3A DAO Audit Report

Prepared by 0xMilenov

Contents

1 About 0xMilenov **2**

2 Protocol Summary **2**

3 Risk Classification **2**
3.1 Impact 2
3.2 Likelihood 2
3.3 Action required for severity levels 3

4 Executive Summary **3**

5 Findings **4**
5.1 High Risk 4
5.2 Medium Risk 6
5.3 Low Risk 8
5.4 Informational 12
5.5 Gas Optimization 15

1 About 0xMilenov

Independent Security Researcher

As a software engineer based in Bulgaria, 0xMilenov is a passionate expert in blockchain security, with a keen focus on **Web3, smart contracts, and vulnerability detection.**

Collaborating with renowned figures, he's contributed significantly to Web3 security projects and developed impactful tools.

0xMilenov's blend of technical expertise, community building, and proactive security research establish him as a dynamic and influential figure in the blockchain security landscape.

Feel free to reach out on X or 0xMilenov

2 Protocol Summary

3A DAO

3A is a decentralized lending protocol offering **efficient leverage with no recurring interest for both crypto assets and on-chain Real World Assets**, ensuring enterprise-grade security.

It is non-custodial and permissionless, allowing users to leverage their long positions, which is particularly attractive for yield-bearing assets such as **liquid-staked tokens, LP tokens, tokenized bonds, and treasuries.**

Loans are paid in EURO3, pegged to EUR, while A3A serves as the governance token of the protocol.

Learn more at `https://3adao.org/`

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of {X} according to the specific commit. Any modifications to the code will require a new security review.

3 Risk Classification

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

3.1 Impact

- 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.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

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

4 Executive Summary

Overview

Project	3A DAO
Repository	3a-borrowing-protocol
Commit	a19ad8f98063...
Date	February 2024

Issues Found

Severity	Count
Critical Risk	0
High Risk	1
Medium Risk	4
Low Risk	1
Informational	1
Gas Optimizations	1
Total Issues	**8**

Summary of Findings

Title	Status
[H-1] VaultFactory is suspicious of a reorg attack	Acknowledged
[M-1] Decimals > 18 will cause a revert for normalizedCollateralAmount	Resolved
[M-2] Missing zero-value check for Chainlink price feed	Resolved
[M-3] Some tokens dont support approve `type(uint256).max` amount	Resolved
[M-4] Prefer `call()` over `transfer()` for address payable in `VaultFactory`	Resolved
[L-1] Low Risk Issues	Recommended
[I-1] Informational Issues	Recommended
[G-1] Gas Optimizations	Recommended

5 Findings

5.1 High Risk

[H-1] VaultFactory is suspicious of a reorg attack

Context: VaultFactory.sol

Impact In the `VaultFactory` contract poses a significant risk of funds theft. It allows malicious actors to exploit reorg attacks, particularly on networks like Polygon, leading to unauthorized control of user funds.

Description The `createVault` creates a new vault and returns its address. The address of the newly created vault is determined by the deployVault function called on the IVaultDeployer interface.

This process uses the create opcode, which generates the contract address based on the contract creator's address and nonce.

```
function createVault(string memory _name) public returns (address) {
  address _msgSender = _msgSender();
  address _vaultAddress = IVaultDeployer(vaultDeployer).deployVault(address(this), _msgSender, _name);
  _addVault(_msgSender, _vaultAddress);
  emit NewVault(_vaultAddress, _name, _msgSender);

  return _vaultAddress;
}
```

The predictable nature of this address generation method (based on nonce) poses a risk, particularly in the event of blockchain reorganizations (reorgs).

During a reorg, transactions (including contract creations) might be rolled back, resetting the nonce. An attacker observing the network could exploit this by deploying their own contract with the same address during a reorg, especially in networks susceptible to reorgs, and intercept funds or interactions meant for the original contract.

