Level Money contracts

Security Review

Cantina Managed review by:
Milotruck, Security Researcher
Xiaoming90, Security Researcher
Delvir0, Junior Security Researcher
Tchkvsky, Junior Security Researcher

October 5, 2024

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 Critical Risk . . . 4
  - 3.1.1 Missing call to `computeCollateralOrlvlUSDAmount()` in `LevelMinting.mintDefault()` 4
  - 3.1.2 `mint()`, `initiateRedeem()`, and `redeem()` lack access control . . . 4
- 3.2 Medium Risk . . . 5
  - 3.2.1 `LevelMinting` contract can be used as a premeditated attack in case a stablecoin depegs to drain the contract . . . 5
  - 3.2.2 Deposit and redeem functions can be DOSed by exploiting the replay prevention mechanism . . . 6
  - 3.2.3 Removing assets from `_supportedAssets` bricks pending redemptions in `LevelMinting` 7
  - 3.2.4 Missing setter for `cooldownDuration` in `LevelMinting` . . . 8
  - 3.2.5 Stablecoins residing on `LevelMinting` contract can be arbitraged if instant redemption is enabled . . . 8
  - 3.2.6 Deposit and redeem functions can be DOSed by exceeding the deposit and redemption cap . . . 9
- 3.3 Low Risk . . . 9
  - 3.3.1 Denylisted user can still send `lvlUSD` via `LevelMinting.mint` . . . 9
  - 3.3.2 Missing approval to Karak's vault in `LevelReserveManager.startRedeemFromKarak()` . 9
  - 3.3.3 Missing input validation for `LevelMinting.initiateRedeem()` . . . 10
  - 3.3.4 Changing `cooldownDuration` in `LevelMinting` does not affect ongoing pending redemptions . . . 10
  - 3.3.5 Collateral could be redeemed without locking user's lvlUSD . . . 11
  - 3.3.6 Ineffective segregation of control in `LevelReserveManager` . . . 12
- 3.4 Informational . . . 13
  - 3.4.1 Unused Code . . . 13
  - 3.4.2 Karak vault shares can be transferred out of the `LevelReserveManager` contract . . . 13
  - 3.4.3 Minor improvements . . . 14
  - 3.4.4 Pending redemptions in `LevelMinting` can exceed the protocol's balance . . . 14
  - 3.4.5 Impact of different block time on `maxMintPerBlock` and `maxRedeemPerBlock` . . . 15

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 Sep 9th to Sep 16th the Cantina team conducted a review of level-money-contracts on commit hash 35bcfa90. The team identified a total of **19** issues in the following risk categories:

- Critical Risk: 2
- High Risk: 0
- Medium Risk: 6
- Low Risk: 6
- Gas Optimizations: 0
- Informational: 5

3 Findings

3.1 Critical Risk

3.1.1 Missing call to `computeCollateralOrlvlUSDAmount()` in `LevelMinting.mintDefault()`

Severity: Critical Risk

Context: LevelMinting.sol#L226-L233

Description: `Bridge.mintDefault()`, directly passes the `order` struct specified by the user to `_mint()`:

```
function mintDefault(
    Order memory order
) external virtual onlyMinterWhenEnabled {
    if (msg.sender != order.benefactor) {
        revert MsgSenderIsNotBenefactor();
    }
    _mint(order, _route);
}
```

However, this means that `lvlusd_amount` and `collateral_amount` do not have any input validation. Therefore, users can specify `lvlusd_amount` as anything and `collateral_amount = 1` to mint infinite lvlUSD.

Recommendation: Add the following call to `computeCollateralOrlvlUSDAmount()`:

```
  function mintDefault(
      Order memory order
  ) external virtual onlyMinterWhenEnabled {
      if (msg.sender != order.benefactor) {
          revert MsgSenderIsNotBenefactor();
      }
-     _mint(order, _route);
+     Order memory _order = computeCollateralOrlvlUSDAmount(order);
+     _mint(_order, _route);
  }
```

Level Money: Fixed in commit 532771c9.

Cantina: Verified, the missing call was added.

