Protocol Design	Acknowledged	Medium	High

Level money V2 manages positions held by it's vault using the [VaultManager contract](#). The vault manager is currently setup to handle different strategies limited to depositing or withdrawing into or out of either Aave or Morpho lending pools. There is an assumption that the vault will only receive assets from these direct interactions, however if protocols offer separate rewards tokens for interacting with their protocols, or any other reason, the manager has no way to interact with these tokens, nor include them as rewards to distribute to token holders. In the case of Morpho, their pools offer a native asset reward, but additionally rewards lenders Morpho tokens, that the vault can receive via the distributor contract. Aave also allows for external incentives that are claimable via its incentive controller. Currently there is no way for the manager to properly handle receiving or interacting with these tokens, unless it is upgraded to do so.

Remediations to Consider

Add methods to handle external rewards for Aave and Morpho, as well as for other potential protocols you may interact with, to prevent them from being locking in the vault and potentially offer additional rewards to lvlUSD holders.

RESPONSE BY LEVEL

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.

M-2

Potential precision loss calculating total assets

TOPIC	STATUS	IMPACT	LIKELIHOOD
Accounting	Fixed ↗	Medium	Low

In [StrategyLib](#) contract, total assets held in strategies for a specific asset are calculated via [getAssets\(\)](#), where the amount of receipt tokens owned is converted to the value of its associated underlying stable coin using pricing from the proper oracle. Before prices are used, the amount of receipt tokens held is converted from its decimals to the underlying stable coins decimals:

```
uint256 shares = receiptToken.balanceOf(vault);

uint256 sharesToAssetDecimals =
    shares.mulDivDown(10 ** ERC20(address(config.baseCollateral)).decimals(), 10 ** receiptToken.decimals());

(int256 assetsForOneShare, uint256 decimals) =
    OracleLib.getPriceAndDecimals(address(config.oracle), config.heartbeat);

assets_ = uint256(assetsForOneShare).mulDivDown(sharesToAssetDecimals, 10 ** decimals);
```

Reference: [StrategyLib.sol#L48-56](#)

In cases where they share the same decimals or the price is pegged 1:1 via rebasing like Aave's aTokens, this isn't an issue. However in the case where the receipt token differs from its underlying in decimals and its price isn't pegged, then converting to decimals before including the price can lead to a loss of precision. In the case of Morpho, it uses ERC4626 vaults with 18 decimals, and the price of its shares is based on yield, so will likely trend to increase relative to the underlying asset. Considering the expected

collateral assets will be the stable coins USDC and USDT, both using 6 decimals, this can lead to precision loss, and inaccurate assessment of held assets. This can effect both withdrawals as well as inaccurate rewards calculations when calling `reward()`.

Remediations to Consider

Multiply by price before converting to the underlying's decimals to prevent precision loss.

M-3

Calculated rewards may be inaccurate if underlying depegs

TOPIC	STATUS	IMPACT	LIKELIHOOD
accounting	Fixed ↗	Medium	Low

When rewards are calculated via `reward()`, asset value in the vault is calculated and related to the total vault shares to determine the amount of yield accrued, this yield is then withdrawn to the treasury, where it is trusted to mint more lvUSD with the received assets, and distribute it to lvUSD stakers. The main calculation for assets held for a given asset is handled by `getAccruedYield()` :

```
function getAccruedYield(address[] calldata assets) public view returns (uint256
    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
    }

    uint256 vaultShares = vault.balanceOf(address(vault));
```

```
    accrued = total - vaultShares;

    return accrued;
}
```

Reference: [RewardsManager.sol#L85-102](#)

`getAccruedYield()` sums the value held for each asset specified into a final total, however, `_getTotalAssets()` returns assets in terms of the strategies underlying token, rather than in terms of lvUSD. Considering each underlying token is intended to be a USD based stable coin, this is typically fine. In the case where an underlying token is depegged, this 1:1 assumption could result in overvaluing assets held, and thus value accrued, resulting in `reward()` pulling more assets out than it should and causing lvUSD to be under-collateralized.