Additionally, the `addCollateralNative` adds native-wrapped collateral to the specific vault. This function requires the vault to be registered and the collateral type to be supported.

However, if the vault's address is predicted and controlled by an attacker (due to the above vulnerability), funds added as collateral could be compromised.

```
function addCollateralNative(address _vault) external payable {
  require(containsVault(_vault), "vault-not-found");
  require(_isCollateralSupported(nativeWrapped), "collateral-not-supported");
  uint256 _amount = msg.value;

  collateral[nativeWrapped] += _amount;

  require(collateral[nativeWrapped] <= collateralCap[nativeWrapped], "collateral-cap-reached");

  IWETH(nativeWrapped).deposit{value: _amount}();
  IERC20(nativeWrapped).safeTransferFrom(address(this), _vault, _amount);

  Vault(_vault).addCollateral(nativeWrapped, _amount);
}
```

`https://polygonscan.com/blocks_forked`

Here you may be convinced that the Polygon has in practice subject to reorgs. Even more, the reorg on the picture is 1.5 minutes long. So, it is quite enough to create the vault and transfer funds to that address, especially when someone uses a script, and not doing it by hand.

Optimistic rollups (Optimism/Arbitrum) are also suspect to reorgs since if someone finds a fraud the blocks will be reverted, even though the user receives a confirmation and already created a vault.

Attack scenario Imagine that Alice deploys a vault, and then sends funds to it. Bob sees that the network block reorg happens and calls `createVault`. Thus, it creates vault with an address to which Alice sends funds. Then Alices' transactions are executed and Alice transfers funds to Bob's controlled vault.

Recommendation To mitigate this risk, it is recommended to use the `create2` opcode for deploying new vault contracts. The `create2` opcode allows for a more unpredictable and safer contract address generation by including a user-provided salt value along with the deployer's address and nonce.

This change would make it significantly more difficult for an attacker to predict or replicate the address of a contract being deployed, thereby securing the contract deployment process against such attack vectors.

5.2 Medium Risk

[M-1] Decimals > 18 will cause a revert for normalizedCollateralAmount

Context: AuctionManager.sol

Impact The protocol allows tokens with any number of decimals to be whitelisted, this could severely compromise auction integrity and financial security.

Description "3A DAO relies on these underlying factors to assess a protocol's creditworthiness and decides if a token can get whitelisted as collateral.This risk model is going to be the backbone of 3A's long-term success"

We see that the whitelisting criteria missing a decimal limit for tokens!

The `newAuction` function inside `AuctionManager` is vulnerable when normalizing collateral amounts. This issue arises if a collateral token with more than 18 decimals is used, **leading to a revert in the normalization calculation.**

```
uint256 _normalizedCollateralAmount = _collateralAmount * (10 ** (18 -
↪    _priceFeed.decimals(address(collateralToken))));
```

Proofs - Secureum Mind Map & Consensys

Recommendation Implement a check to ensure collateral tokens do not exceed 18 decimals:

```
require(_priceFeed.decimals(address(collateralToken)) <= 18, "Token decimals exceed limit");
```

[M-2] Missing zero-value check for Chainlink price feed

Context: TokenToPriceFeed.sol, ChainlinkPriceFeed.sol

Impact The lack of a zero-value validation in `tokenPrice` for the Chainlink oracle price feed can lead to significant financial risks. If the oracle returns a zero price due to a malfunction, attack, or bug, the contract would incorrectly process this as a valid price.

Description The TokenToPriceFeed's `tokenPrice()` functions, which overrides `IPriceFeed`, retrieve prices from a Chainlink oracle **but do not check if the returned price is zero.** This oversight can cause critical issues in financial applications, like incorrect liquidations or reward miscalculations, especially if multiple systems rely on this data.

```
function tokenPrice(address _token) public view override returns (uint256) {
  return IPriceFeed(tokens[_token].priceFeed).price();
}
```

Recommendation To address this issue, add a require(_price > 0, "Invalid price data") check to ensure non-zero price data.