3.1.2 `mint()`, `initiateRedeem()`, and `redeem()` lack access control

Severity: Critical Risk

Context: LevelMinting.sol#L193, LevelMinting.sol#L254

Description: The issue simply lies in the fact that any caller of the `mint()`, `initiateRedeem()` or `redeem()` function can set the `order.benefactor` to an arbitrary address. This enables an attacker to set `order.benefactor` to any address that holds lvlUSD and the `order.beneficiary` to the attacker's address which transfers the collateral to the attacker.

Proof of concept:

```
function test_bobRedeemFromUser() public {
    console.log("balance of bob", stETHToken.balanceOf(bob)); // = 0

    vm.startPrank(owner);
    LevelMintingContract.setCheckRedeemerRole(false);
    vm.stopPrank();

    (
        ILevelMinting.Order memory order,
        ILevelMinting.Route memory route
    ) = mint_setup(_stETHToDeposit, _stETHToDeposit, 1, false); // note sets order for default values
↪   (benefactor)
    LevelMintingContract.mint(order, route);


    ILevelMinting.Order memory order2 = ILevelMinting.Order({
        order_type: ILevelMinting.OrderType.REDEEM,
        nonce: 3,
        benefactor: beneficiary, //note beneficiary received token in above mint_setup
        beneficiary: bob, //note set to bob in order to steal the funds
        collateral_asset: address(stETHToken),
        lvlusd_amount: _stETHToDeposit, // note to match mint_setup
        collateral_amount: _stETHToDeposit // note to match mint_setup
    });
    vm.startPrank(beneficiary);
    lvlusdToken.approve(address(LevelMintingContract), _stETHToDeposit);

    vm.startPrank(bob);
    LevelMintingContract.initiateRedeem(order2);
    vm.warp(8 days);
    LevelMintingContract.completeRedeem(address(stETHToken));

    console.log("balance of bob", stETHToken.balanceOf(bob)); // = 50e18
}
```

Note: *the attack will be possible if:*

1. *The victim has residual lvlUSD allowance to the `LevelMinting` contract or ...*
2. *The attacker frontruns the victims `mint()` or `redeem()` call after the approval transaction.*

Recommendation: Implement access control in `mint()`, `initiateRedeem()`, and `redeem()` which blocks an user from setting the `order.benefactor` to any address than `msg.sender` or if the user has an allowance.

Level Money: Fixed in commit 5afc1851.

Cantina: Verified, `mint()`, `initiateRedeem()` and `redeem()` now check that `order.benefactor` is set to `msg.sender`.

3.2 Medium Risk

3.2.1 `LevelMinting` contract can be used as a premeditated attack in case a stablecoin depegs to drain the contract

Severity: Medium Risk

Context: LevelMinting.sol#L277

Description: In addition to the arbitrage finding (see the issue "Stablecoins residing on `LevelMinting` contract can be arbitraged if instant redemption is enabled"), swapping a de-pegged stablecoin can be done via an alternative way. This way enables us to deposit an `x` amount of a de-pegged stable and:

1. Receive a non de-pegged asset.
2. Receive an value amount that is higher than the deposited value.

When calling `initiateRedeem()`, we specify which collateral we want to redeem, which is then stored.

1. This is not the case when initiating a deposit, the collateral type is not stored while receiving lvlUSD.
2. Next to that, it's possible to call the `initiateRedeem()` to start the cooldown period without having any collateral deposited.

Since stable collateral and lvlUSD are swapped 1:1, this enables the following planned attack in case a depeg occurs:

1. Alice calls `initiateRedeem()` for all collateral types (must specify the amount to withdraw).
2. USDT depegs after e.g. 6 months to $0.50 for simplicity .
3. Alice swaps 100 UDSC to 200 USDT on Uniswap and calls `LevelMinting.deposit()`, receiving 200 lvlUSD.
4. Alice then immediately calls `LevelMinting.completeRedeem(address USDC)` where she receives 200 USDC.
5. Protocol ends up with 200USDT at $0.50 (100 loss).

Note: *Step 1 can be done with multiple wallets that Alice holds with each their own specified amount to ensure that she will be able to swap as many as possible.*

Recommendation: It's good to note that points 1. and 2. both introduce a vulnerability on their own, when combined the severity increases to such attack. Hence both need to be fixed:

Recommendation: Point 1:

- Consider to store the deposited asset type which is then also used to redeem.
- Alternatively, implement an oracle to exchange lvlUSD at a price calculated according to the asset.