Remediations to Consider

Convert each underlying asset into lvUSD via oracles, similar to how it is handled in `computeMint()` and `computeRedeem()`.

M-4 USDT approvals can revert if current value is non-zero

TOPIC	STATUS	IMPACT	LIKELIHOOD
Use Cases	Fixed ↗	High	Low

A [known issue with USDT](#) is that it requires the current approval amount for a protocol to be zero before setting to another non-zero value. When depositing into a protocol with either `_depositToAave()` or `_depositToMorpho()`, `vault.increaseAllowance()` is called to give the protocol approval to spend tokens:

```
function increaseAllowance(address token, address spender, uint256 amount) external  
    require(token.code.length != 0, "Token does not exist");
```

```
ERC20(token).safeApprove(spender, amount); //TODO replace with forceapprove c
}
```

Reference: [BoringVault.sol#L50-53](#)

As mentioned in the TODO, `safeApprove` should be replaced by `forceApprove`, since Solady's `safeApprove` does not handle the edge case of USDT approvals, if a protocol being interacted with does not consume all USDT approvals when depositing, the lingering approval will cause future calls to `increaseAllowance` to fail. Considering USDT will be one of the main collateral tokens used, this edge case should be addressed to ensure deposits do not revert. Although deposits are allowed to revert in mint, the end result would be USDT directly transferred to the vault and not generating the expected yield that it would if deposited into the set default strategy. Additionally the vault would be blocked from depositing USDT into protocols that this occurs, until an external `BoringVault.manage()` call is made to revoke unspent USDT approvals.

Remediations to Consider

Use Solady's `safeApproveWithRetry()`, or another resolution from an trusted repo. Alternatively you can set approvals to zero before calling `increaseAllowance()`

M-5 Risk of being under collateralized after calling reward()

TOPIC	STATUS	IMPACT	LIKELIHOOD
Insolvency	Fixed ↗	High	Low

After `reward()` is called, based on price evaluations of assets held in the vault, all excess value not required to collateralize lvlUSD is sent to the treasury. This means that the value of the vaults assets equals the value of lvlUSD. This makes sense, however there can be near-term price updates that could then result in the value of assets held to not collateralize circulating lvlUSD, particularly for assets held in morpho pools where bad debt could lead to a decrease in price. Large price deviations


of collateralized assets held in the vault is always a risk that could lead to depegging, which is why stable protocols and use of stable coins are the expected strategies, and a buffer of generated yield helps mitigate potential downside price fluctuations. In the case of directly after `reward()` is called, there is no longer an asset buffer and could lead to lvUSD being under collateralized, with little direct incentive to mint more until the peg is re-achieved, as minting always costs \$1 USD value of collateral asset. Maintaining a little asset value buffer may be advised based on the value of assets held in more volatile or risky pools/protocols to reduce risk of under collateralization and thus depegging.

Remediations to Consider

Add a optional percent or flat amount buffer parameter to `reward()` that keeps a specified amount of excess assets to potentially sufficiently over collateralize lvUSD to reduce downside risk.

M-6

Suboptimal yield withdrawal strategy reduces protocol returns

TOPIC	STATUS	IMPACT	LIKELIHOOD
Rewards Management	Fixed 	Medium	Medium

The `RewardsManager.reward()` function implements a suboptimal strategy for withdrawing yield, resulting in reduced returns for the protocol over time. The current implementation withdraws yield by converting receipt tokens to base tokens without first utilizing available base tokens in the vault, leading to compounding interest losses.

The root cause lies in the `reward()` function's withdrawal logic. When withdrawing accrued yield, the function:

1. Calculates total accrued yield across all strategies

2. Directly converts receipt tokens to base tokens to fulfill the yield amount

3. Transfers the withdrawn amount to the treasury

This approach fails to optimize for maximum yield because it doesn't prioritize using already-available base tokens in the vault before converting receipt tokens that are actively earning interest.