[M-3] Some tokens dont support approve `type(uint256).max` amount

Context: LiquidationRouter.sol, AuctionManager.sol

Impact The issue can cause transaction failures for tokens like UNI or COMP that do not support unlimited approval amounts, potentially impacting the protocol's liquidity operations, on `Ethereum Mainnet`.

Description Some tokens doesn't support approve spender type(uint256).max amount. So the `addSeizedCollateral` will not work for some tokens like UNI or COMP and will revert when approve type(uint256).max amount. It has many examples like this one

```
function addSeizedCollateral(address _collateral, uint256 _amount) external onlyVault {
    IERC20(_collateral).safeTransferFrom(msg.sender, address(this), _amount);

    IERC20(_collateral).safeApprove(stabilityPool, 0);
    IERC20(_collateral).safeApprove(stabilityPool, type(uint256).max);

    IERC20(_collateral).safeApprove(auctionManager, 0);
    IERC20(_collateral).safeApprove(auctionManager, type(uint256).max);

    collateralSet.add(_collateral);
    collateral[_collateral] += _amount;
    emit SeizedCollateralAdded(_collateral, vaultFactory, msg.sender, _amount);
}
```

Recommendation Consider approving the exact amount that needs to be transferred or adding an external function that allows the revocation of approvals.

[M-4] Prefer `call()` over `transfer()` for address payable in `VaultFactory`

Context: VaultFactory.sol

Description Utilizing the outdated `transfer()` function on an address can lead to transaction failures due to the 2300 gas stipend.

The use of the deprecated `transfer()` function for an address will inevitably make the transaction fail when:

The claimer smart contract does not implement a payable function. The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit. The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.

Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.

‘Consensys: Don't use '.transfer' or '.send'

It have many confirmed issues like this one, exmaple

```
    payable(_to).transfer(_amount);
```

Recommendation Opting for the `call()` function is recommended in such scenarios to circumvent this issue.

5.3 Low Risk

[L-1] Low Risk Issues

	Issue	Instances	Gas
[L-01]	Arrays can grow in size without a way to shrink them	2	
[L-02]	The `decimals()` function isn't included in the ERC-20 specification	1	
[L-03]	`safeApprove()` is deprecated	3	
[L-04]	Solidity 0.8.20's reliance on `PUSH0` could pose compatibility issues on alternate chains	21	
[L-05]	Use `Ownable2Step.acceptOwnership()` instead of `Ownable.transferOwnership`	7	
[L-06]	The file permits a version of Solidity that is vulnerable to an assembly optimizer bug	21	
[L-07]	Note that `symbol()` is not included in the ERC-20 standard	1	
[L-08]	Known vulnerabilities in current `@openzeppelin/contracts` version	1	

[L-01] Arrays can grow in size without a way to shrink them

As these arrays cannot shrink, if the array has a maximum size, it won't be possible to change its elements once it reaches that size. Otherwise, it can grow indefinitely in size, which can increase the likelihood of out-of-gas errors.

```
File: contracts/AuctionManager.sol

}195:   auctions.push(
```

195

```
File: contracts/StabilityPool.sol

}444:   epochToScaleToTokenToSum[currentEpochCached][currentScaleCached].push() = tokenToS;
```

444

[L-02] The decimals() function isn't included in the ERC-20 specification

While the decimals() function isn't originally included in the ERC-20 standard, it was introduced later as an optional add-on. Given this, not all valid ERC20 tokens implement this interface.

Therefore, indiscriminately casting all tokens to this interface and subsequently invoking this function can be risky

```
File: contracts/oracles/ChainlinkPriceFeed.sol

}29:     uint8 decimals = oracle.decimals();
```

29

[L-03] `safeApprove()` is deprecated

The function has been marked as deprecated and is recommended to be replaced with safeIncreaseAllowance() and safeDecreaseAllowance().