Point 2:

- The issue "Collateral could be redeemed without locking user's lvlUSD" needs to be fixed for this.

Level Money: Fixed in commit 4437b47b.

Cantina: Verified, Chainlink oracles are now used to adjust the amount of lvlUSD or collateral that is minted/redeemed should a depeg occur:

- If a collateral asset has a negative depeg, the amount of lvlUSD minted will be discounted based on that collateral's price.
- If a collateral asset has a positive depeg, the amount of collateral redeemed will be discounted.

3.2.2 Deposit and redeem functions can be DOSed by exploiting the replay prevention mechanism

Severity: Medium Risk

Context: LevelMinting.sol#L200, LevelMinting.sol#L258

Description: This issue affects both minting and redemption, as both use the same `_deduplicateOrder` function to prevent replay.

The `_deduplicateOrder` function relies on the `order.benefactor` and `order.nonce` values to prevent replaying of orders. The `order.benefactor` and `order.nonce` can be set to an arbitrary value when the `mint` function is executed.

Assuming that Alice executes a `mint()` TX with `lvlusd_amount`, `order.benefactor` and `order.nonce` set to `1000e18`, `Bob` and `999`, respectively. A malicious user could front-run Alice's TX and submit a similar TX with `lvlusd_amount`, `order.benefactor` and `order.nonce` set to `1 wei`, `Bob` and `999`.

Thus, when Alice's TX is executed, it will revert as the nonce has already been "used".

Malicious users could keep repeating this attack to DOS Alice. The cost of attack might be low if this can be executed on an L2 EVM environment where front-running is possible (***sidenote:*** *front-running might not be possible on some L2 chains*).

The impact will be greater when used for DOS redemptions while the cooldown period is enabled since users will need to wait another 7 days to withdraw.

Recommendation: Consider removing nonces and `_deduplicateOrder()` altogether, as they are not needed anymore since signature verification was removed. If nonces are removed, `completeRedeem()` needs to be changed as such too:

```
- userCooldown.cooldownEnd = 0;
- cooldowns[msg.sender][token] = userCooldown;
+ delete cooldowns[msg.sender][token];
```

Level Money: Fixed in commit 5afc1851.

Cantina: Verified, nonces and `_deduplicateOrder()` have been removed.

3.2.3 Removing assets from `_supportedAssets` bricks pending redemptions in `LevelMinting`

Severity: Medium Risk

Context: LevelMinting.sol#L559, LevelMinting.sol#L372-L377

Description: In `LevelMinting`, the `_transferToBeneficiary()` function checks if the `asset` to be transferred is in the `_supportedAssets` set:

```
if (!_supportedAssets.contains(asset)) revert UnsupportedAsset();
```

`_transferToBeneficiary()` is used to transfer collateral assets to users upon withdrawal. As such, once a collateral asset is removed from `supportedAssets`, it cannot be withdrawn by users anymore as this check would fail.

This is problematic as:

- All pending redemptions with that asset as collateral cannot be completed.
- If the protocol still owns some of that collateral asset, it cannot be withdrawn forever.

This makes it such that an asset cannot be removed from `supportedAssets` unless the protocol's balance for that asset is zero.

Recommendation: Consider adding another set that tracks all past and present collateral assets:

```
  function addSupportedAsset(
      address asset
  ) public onlyRole(DEFAULT_ADMIN_ROLE) {
      if (
          asset == address(0) ||
          asset == address(lvlusd) ||
          !_supportedAssets.add(asset)
      ) {
          revert InvalidAssetAddress();
      }
+     _redeemableAssets.add(asset);
+
      emit AssetAdded(asset);
  }
```

The check in `_transferToBeneficiary()` can then be changed to use `_redeemableAssets`:

```
- if (!_supportedAssets.contains(asset)) revert UnsupportedAsset();
+ if (!_redeemableAssets.contains(asset)) revert UnsupportedAsset();
```

Level Money: Fixed in commit 5afc1851.

Cantina: Verified, the recommended fix was implemented.

3.2.4 Missing setter for `cooldownDuration` in `LevelMinting`

Severity: Medium Risk

Context: LevelMinting.sol#L97, LevelMinting.sol#L133-L143

Description: In the `LevelMinting` contract, `cooldownDuration` is set to 7 days in the constructor. However, the contract does not have a function to update the value of `cooldownDuration`.