Consider this scenario:

Initial state:

- Vault holds 50 USDC (base token) and 950 sUSDC (Steakhouse USDC, receipt token worth \$950)
- After one day at 10% APY:
 - The 950 sUSDC appreciates to \$1,045
 - Total vault value: \$1,095
 - Accrued yield: \$95

Current behavior:

- Converts ~86.3 sUSDC to USDC to withdraw \$95 yield
- Remaining vault value: 50 USDC + 863.7 sUSDC = \$1,000
- After another day at 10% APY: Total value grows to \$1,086.4

Optimal behavior would:

- Use the existing 50 USDC first
- Convert only 40.9 sUSDC to obtain the remaining \$45 needed
- Leave 909.1 sUSDC in the vault
- After another day at 10% APY: Total value grows to \$1,090.9

The difference of \$4.5 per cycle compounds over time, which proves the suboptimal yield of the protocol.

Remediations to Consider Implement a two-step withdrawal process in the `reward()` function:

1. First, check and withdraw available base tokens from the vault
2. Only convert receipt tokens if additional funds are needed

M-7

Burnt lvlUSD not reflected in vault shares

TOPIC	STATUS	IMPACT	LIKELIHOOD
accounting	Acknowledged	Medium	Medium

The expected invariant is that the shares of lvlUSD should equal the total balance of lvlUSD. The [LvlUSD token](#) is burnable, which means that the owner of the tokens can destroy their tokens by calling `burn()` or `burnFrom()` on the token, lowering the supply of lvlUSD. The vault supply is only informed of lvlUSD minting and burning via `LevelMinting.sol`, so burning tokens in this way does not result in the equivalent vault shares to be burned, causing the invariant to be broken. The effect of this can be seen in the case of `RewardManager`'s `reward()` where it determines accrued assets based on the vaults total supply, but in the case of burnt lvlUSD tokens this supply would be higher than lvlUSD, so the amount of assets required to collateralize lvlUSD should be lower than is calculated.

Remediations to Consider

Consider updating the vault shares on burning, evaluate assets required to collateralize lvlUSD based on it's own supply rather than vault shares, or watch burning events and have the admin multisig burn a proportional amount of vault shares via `vault.exit()`

RESPONSE BY LEVEL

We'll watch for burn events and have the admin timelock can call `vault.exit` to recover the collateral that used to back the burned `lvlUSD`. Since `lvlUSD` is an immutable contract, we won't be able to affect it's logic.

🔑 Minting can be prevented by a griever

TOPIC	STATUS	IMPACT	LIKELIHOOD
Griefing	Fixed ↗	Medium	Low

When minting `lvlUSD` via `mint()` there is a limit to the amount of `lvlUSD` that can be minted in each block:

```
mintedPerBlock[block.number] += lvlUsdMinted;

if (mintedPerBlock[block.number] > maxMintPerBlock) revert ExceedsMaxBlockLim:
```

Reference: [LevelMintingV2.sol#L56-58](#)

The intention behind constraining the amount of `lvlUSD` per block is to prevent large influxes in cases of depegging or arbitrage opportunities. However, if someone were to want to prevent users from being able to mint `lvlUSD`, or prevent a net positive issuance over time, they could continually frontrun and mint the max limit per block, preventing additional mints from occurring that block. They could then initiate a redemption via `initiateRedeem()` to redeem the minted `lvlUSD` back for their collateral after waiting the `cooldownDuration` (default of 5 minutes). If a user was so inclined and had the funds they repeat this for each block within the `cooldownDuration` and loop redeemed assets back to continue the attack. Notably, `lvlUSD` has a deny list, preventing transfers to or from any address on this list, however only the address when minting is checked if they are on this deny list, and when calling

`completeRedeem()` any beneficiary can be set, allowing the griever to circumvent any attempts to use the deny list to prevent this attack. This attack is quite costly to implement, and has no obvious incentives to do so, it is good to be aware that it exists.