If you're solely setting the allowance to a value signifying infinite, you can utilize safeIncreaseAllowance() as a substitute.

While the function might operate correctly now, any discovered flaws in this OpenZeppelin version, coupled with a mandatory upgrade to a version lacking this function, could result in unforeseen hold-ups in adapting and evaluating alternative contracts.

```
File: contracts/AuctionManager.sol

}261:      collateralToken.safeApprove(address(_lastResortLiquidation), 0);
}262:      collateralToken.safeApprove(address(_lastResortLiquidation), type(uint256).max);
```

261, 262

```
File: contracts/LiquidationRouter.sol

}150:      IERC20(_collateral).safeApprove(stabilityPool, 0);
}151:      IERC20(_collateral).safeApprove(stabilityPool, type(uint256).max);
}153:      IERC20(_collateral).safeApprove(auctionManager, 0);
}154:      IERC20(_collateral).safeApprove(auctionManager, type(uint256).max);
```

150, 151, 153, 154

```
File: contracts/Vault.sol

}417:      IERC20(_collateral).safeApprove(IVaultFactory(factory).liquidationRouter(), 0);
}418:      IERC20(_collateral).safeApprove(IVaultFactory(factory).liquidationRouter(), type(uint256).max);
```

417, 418

[L-04] Solidity 0.8.20's reliance on PUSH0 could pose compatibility issues on alternate chains

Using Solidity 0.8.20's new PUSH0 opcode can cause deployment issues on some L2s due to incompatibility.

To work around this issue, use an earlier version

```
File: ~All files

}2: pragma solidity ^0.8.4;
```

[L-05] Use Ownable2Step.acceptOwnership() instead of Ownable.transferOwnership

Better use Ownable2Step.acceptOwnership() because it is more secure due to 2-stage ownership transfer.

```
File: contracts/interfaces/ILiquidationRouter.sol

}23:     function transferOwnership(address newOwner) external;
```

23

```
File: contracts/interfaces/IMintableToken.sol

}5: import "./IOwnable.sol";
```

5

```
File: contracts/interfaces/IMintableTokenOwner.sol

}5: import "./IOwnable.sol";
```

5

```
File: contracts/interfaces/IOwnable.sol

}14:    function transferOwnership(address newOwner) external;
```

14

```
File: contracts/interfaces/ITokenPriceFeed.sol

}4: import "./IOwnable.sol";
```

4

```
File: contracts/MintableTokenOwner.sol

}28:         token.transferOwnership(_newOwner);
```

28

```
File: contracts/TokenToPriceFeed.sol

}120:     function transferOwnership(address _newOwner) public override(Ownable, IOwnable) {
}121:         Ownable.transferOwnership(_newOwner);
```

120, 121

[L-06] The file permits a version of Solidity that is vulnerable to an assembly optimizer bug

In Solidity versions 0.8.13 and 0.8.14, a known optimizer bug can cause uninitialized memory due to the optimization of mstore operations when a variable's usage and storage occur in separate assembly blocks.

While the current codebase doesn't exhibit this pattern, it utilizes mstore within assembly blocks, presenting a potential risk for future alterations.

It is advisable to avoid using the affected Solidity versions whenever possible

```
File: ~All files

}2: pragma solidity ^0.8.4;
```

[L-07] Note that symbol() is not included in the ERC-20 standard

The symbol() function isn't part of the ERC-20 standard but was introduced as an optional extension.

Consequently, not all valid ERC20 tokens support this interface.