Therefore, the protocol admin will not be able to update the withdrawal period that users need to go through or disable it. This is problematic as users the protocol's withdrawal period cannot be adjusted to match the withdrawal period of vaults in `LevelReserveManager`. Additionally, `redeem()` can never be called.

Recommendation: Add a permissioned function to update `cooldownDuration`.

Level Money: Fixed in commit 5afc1851.

Cantina: `setCooldownDuration()` was added to set `cooldownDuration`, but it should check that `newDuration` does not exceed `MAX_COOLDOWN_DURATION`.

Level Money: Fixed in commit 628bf291.

Cantina: Verified, the check was added.

3.2.5 Stablecoins residing on `LevelMinting` contract can be arbitraged if instant redemption is enabled

Severity: Medium Risk

Context: LevelMinting.sol#L23

Description: The `LevelMinting` supports multiple collateral assets (e.g., USDC, USDT, DAI).

It is possible that the `LevelMinting` contract will hold some USDC, USDT, and DAI at the same time. One lvlUSD can always be redeemed for exactly one USDC, USDT, or DAI (1:1 ratio). If the cooldown is disabled, it is possible for users to deposit and redeem within a single transaction. As a result, if the price of a stablecoin deviates from one dollar, MEV/arbitrager can take advantage of the `LevelMinting` contract to swap lower-value collateral with higher-value collateral (deposit + redeem). Unlike a swap in typical pools (e.g., Uniswap Pool), performing such a swap in `LevelMinting` contract does not incur any fee for the arbitrager.

In the end, the `LevelMinting` contract will left with a stablecoin of lower value as a higher-value one will be redeemed by MEV/arbitrager. If there is one of the stablecoins depeg (e.g., events similar to terraUSD crash), the impact will be greater, and it is possible that all the stablecoin residing on the `LevelMinting` contract will be swapped to the worthless stablecoin within a short period of time.

Recommendation: Consider avoiding enabling or having the instant redemption feature, as this would effectively turn the `LevelMinting` contract into a pool offering "fee"-less swaps, which MEV/arbitrageurs could exploit due to the nature of lvlUSD being pegged to stablecoin (e.g., USDC, USDT, DAI) in a 1:1 ratio.

Level Money: Fixed in commit 4437b47b.

Cantina: Verified, the amount of lvlUSD or collateral that is minted/redeemed is adjusted based on the collateral's price returned by Chainlink oracles:

- If a collateral asset has a negative depeg, the amount of lvlUSD minted will be discounted.
- If a collateral asset has a positive depeg, the amount of collateral redeemed will be discounted.

This prevents arbitrage from occurring.

3.2.6 Deposit and redeem functions can be DOSed by exceeding the deposit and redemption cap

Severity: Medium Risk

Context: LevelMinting.sol#L105, LevelMinting.sol#L113

Description: There is a maximum number of stablecoins that can be minted per block. In the current design, it is possible that the cooldown for redemptions is turned off. In this case, users can immediately execute the `redeem()` to withdraw assets.

Assuming that the deposit cap and redemption cap are both set to 10000. In an L2 environment where the gas fee is cheap, malicious users can DOS the deposit and redemption by atomically depositing and redeeming 10000 within a single transaction every block, thus resulting in the cap to be reached. As such, innocent users attempting to deposit or redeem will revert.

Recommendation: Consider imposing a fee on deposits and/or withdrawals to deter such an attack.

Level Money: Acknowledged.

Cantina: Acknowledged.

3.3 Low Risk

3.3.1 Denylisted user can still send `lvlUSD` via `LevelMinting.mint`

Severity: Low Risk

Context: LevelMinting.sol#L210, lvlUSD.sol#L71

Description: `lvlUSD` applies a deny list in the `_beforeTokenTransfer()` where, next to `to` and `from`, `denyList(msg.sender)` is checked. When minting, `LevelMinting` would be inserted in `denyList(msg.sender)` since it's calling `lvlUSD.mint` for the user. As the denylist is implemented due to regulations, this effectively can be seen as a transfer of lvlUSD by a denylisted user to any contract.

Recommendation: Add the following to `LevelMinting._mint`:

```
+ require !(lvlUSD.denylisted(msg.sender));
```

Level Money: Fixed in commit 5afc1851.

Cantina: Verified, the recommended check was added.