Remediations to Consider

Consider checking if `msg.sender` is on the deny list when calling `completeRedeem()`, it is important to note that doing so could result in unredeemable assets in the case where the user is added to the deny list before completing their redemption, locking assets in the silo.

L-2 Oracle not updated consistently

TOPIC	STATUS	IMPACT	LIKELIHOOD
Oracles	Acknowledged	Low	Low

When minting lvlUSD before it pulls the price from the oracle it attempt to update beforehand:

```
if (isLevelOracle[order.collateral_asset]) {
    oracles[order.collateral_asset]._tryUpdateOracle();
}
```

Reference: [LevelMinting.sol#L50-52](#) This is also done for VaultManager's `_deposit()` and `_withdraw()` functions. This is done to ensure the price is accurate, specifically if the oracle is a `ERC4626DelayedPriceOracle`. In `initiateRedeem()` however, the oracle is not attempted to be updated, resulting in a potentially stale pricing if other calls that do update the oracle are not called. Additionally the same occurs in `RewardsManager.getAccruedYield()` (<https://github.com/Level-Money/contracts/blob/9c69738aac06ef0edb3be6a7c9f4c6ef320f8cef/src/v2/usd/RewardsManager.sol#L84-L102>) .

Remediations to Consider

Consider attempting to update the oracle in `initiateRedeem()` to be more consistent and have potentially more up to date prices.

RESPONSE BY LEVEL

This would be true if we let users redeem lvlUSD for non-base collateral (ie Morpho vault tokens). Since redemptions are only going to be in base collateral (for which we only use Chainlink oracles), `_tryUpdateOracle` will never be called.

L-3

Users who initiate a new redemption while having a completed cooldown must wait again unnecessarily

TOPIC	STATUS	IMPACT	LIKELIHOOD
Redemption Process	Acknowledged	Low	Low

In `LevelMintingV2.sol`, the `initiateRedeem()` function allows users to start the redemption process for their lvlUSD tokens. However, when users attempt to initiate a new redemption while having a completed cooldown period from a previous redemption that hasn't been claimed yet, the function still enforces a new cooldown period instead of reverting.

This creates a suboptimal user experience where users who forget to claim their completed redemption first will have their funds unnecessarily locked for another cooldown period when initiating a new redemption.

Remediations to Consider Add a check at the start of `initiateRedeem()` to ensure users cannot initiate new redemptions if they have completed but unclaimed redemptions. Alternatively, at the UI level, implement a warning message to notify users that initiating a new redemption will reset the cooldown period for their unclaimed redemptions.

We will update this in the UI, and add a comment to the smart contract code.



Potential overflow in `computeMint()` when using high decimal tokens

TOPIC	STATUS	IMPACT	LIKELIHOOD
Arithmetic Overflow	Fixed ↗	Low	Low

The `computeMint()` function in `LevelMintingV2.sol` performs calculations to determine the amount of LVLUSD to mint based on collateral value. When using high decimal tokens (particularly receipt tokens for stablecoins like PYUSD), the intermediate calculations can potentially overflow due to large exponents in the numerator.

The issue occurs in the calculation: `collateralAmount * numerator / denominator`

For receipt tokens, the numerator is the product of `10 ** LVLUSD_DECIMAL`, `collateralOraclePrice`, and `underlyingPrice`, while the denominator includes the corresponding decimal adjustments.

With high decimal tokens like PYUSD (18 decimals) and its receipt token sPYUSD (18 decimals):

- If `collateralOraclePrice` = 1.1e18 (sPYUSD/PYUSD)
- And `underlyingPrice` = 1e18 (PYUSD/USD)
- The numerator would be ~1.1e56

This means a collateral amount of 105,265 PYUSD (~\$115,792) would cause an overflow in the intermediate multiplication.

While users can mitigate this by splitting deposits into multiple transactions, this workaround results in poor UX. More critically, when the `msg.sender` is a contract that cannot configure collateral amounts, this issue can lead to protocol integration problems.