Blindly casting tokens to this interface and calling the function can be unsafe.

```
File: contracts/TokenToPriceFeed.sol

}107:     erc20.symbol(),
```

107

[L-08] Known vulnerabilities in current @openzeppelin/contracts version

Given the presence of known vulnerabilities in the current @openzeppelin/contracts version, it is advisable to update to at least @openzeppelin/`contracts@5.0.1` to address these issues and enhance the contract's security

```
File: package.json

}14:     "@openzeppelin/contracts": "^4.9.0",
```

14

5.4 Informational

[I-1] Informational Issues

	Issue	Instances	Gas Savings
[N-01]	The risks of relying on `assert()` in solidity contracts	3	
[N-02]	Prefer `string.concat()` or `bytes.concat()` to `abi.encodePacked`	1	
[N-03]	Unemitted `Event` declarations	1	
[N-04]	Avoid hard-coding address values	6	
[N-05]	Statements like `require()` / `revert()` should have descriptive reason strings	1	
[N-06]	Recommendation to use `uint48` for time-centric variables	15	

[N-01] The risks of relying on `assert()` in solidity contracts

In Solidity, the assert() function is meant to handle conditions that should never be false, and if they are, it points to a bug within the contract.

Before Solidity 0.8.0, when assert() is triggered, it consumes all of the remaining gas in a transaction, which differs from the behavior of require() and revert() that return unspent gas.

Even in versions after 0.8.0, it's prudent to minimize the use of assert(). The Solidity documentation suggests that code should never reach a state where assert() is triggered. Instead, for better clarity and understanding of contract failures, developers are advised to utilize require() statements or custom error messages.

By doing so, they can provide a more transparent and user-friendly contract, which aids in identifying and rectifying issues more efficiently.

```
File: contracts/StabilityPool.sol

}416:      assert(_stableCoinLossPerUnitStaked <= DECIMAL_PRECISION);
}482:      assert(newP > 0);
}685:      assert(_debtToOffset <= _totalStableCoinDeposits);
```

416, 482, 685

[N-02] Prefer `string.concat()` or `bytes.concat()` to `abi.encodePacked`

With the release of Solidity version 0.8.4, the language introduced bytes.concat() as a clearer alternative to the abi.encodePacked(< bytes >,< bytes >) method for concatenating byte sequences.

Similarly, Solidity version 0.8.12 brought string.concat(), providing a direct means for string concatenation, sidestepping the abi.encodePacked(< str >,< str >) approach.

Incorporating these recent methods can improve code readability and align your contracts with modern Solidity practices.

```
File: contracts/utils/PoolAddress.sol

}35:       abi.encodePacked(
```

35

[N-03] Unemitted `Event` declarations

Certain events have been defined in the code but aren't ever triggered.

To enhance code clarity and cleanliness, consider eliminating these unused event definitions.

The instances mentioned below fall into this category.

```
File: contracts/A3AStaking.sol

}45:     event A3aTokenAddressSet(address _a3aTokenAddress);
```

45

[N-04] Avoid hard-coding address values

For greater flexibility across different network deployments, it's advisable to use immutable for declaring address values and assign them using constructor parameters. This approach ensures consistent code and eliminates the need for recompilation when address assignments change.

```
File: contracts/oracles/MockConvertedPriceFeed.sol

}11:     address public constant DAI = 0x8f3Cf7ad23Cd3CaDbD9735AFf958023239c6A063;
}12:     address public constant WETH = 0x7ceB23fD6bC0adD59E62ac25578270cFf1b9f619;
}13:     address public constant WMATIC = 0x0d500B1d8E8eF31E21C99d1Db9A6444d3ADf1270;
}14:     address public constant QNT = 0x36B77a184bE8ee56f5E81C56727B20647A42e28E;
}15:     address public constant PAXG = 0x553d3D295e0f695B9228246232eDF400ed3560B5;
}16:     address public constant USDC = 0x2791Bca1f2de4661ED88A30C99A7a9449Aa84174;
```

11, 12, 13, 14, 15, 16

[N-05] Statements like `require()` / `revert()` should have descriptive reason strings

```
File: contracts/utils/PoolAddress.sol

}30:       require(key.token0 < key.token1);
```

30

[N-06] Recommendation to use `uint48` for time-centric variables

For time-relevant variables, `uint32` might be restrictive as it ends in 2106, potentially posing challenges in the far future.