3.3.2 Missing approval to Karak's vault in `LevelReserveManager.startRedeemFromKarak()`

Severity: Low Risk

Context: LevelReserveManager.sol#L124-L127, Vault.sol#L146

Description: The `LevelReserveManager.startRedeemFromKarak()` function directly calls `Vault.startRedeem()` in the specified Karak vault:

```
withdrawalKey = IKarakVault.IVault(vault).startRedeem(
    shares,
    address(this)
);
```

However, `Vault.startRedeem()` calls `this.transferFrom()` to transfer shares from the caller to the vault:

```
this.transferFrom(msg.sender, address(this), shares);
```

As such, to call `startRedeem()`, this contract needs to grant allowance to the vault to transfer its shares. Alternatively, the allowance can be granted in a separate call to `approveSpender()`.

Recommendation: Consider adding an approval of `shares` to `vault` before calling `startRedeem()`:

```
+ IERC20(vault).forceApprove(vault, shares);
  withdrawalKey = IKarakVault.IVault(vault).startRedeem(
      shares,
      address(this)
  );
```

Level Money: Fixed in commit 5afc185.

Cantina: The fix for this is missing. It was implemented in commit 5afc1851, but seems to have been accidentally overwritten afterwards in commit a7f5af0c.

Level Money: Fixed in commit 628bf291.

Cantina: Verified, the recommended fix was implemented.

3.3.3 Missing input validation for `LevelMinting.initiateRedeem()`

Severity: Low Risk

Context: LevelMinting.sol#L277-L295

Description: The `LevelMinting.initiateRedeem()` function only checks that `order.order_type` is `OrderType.REDEEM`. There are no other checks for the fields in the `order` struct. As such, users can initiate redemption with:

1. `order.collateral_asset` as an asset not in `supportedAssets`. This allows them to initiate redemption for a collateral that will be added in the future. Or they might accidentally initiate a redemption for a collateral asset that never gets added.
2. `order.lvlusd_amount` not equal to `order.collateral_amount`.

In both cases, the initiated redemption cannot be completed.

Recommendation: Consider adding the following checks:

1. Add a `supportedAssets.contains(collateral_asset)` check to `initiateRedeem()`.
2. Move the `checkCollateralAndlvlUSDAmountEquality()` check from `completeRedeem()` to `initiateRedeem()`.

This prevents users from accidentally initiating a redemption that cannot be completed.

Level Money: Fixed in commit 5afc1851.

Cantina: The recommended checks were added to `initiateRedeem()`. However, since assets that that were removed from `supportedAssets` should still be redeemable, `initiateRedeem()` should now check if `collateral_assets` is in `_redeemableAssets` instead.

Level Money: Fixed in commit 532771c9.

Cantina: Verified, assets removed from `supportedAssets` will now still be withdrawable through `initiateRedeem()`. To allow the protocol admin to make an asset non-withdrawable, `removeRedeemableAssets()` was added to remove assets from `_redeemableAssets`.

3.3.4 Changing `cooldownDuration` in `LevelMinting` does not affect ongoing pending redemptions

Severity: Low Risk

Context: LevelMinting.sol#L282-L285, LevelMinting.sol#L302-L304

Description: When users call `LevelMinting.initiateRedeem()` to initiate the redemption process, the timestamp at which the redemption can be completed (i.e. `cooldownEnd`) is stored as `block.timestamp + cooldownDuration`:

```
UserCooldown memory newCooldown = UserCooldown({
    cooldownEnd: uint104(block.timestamp + cooldownDuration),
    order: order
});
```

After `cooldownEnd` has passed, users can then call `completeRedeem()` to process their withdrawal:

```
if (
    block.timestamp >= userCooldown.cooldownEnd || cooldownDuration == 0
) {
```

However, if `cooldownDuration` is decreased after a user initiates a redemption, the user will still need to wait for the outdated cooldown period to end.

For example:

- `cooldownDuration = 7 days`.
- User calls `initiateRedeem()` and `cooldownEnd` is set to T+7 days.
- Admin decreases `cooldownDuration` to 3.5 days.
- User can only call `completeRedeem()` at T+7 days, whereas other users who called `initiateRedeem()` after `cooldownDuration` was decreased can withdraw in 3.5 days.