Remediations to Consider Modify the calculation to reduce intermediate values by factoring out common decimals between numerator and denominator:

```
numerator = collateralOraclePrice * underlyingPrice;
denominatorDecimals = collateralToken.decimals() + collateralOracleDecimals + u

if (denominatorDecimals > LVLUSD_DECIMAL) {
    return collateralAmount.mulDivDown(numerator, 10 ** (denominatorDecimals -
} else {
    return collateralAmount * numerator * (10 ** (LVLUSD_DECIMAL - denominatorDecimals))
}
```

This maintains the same mathematical relationship while reducing the size of intermediate values.

L-5

Rewards calculation may be inaccurate due to non-exhaustive asset tracking

TOPIC	STATUS	IMPACT	LIKELIHOOD
Rewards Calculation	Fixed ↗	Low	Low

The `reward()` function in `RewardsManager` contract relies on an array of assets passed as input to calculate accrued yield. This design is problematic because it allows for potential calculation inaccuracies if the caller does not provide a complete list of all assets that should be included in the yield computation.

The root issue lies in the fact that while strategies are set per asset via `setAllStrategies()`, there is no mechanism to ensure that all relevant assets are

included when `reward()` is called. This could lead to lower yield calculation if some assets are accidentally omitted from the input array.

Remediations to Consider Track all valid assets internally in the contract rather than relying on input parameters:

```
contract RewardsManager {
+   address[] public validAssets;

    function setAllStrategies(address asset, StrategyConfig[] memory strategies
+       if (!_containsAsset(validAssets, asset)) {
+           validAssets.push(asset);
+       }
    // ... existing code ...
}

-   function reward(address[] calldata assets) external notPaused requiresAuth
+   function reward() external notPaused requiresAuth {
-       uint256 accrued = getAccruedYield(assets);
+       uint256 accrued = getAccruedYield(validAssets);
    // ... rest of the function ...
}
}
```

This ensures that yield calculations always consider all assets that have been properly configured in the system.

I-1 Rebalancing can be grieved

TOPIC

Griefing

IMPACT

Informational *

The [VaultManager](#) is able to control the vault to directly withdraw and deposit assets into supported protocols via [deposit\(\)](#) and [withdraw\(\)](#), allowing them to withdraw assets from one protocol and deposit them into another. However, since when

redemptions are initiated, assets are also withdrawn from the vault first, then from default withdrawal strategies, it is possible that redemptions can cause rebalances to fail if the expected assets are no longer held in the vault or the expected amount is no longer in a default withdrawal strategy. This is not likely to arise as much of an issue, but ensuring strategists execute their rebalances in a single atomic transaction, and potentially using a private RPC is suggested to prevent this from occurring.

RESPONSE BY LEVEL

We plan on rebalancing through atomic transactions with private RPC URLs.

I-2 Rewards may not be claimable without rebalancing assets

TOPIC

Accounting

IMPACT

Informational *

`RewardsManager.reward()` (<https://github.com/Level-Money/contracts/blob/9c69738aac06ef0edb3be6a7c9f4c6ef320f8cef/src/v2/usd/RewardsManager.sol#L36-L52>) function calculates rewarded assets accrued, and pulls these assets from the vault into the treasury. Only a single asset is pulled from, the 0th index of the assets array parameter, to cover the rewards owed across potentially many different assets used in the protocol, which means that there is required to be enough free liquidity of that asset to cover all accrued rewards for the protocol. If this is not the case, coordination between the strategist freeing up assets before the treasury can call reward may be required. It is suggested there be enough assets available to cover rewards and for `reward()` to be called semi-frequently

RESPONSE BY LEVEL

Since we expect to call reward at least once a week, we expect amount of rewards claimed as a percentage of the reserves to be relatively low. To take an example, if the APR on lending protocols is 10% (which is rare), the weekly percentage of reserves we'd need to redeem as rewards is less than 20 bps.