On the other hand, using excessively large types like `uint256` is unnecessary. Thus, `uint48` offers a balanced choice for such use cases.

```
File: contracts/oracles/ChainlinkPriceFeed.sol

}39:         (, int256 _price, , uint256 _timestamp, ) = oracle.latestRoundData();
```

39

```
File: contracts/A3AStaking.sol

}31:     uint256 public lastFeeOperationTime;
}70:     function setInitialLastFee(uint256 _timestamp) public onlyOwner {
```

31, 70

```
File: contracts/AuctionManager.sol

}40:        uint256 auctionStartTime;
}41:        uint256 auctionEndTime;
}61:        uint256 _auctionStartTime,
}62:        uint256 _auctionEndTime
}183:        uint256 _auctionStartTime = block.timestamp;
}184:        uint256 _auctionEndTime = _auctionStartTime + auctionDuration;
}230:     ) external view returns (uint256 _totalCollateralValue, uint256 _debtToAuctionAtCurrentTime) {
}289:          uint256 _debtToAuctionAtCurrentTime = _highestDebtToAuction -
```

40, 41, 61, 62, 183, 184, 230, 289

```
File: contracts/StabilityPool.sol

}60:     uint256 public latestA3ARewardTime;
}715:    uint256 newA3ARewardTime = block.timestamp;
}722:    uint256 timePassedInMinutes = (newA3ARewardTime - latestA3ARewardTime) /
↪   SECONDS_IN_ONE_MINUTE;
}723:    uint256 issuance = a3aPerMinute * timePassedInMinutes;
```

60, 715, 722, 723

5.5 Gas Optimization

[G-1] Gas Optimizations

	Issue	Instances	Gas Savings
[G-01]	Missing validation check in redeemReward	2	6
[G-02]	Optimize gas by using `!= 0` over `> 0` in `require()` for unsigned integers	9	54
[G-03]	a = a + b consumes less gas than a += b for state variables	7	112
[G-04]	Use assembly to validate msg.sender	7	84
[G-05]	Store array length outside loops for efficiency	3	9
[G-06]	Using assembly to check for zero can save gas	7	49
[G-07]	Revert strings are less gas-efficient than custom errors	14	700
[G-08]	Using a double if statement instead of a logical AND(&&)	1	30
[G-09]	Use shift right/left instead of division/multiplication if possible	2	44
[G-10]	Upgrade to a newer Solidity version.	21	N/A
[G-11]	Increments/decrements can be unchecked in for-loops	5	125
[G-12]	Separate require() conditions for gas efficiency	2	6

[G-01] Missing validation check in redeemReward

Context StabilityPool

Impact Users with a zero deposit are allowed to execute the function, potentially causing unnecessary gas expenditure.

Description

```
function redeemReward() external {
    Snapshots memory snapshots = depositSnapshots[msg.sender];
    uint256 contributorDeposit = deposits[msg.sender];

    uint256 compoundedDeposit = _getCompoundedDepositFromSnapshots(contributorDeposit, snapshots);
    _redeemReward();
    _updateDepositAndSnapshots(msg.sender, compoundedDeposit);
}
```

Recomendation. Add validation check.

```
function redeemReward() external {
    Snapshots memory snapshots = depositSnapshots[msg.sender];
    uint256 contributorDeposit = deposits[msg.sender];
+   require(contributorDeposit > 0, "deposit-is-0")

    uint256 compoundedDeposit = _getCompoundedDepositFromSnapshots(contributorDeposit, snapshots);
    _redeemReward();
    _updateDepositAndSnapshots(msg.sender, compoundedDeposit);
}
```

[G-02] Optimize gas by using `!= 0` over `> 0` in `require()` for unsigned integers

For contracts using Solidity versions up to `0.8.13`, there's a slight gas optimization to be had when using `!= 0` instead of `> 0` in `require()` checks on unsigned integers.