Likewise, if `cooldownDuration` is increased, users will still be able to withdraw in 7 days.

Recommendation: Consider storing the timestamp at which `initiateRedeem()` was called instead:

```
  UserCooldown memory newCooldown = UserCooldown({
-     cooldownEnd: uint104(block.timestamp + cooldownDuration),
+     cooldownStart: uint104(block.timestamp),
      order: order
  });
```

In `completeRedeem()`:

```
  if (
-     block.timestamp >= userCooldown.cooldownEnd || cooldownDuration == 0
+     block.timestamp >= userCooldown.cooldownStart + cooldownDuration || cooldownDuration == 0
  ) {
```

Level Money: Fixed in commit 5afc1851.

Cantina: Verified, the recommended fix was implemented.

3.3.5 Collateral could be redeemed without locking user's lvlUSD

Severity: Low Risk

Context: LevelMinting.sol#L277

Description: It was observed that allowing users to initiate redemption (or start the 7-day cooldown) without locking their collateral would not be beneficial for the protocol.

Assume Bob wants the ability to withdraw at any time without waiting for the cooldown period. To achieve this, Bob will always execute the `initiateRedeem` function immediately after depositing, allowing the cooldown to "age" in the background. Bob has no intention of completing the withdrawal after the 7-day period but wants the option to withdraw instantly once Day 7 passes. Meanwhile, he continues to utilize the minted lvlUSD (e.g., staking it on another protocol to earn yield) for the next 14 days before deciding to complete the withdrawal.

In this scenario:

- Bob earns yields in X protocol for 14 days.
- Protocol picks up Bob's `RedeemInitiated event` on Day 0 and its bot initiates a withdrawal to restaking protocol. Assuming for this specific restaking protocol, it stops earning rewards when the staked collateral enters the withdrawal queue. Thus, from Day 0 to Day 7, the protocol earns 0 rewards. Assuming that afterward, the protocol's algorithm stakes the unused collateral back to the restaking protocol, and earn a total rewards of 7 (1 reward/day) from Day 8 to Day 14.

Assuming another scenario where the protocol requires users to lock their collateral upon initiating the redemption. In this case, since Bob needs to utilize the lvlUSD from Day 0 to Day 14, he cannot afford to initiate the redemption and have his lvlUSD locked. Therefore, he would only initiate the redemption on Day 15, after he's finished earning yield on protocol X. As a result, from Day 0 to Day 14, the protocol would earn a total of 14 rewards (1 reward per day).

Comparing the first and second scenarios, the first scenario would result in less revenue for the protocol.

Recommendation: Consider updating the `initiateRedeem()` function to require users to transfer their lvlUSD to this contract, and their lvlUSD is only burnt when `completeRedeem()` is called.

If this approach is adopted, ensure that the contract includes logic to handle cases where the cooldown is disabled while pending redemptions are ongoing, since the lvlUSD is no longer directly burned from the user.

Level Money: Fixed in commit 5afc185.

Cantina: The recommended approach for locking and subsequently burning lvlUSD was implemented. However, the case where the cooldown is disabled while pending redemptions are ongoing isn't handled. Consider the following scenario:

- cooldownDuration is set to 7 days.
- User calls `initiateRedeem()`, which transfers lvlUSD into the `LevelMinting` contract.
- Admin sets `cooldownDuration` to 0 days.
- User can no longer call `completeRedeem()` due to the `ensureCooldownOn` modifier.
- User cannot call `redeem()` as his lvlUSD is locked in the `LevelMinting` contract.

Whenever the admin disables the cooldown period, all lvlUSD locked for pending redemptions instantly become stuck. A possible fix for this would be to remove the `ensureCooldownOn` modifier from `completeRedeem()`, which allows pending redemptions to be completed even when `cooldownDuration = 0`.

Level Money: Fixed in commit 628bf291.

Cantina: Verified, the `ensureCooldownOn` modifier was removed.

3.3.6 Ineffective segregation of control in `LevelReserveManager`

Severity: Low Risk

Context: LevelReserveManager.sol#L53, LevelReserveManager.sol#L64

Description: The purpose of the allowlist is to restrict the addresses to which reserve funds can be transferred by the admins. This ensures that reserve funds cannot be transferred to unauthorized addresses or wallets.