Q-1

Public functions with `requiresAuth` modifier can be bypassed in future upgrades

TOPIC

Access Control

STATUS

Fixed [↗](#)

QUALITY IMPACT

Low

Several functions including `LevelMintingV2.addMintableAsset()`, `LevelMintingV2.addRedeemableAsset()`, `LevelMintingV2.addOracle()`, `LevelMintingV2.removeOracle()`, `LevelMintingV2.setHeartBeat()`, `StrictRolesAuthority.setUserRole()`, and `StrictRolesAuthority.removeUserRole()` are marked as `public` while being protected by the `requiresAuth` modifier. This creates a potential security risk in future upgrades of the contract.

The issue stems from how function selectors work in Solidity. The `requiresAuth` modifier checks `msg.sig` to determine if the caller has the required authorization. However, in future upgrades, if a new function is added that internally calls any of these `public` functions, that internal call would bypass the `requiresAuth` check since `msg.sig` would be the selector of the outer function, not the internal one being called.

While this is not an immediate vulnerability since there are currently no functions that make such internal calls, it represents poor security architecture that could be exploited in future upgrades if proper care is not taken.

Consider changing the visibility of these functions from `public` to `external` to prevent them from being called internally.

Q-2

Redundant denylist check in mint function leads to unnecessary gas costs

TOPIC

Redundant logic

STATUS

Fixed [↗](#)

QUALITY IMPACT

Low

The `LevelMintingV2.mint()` function performs an explicit check for denylisted addresses by calling `lvlusd.denylisted(msg.sender)`. However, this check is redundant since the lvlUSD token already performs the same validation in its `_beforeTokenTransfer()` hook, which is called during the minting process.

Consider removing the redundant check:

```
function mint(Order calldata order) external requiresAuth notPaused returns (u:
-   if (lvlusd.denylisted(msg.sender)) revert DenyListed();

    // ... rest of the function
```

Q-3

Redundant subtraction operation in `completeRedeem()` can be simplified

TOPIC

Code Simplification

STATUS

Fixed [↗](#)

QUALITY IMPACT

Low

In `LevelMintingV2.completeRedeem()`, the code first assigns `pendingRedemption[msg.sender][asset]` to `collateralAmount` and then subtracts `collateralAmount` from `pendingRedemption[msg.sender][asset]`. Since these values are equal and the intent is to clear the pending redemption, this operation can be simplified to a direct assignment to zero.

Replace the subtraction operation with a direct assignment to zero since we're clearing the entire pending redemption:

```
collateralAmount = pendingRedemption[msg.sender][asset];  
- pendingRedemption[msg.sender][asset] -= collateralAmount;  
+ pendingRedemption[msg.sender][asset] = 0;
```

Q-4 Inefficient implementation of `getAssets` in `VaultLib._getTotalAssets`

TOPIC	STATUS	QUALITY IMPACT
Redundant logic	Fixed ↗	Low

In `VaultLib._getTotalAssets`, the function iterates through each strategy to call `StrategyLib.getAssets(config, address(vault))` individually. However, `StrategyLib` already provides an overloaded `getAssets` function that accepts an array of `StrategyConfig` and performs the iteration internally. Using the array version would eliminate an unnecessary loop layer and slightly improve gas efficiency.


Remediations to Consider Replace the manual iteration with a direct call to the array version of `getAssets`:

```
function _getTotalAssets(BoringVault vault, StrategyConfig[] memory strategies)  
    internal  
    view  
    returns (uint256 total)  
{  
    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));  
-   }  
+   totalForAsset += StrategyLib.getAssets(strategies, address(vault));  
}
```



```
    return totalForAsset;  
}
```

Q-5 Redundant calculation of deposited/withdrawn amounts in Vault operations


TOPIC	STATUS	QUALITY IMPACT
Redundant logic	Fixed 	Low

In `VaultLib.sol`, there are multiple instances where the code calculates deposited/withdrawn amounts by taking the difference between balance before and after the operation. However, these calculations are redundant since the amounts are always equal to the input parameter `amount` in the functions `_depositToAave()`, `_depositToMorpho()`, and `_withdrawFromMorpho()`.