When the optimizer is enabled, the `!= 0` check consumes 6 gas units less. Although it might appear that `> 0` is more gas-efficient, this holds true only without the optimizer and outside of `require` statements.

For contracts where every gas unit matters, consider this optimization.

Reference: Tweet by @gzeon.

Ensure the Optimizer is enabled for maximal benefit.

```
File: ~All files

    require(_amount > 0, "amount-is-0");
```

```
+   require(_amount != 0, "amount-is-0");
```

[G-03] a = a + b consumes less gas than a += b for state variables, except for arrays and mappings

```
File: ~All files / Example with AuctionManager contract

}144:       _totalCollateralValue += _collateralValue;
}180:       _totalCollateralValue += _collateralValue;
```

```
+         _totalCollateralValue = _totalCollateralValue + _collateralValue;
+         _totalCollateralValue = _totalCollateralValue + _collateralValue;
```

144, 180

[G-04] Use assembly to validate msg.sender

We can use assembly to efficiently validate msg.sender with the least amount of opcodes necessary. For more details check the following report here

```
File: contracts/AuctionManager.sol

}156:      require(msg.sender == address(liquidationRouter), "not-allowed");
```

156

```
File: contracts/LiquidationRouter.sol

}84:       require(msg.sender == stabilityPool, "not-allowed");
}89:       require(msg.sender == lastResortLiquidation, "not-last-resort-liquidation");
```

84, 89

[G-05] Store array length outside loops for efficiency

When array length isn't cached, the Solidity compiler repeatedly reads it in every loop iteration.

For storage arrays, this means an added sload operation costing 100 extra gas for each subsequent iteration.

For memory arrays, it's an additional mload operation costing 3 extra gas post the first iteration.

```
File: contracts/LiquidationRouter.sol

}125:      for (uint256 i = 0; i <  collateralSet.length(); i++) {
```

125

```
File: contracts/StabilityPool.sol

}276:      for (uint128 i = 0; i <  _snapshots.tokenToSArray.length; i++) {
}285:      for (uint128 i = 0; i <  nextTokensToSum_cached.length; i++) {
}424:      for (uint128 i = 0; i <  currentTokenToSArray.length; i++) {
}552:      for (uint128 j = 0; j <  _snapshots.tokenToSArray.length; j++) {
}561:      for (uint128 j = 0; j <  nextTokensToSum_cached.length; j++) {
}758:      for (uint256 i = 0; i <  _depositorCollateralGains.length; i++) {
```

276, 285, 424, 552, 561, 758

```
File: contracts/Vault.sol

}177:      for (uint256 i = 0; i <  collateralSet.length(); i++) {
}385:      for (uint256 i = 0; i <  collateralSet.length(); i++) {
}408:      for (uint256 i = 0; i <  collateralSet.length(); i++) {
```

177, 385, 408

[G-06] Using assembly to check for zero can save gas

Using assembly to check for zero can save gas by allowing more direct access to the evm and reducing some of the overhead associated with high-level operations in solidity.

```
File: ~All files / Example with BONQMath.sol

}67:      if (_minutes == 0) {
}77:      if (n % 2 == 0) {
```

67, 77

[G-07] Revert strings are less gas-efficient than custom errors

From Solidity v0.8.4 onward, custom errors have been introduced.

These errors provide a savings of approximately 50 gas each instance they're triggered, as they circumvent the need to allocate and keep the revert string.

Omitting these strings also conserves gas during deployment.

Furthermore, custom errors are versatile, applicable both inside and outside contracts, including in interfaces and libraries.

As stated in the Solidity blog:

With the introduction of Solidity v0.8.4, a streamlined and gas-efficient method has been provided to elucidate to users the reasons behind an operation's failure through custom errors.

Prior to this, although strings could be used to detail failure reasons (e.g., revert('Insufficient funds.');), they were notably costlier, especially in terms of deployment, and incorporating dynamic information into them was challenging.