However, the existing allowlist does not introduce segregation of controls to the system. A malicious admin could grant themselves the `ALLOWLIST_ROLE` role via the `grantRole` function that they have access to and proceed to add the unauthorized address/wallet via the `addToAllowList` function. Once that has been done, malicious admins could proceed to transfer the reserve funds to the unauthorized address/wallet.

- `SingleAdminAccessControl.sol`:

```
43:     /// @notice grant a role
44:     /// @notice can only be executed by the current single admin
45:     /// @notice admin role cannot be granted externally
46:     /// @param role bytes32
47:     /// @param account address
48:     function grantRole(
49:         bytes32 role,
50:         address account
51:     ) public override onlyRole(DEFAULT_ADMIN_ROLE) notAdmin(role) {
52:         _grantRole(role, account);
53:     }
```

Recommendation: If segregation of controls is intended, the right to transfer ETH via `transferEth` and ERC20 via `transferERC20` from the Reserve Manager should be granted to Party A, while the right to update the allowlist should be granted to Party B. Party A and Party B should be different addresses and should not have the right to execute the `grantRole` function.

Level Money: Acknowledged, we are okay with this as our admin is a multisig.

Cantina: Acknowledged.

3.4 Informational

3.4.1 Unused Code

Severity: Informational

Context: *(See each case below)*

Description:

- **Instance 1:** `DelegatedSigner` feature exists in the Etherna contract but has been removed in the `LevelMinting` contract. Thus, `setDelegatedSigner` and `removeDelegatedSigner` functions can be removed since they are no longer necessary:
  - LevelMinting.sol#L348
  - LevelMinting.sol#L354
- **Instance 2:** Order's nonces and `_deduplicateOrder()` are no longer needed since signature verification was removed:
  - LevelMinting.sol#L200
  - LevelMinting.sol#L258

Instance 3: `LevelMinting` does not perform any check against the order's signature, unlike Etherna. Thus, the following codes and functions can be removed: - `hashOrder` function - `getDomainSeparator()` function. - `_computeDomainSeparator` function. - `encodeOrder()` function. - `taker_order_hash` variable in the `verifyOrder()` function.

Recommendation: Consider removing the unused from the codebase.

- **Instance 2:** If nonces are removed, `completeRedeem()` needs to be changed as such too:

```
- userCooldown.cooldownEnd = 0;
- cooldowns[msg.sender][token] = userCooldown;
+ delete cooldowns[msg.sender][token];
```

Level Money: Fixed in commit 5afc1851.

Cantina: Verified, the code mentioned above has been removed.

3.4.2 Karak vault shares can be transferred out of the `LevelReserveManager` contract

Severity: Informational

Context: LevelReserveManager.sol#L48-L58

Description: In `LevelReserveManager`, the protocol admin can call `transferERC20()` to transfer out any ERC20 token from the contract. This is meant for the admin to transfer collateral out of the contract.

```
function transferERC20(
    address tokenAddress,
    address tokenReceiver,
    uint256 tokenAmount
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (allowlist[tokenReceiver]) {
        IERC20(tokenAddress).safeTransfer(tokenReceiver, tokenAmount);
    } else {
        revert InvalidRecipient();
    }
}
```

However, note that Karak's vaults are ERC-4626 vaults, so any shares held by this contract can be transferred out using this function as well.

Recommendation: Consider adding a whitelist of collateral assets that can be transferred using `transferERC20()` and `approveSpender()`. Alternatively, consider documenting that the protocol admin has the ability to transfer Karak vault shares out of the contract.

Level Money: Acknowledged, we are okay with this as the protocol admin is a multisig and fully trusted.

Cantina: Acknowledged.

3.4.3 Minor improvements

Severity: Informational

Context: LevelMinting.sol#L227-L239, LevelMinting.sol#L224-L225, LevelMinting.sol#L302-L304, LevelReserveManager.sol#L21, LevelReserveManager.sol#L29, LevelMinting.sol#L467, LevelMinting.sol#L306, LevelMinting.sol#L257, LevelReserveManager.sol#L120-L129

Description/Recommendation:

1. LevelMinting.sol#L227-L239: In `checkCollateralAndlvlUSDAmountEquality()`, `assert` is used to perform checks. Using `require` instead of `assert` is recommended as `assert` burns all remaining gas from the caller if it fails.
2. LevelMinting.sol#L224-L225: Consider using the `IERC20` interface instead:

```
- uint8 collateral_asset_decimals = ERC20(order.collateral_asset)
+ uint8 collateral_asset_decimals = IERC20(order.collateral_asset)
    .decimals();
```