Consider returning the input `amount` parameter instead of calculating the difference. This saves gas by eliminating redundant storage reads and arithmetic operations.

Similar changes should be applied to `_depositToMorpho()` and `_withdrawFromMorpho()`.

Q-6 Redundant token existence checks in BoringVault's enter/exit functions add unnecessary gas cost

TOPIC	STATUS	QUALITY IMPACT
Redundant logic	Fixed 	Low

The BoringVault contract implements redundant token existence checks in both `enter()` and `exit()` functions using `require(address(asset).code.length != 0, "Token does not exist")`. These checks are unnecessary because `SafeTransferLib.safeTransfer()` already includes this check in the [Solmate version](#) being used.

Consider removing the redundant token existence checks from both functions since they add unnecessary gas overhead.



Missing validation checks in strategy configuration functions

TOPIC

Input Validation

STATUS

Fixed

QUALITY IMPACT

Low

In both `RewardsManager.setAllStrategies()` and `VaultManager.setDefaultStrategies()`, there are missing validation checks for strategy configuration parameters. While `RewardsManager` validates `baseCollateral`, it doesn't verify the `category` field. Conversely, `VaultManager` checks if the category is valid but doesn't validate if the `baseCollateral` matches the input asset. Consider adding validation in both functions



Inconsistent initialization pattern in `PauserGuarded` contract used by non-upgradeable contracts

TOPIC

Initialization Pattern

STATUS

Fixed

QUALITY IMPACT

Low

The `PauserGuarded` contract follows an upgradeable contract pattern with initialization functions, but it's being used by non-upgradeable contracts like `BoringVault`. The `__PauserGuarded_init()` function lacks the `onlyInitializing` modifier from OpenZeppelin's upgradeable pattern, which appears to be intentionally removed to allow non-upgradeable contracts like `BoringVault` to use it.

This creates an inconsistent and potentially confusing pattern where the `PauserGuarded` contract uses upgradeable contract patterns (storage gaps, initialization functions), but removes key safety checks (`onlyInitializing`) to support non-upgradeable contracts. This mixed usage leads to `BoringVault` contract having bloated features for upgradability while not being upgradeable. Moreover, this can create confusion for future contracts that inherit from the `PauserGuarded` contract.

Consider creating a non-upgradeable version of `PauserGuarded` specifically for non-upgradeable contracts like `BoringVault`. Alternatively, maintain the upgradeable pattern properly by keeping `onlyInitializing` and making `BoringVault` upgradeable if that flexibility is desired.

Disclaimer

Macro makes no warranties, either express, implied, statutory, or otherwise, with respect to the services or deliverables provided in this report, and Macro specifically disclaims all implied warranties of merchantability, fitness for a particular purpose, noninfringement and those arising from a course of dealing, usage or trade with respect thereto, and all such warranties are hereby excluded to the fullest extent permitted by law.

Macro will not be liable for any lost profits, business, contracts, revenue, goodwill, production, anticipated savings, loss of data, or costs of procurement of substitute goods or services or for any claim or demand by any other party. In no event will Macro be liable for consequential, incidental, special, indirect, or exemplary damages arising out of this agreement or any work statement, however caused and (to the fullest extent permitted by law) under any theory of liability (including negligence), even if Macro has been advised of the possibility of such damages.

The scope of this report and review is limited to a review of only the code presented by the Level team and only the source code Macro notes as being within the scope of Macro's review within this report. This report does not include an audit of the deployment scripts used to deploy the Solidity contracts in the repository corresponding to this audit. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. In this report you may through hypertext or other computer links, gain access to websites operated by persons other than Macro. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such websites' owners. You agree that Macro is not responsible for the content or operation of such websites, and that Macro shall have no liability to your or any other person or entity for the use of third party websites. Macro assumes no responsibility for the use of third party software and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.