It's advisable to transition all revert strings to custom errors in your solution, especially focusing on those that appear multiple times.

```
File: ~All files

        require(...
```

[G-08] Using a double if statement instead of a logical AND(andand)

Using a double if statement instead of a logical AND (&&) can provide similar short-circuiting behavior whereas double if is slightly more gas efficient.

```
File: contracts/utils/linked-address-list.sol

}46:        if (_before && (_reference == address(0x0) || _reference == _list._first)) {
}53:        } else if (!_before && (_reference == address(0x0) || _reference == _list._last)) {
}91:        if (_element == _list._last && _element == _list._first) {
```

46, 53, 91

[G-09] Use shift right/left instead of division/multiplication if possible

While the `DIV` / `MUL` opcode uses 5 gas, the `SHR` / `SHL` opcode only uses 3 gas. Furthermore, beware that Solidity's division operation also includes a division-by-0 prevention which is bypassed using shifting. Eventually, overflow checks are never performed for shift operations as they are done for arithmetic operations. Instead, the result is always truncated, so the calculation can be unchecked in Solidity version `0.8+`

- Use `>> 1` instead of `/ 2`
- Use `>> 2` instead of `/ 4`
- Use `<< 3` instead of `* 8`
- ...
- Use `>> 5` instead of `/ 2^5 == / 32`
- Use `<< 6` instead of `* 2^6 == * 64`

TL;DR:

- Shifting left by N is like multiplying by 2^N (Each bits to the left is an increased power of 2)
- Shifting right by N is like dividing by 2^N (Each bits to the right is a decreased power of 2)

Saves around 2 gas + 20 for unchecked per instance

```
File: contracts/utils/BONQMath.sol

}47:        decProd = (prod_xy + (DECIMAL_PRECISION / 2)) / DECIMAL_PRECISION;
}79:        n = n / 2;
}84:        n = (n - 1) / 2;
```

47, 79, 84

```
File: contracts/utils/constants.sol

}21:    uint256 public constant PERCENT_05 = PERCENT / 2; // Represents 0.5%
```

21

[G-10] Upgrade to a newer Solidity version.

Using a more recent version of Solidity offers various benefits, including enhanced functionality, bug fixes, and potential gas savings. It ensures your smart contracts are more secure and compatible with the latest tools and libraries

```
File: ~All files

}2: pragma solidity ^0.8.4;
```

[G-11] Increments/decrements can be unchecked in for-loops

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

The change would be:

```
- for (uint256 i; i <  numIterations; i++) {
+ for (uint256 i; i <  numIterations;) {
 // ...
+   unchecked { ++i; }
}
```

These save around **25 gas saved** per instance.

The same can be applied with decrements (which should use `break` when `i == 0`).

The risk of overflow is non-existent for `uint256`.

```
File: ~All files / Example with AuctionManager.sol

}170:         for (uint256 i = 0; i <  _collateralsLength; i++) {
}259:         for (uint256 i = 0; i <  _collateralsLength; i++) {
}299:         for (uint256 i = 0; i <  _collateralsLength; i++) {
```

170, 259, 299

[G-12] Separate require() conditions for gas efficiency

It's beneficial to separate conditions in `require()` statements rather than using the `&&` operator.

As outlined in this discussed issue, while there might be a slightly higher deployment gas cost initially, the overall gas savings in runtime calls makes this approach more cost-effective in the long run.

Implementing this change can result in a saving of approximately *3 gas per instance*.

```
File: contracts/AuctionManager.sol

}232:       require(!_auction.auctionEnded && block.timestamp < = _auction.auctionEndTime, "auction-ended");
```

232

```
File: contracts/TokenToPriceFeed.sol

}92:        require(_mlr  >= 100 && _mlr < = _mcr, "MLR <  100 or MLR  > MCR");
```

92