3. LevelMinting.sol#L302-L304: The second condition `cooldownDuration == 0` is unnecessary as this condition will never be true within the function due to the `ensureCooldownOn` modifier. Consider removing it:

```
if (
-     block.timestamp >= userCooldown.cooldownEnd || cooldownDuration == 0
+     block.timestamp >= userCooldown.cooldownEnd
) {
```

4. LevelReserveManager.sol#L21: This line can be removed as all ERC20 operations are done with the `IERC20` interface.
5. LevelReserveManager.sol#L29: Unused state variable can be removed.
6. LevelMinting.sol#L467: `verifyOrder()` return action can be removed as it's not used anywhere.
7. LevelMinting.sol#L306: Assigning `userCooldown.cooldownEnd = 0` will actually result in `block.timestamp >= userCooldown.cooldownEnd` always continuing. The factor that blocks being able to call this again is in `verifyNonce()`, which makes `cooldowns[msg.sender][token] = userCooldown` redundant.
8. LevelMinting.sol#L257: `initiateRedeem()`, `redeem()` and `_redeem()` all implement the `orderType == OrderType.REDEEM` check. This check van be removed in `_redeem()`.
9. LevelReserveManager.sol#L120-L129: Karak's withdrawal waiting delay is currently 9 days. The max cooldown period is 7 days in `LevelMinting.sol`. If the LevelReserveManager doesn't have enough tokens to cover a withdrawal request (within the 7 days period set) and instead choose to redeem from Karak, users will have to wait two more days and their initial request might fail.

Level Money: Acknowledged issue 9. The remaining issues have been fixed in commits 5afc1851 and 628bf291.

Cantina: Verified.

3.4.4 Pending redemptions in `LevelMinting` can exceed the protocol's balance

Severity: Informational

Context: LevelMinting.sol#L282-L287, LevelMinting.sol#L297-L317

Description: In `LevelMinting`, users can call `initiateRedeem()` to initiate a redemption for any amount of collateral asset:

```
UserCooldown memory newCooldown = UserCooldown({
    cooldownEnd: uint104(block.timestamp + cooldownDuration),
    order: order
});

cooldowns[msg.sender][order.collateral_asset] = newCooldown;
```

After the 7-day cooldown period, users subsequently call `completeRedeem()` to withdraw their collateral assets. Since the contract does not track the amount of collateral assets in pending redemptions, it is possible for the total amount of collateral asset in pending redemptions to be more than the actual amount of collateral asset owned by the protocol.

For example:

- USDC and USDT are both in `supportedAsset`.
- Alice deposits 100 USDC.
- Bob deposits 100 USDT.
- Both Alice and Bob call `initiateRedeem()` for 100 USDT.
- After 7 days, Alice calls `completeRedeem()` first and withdraw 100 USDT.
- When Bob calls `completeRedeem()`, the protocol has 0 USDT so his redemption can't be completed.

Now, for Bob to exit the protocol, he is forced to either go through another 7-day cooldown period to withdraw USDC or wait for the protocol to have more USDT collateral.

Recommendation: Consider tracking the amount of assets in all pending redemptions for each supported collateral and ensuring that the amount in all pending redemptions does not exceed the protocol's balance. Otherwise, consider documenting this limitation.

Level Money: Acknowledged.

Cantina: Acknowledged.

3.4.5 Impact of different block time on `maxMintPerBlock` and `maxRedeemPerBlock`

Severity: Informational

Context: LevelMinting.sol#L105, LevelMinting.sol#L113

Description: Block time in Ethereum and other L2 chains is different. In Ethereum, the block time is 12 seconds, while other L2 chains have a different block time.

Assume the protocol intends to restrict minting to only 1 million tokens per minute. In this case, one would set (200,000 tokens per block) on Ethereum. However, on Optimism where block time is 2 seconds, the rate would need to be much smaller (33333 tokens per block).

Recommendation: Take note of the different block time on different blockchains when configuring the `maxMintPerBlock` and `maxRedeemPerBlock`

Level Money: Acknowledged.

Cantina: Acknowledged.