Cyberscope

Audit Report

Aqua Bank

August 2025

Repository:

https://github.com/bluewhale-logan/aquabank-contract-staking

Commit: e023335965ad0934891087e8ce179ba28fe51e5f

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope’s smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/bluewhale-logan/aquabank-contract-staking
Commit	e023335965ad0934891087e8ce179ba28fe51e5f

Audit Updates

Initial Audit	10 Jul 2025 https://github.com/cyberscope-io/audits/blob/main/ab/v1/audit.pdf
Corrected Phase 2	01 Aug 2025

Source Files

Filename	SHA256
TransparentUpgradeableProxy.sol	a1900b226ecf9c4a86fa39ed6ad323414decf0a113c60e79d9aac9e860b62fae
TransferHelper.sol	69764365ecc3ca921384181b1b2a8c18fa09b7675b9825a2f9259ddb66dabbe8
StandardProxy.sol	6d4c71459aff706826cefd493972bfe36f00f2215eab6dfafed69d694736c194
EulerStaking.sol	3f0a4af4bdbd5dcef93227a6584aa8319e98ac6c19c23e4c00af05c0ad191af4
EulerMinting.sol	79002931a8f977f30627331133760f88e9123ae3fc91fa913896204c85f4092a
BenqiStaking.sol	56ebe92641d9a4d97a194e1509883e15e2a6ec6193ce61189320098b602cfdaf
BenqiMinting.sol	e2d8da01dab731d520d4e0ba73a187b61e057e10d013cd05100af362b0f4b800

BankToken.sol	ab87d86fc91d48dabddf8ba7335fda057e84c83a47df7dffc2e7edc17a26fced
BankStaking.sol	e0616e6ca936f19cba088f586235fd49b02a013d5f7352b932964435191e073a
BankMinting.sol	70b6db5fb103c7a05fea31137900b247fb1f1908fea6930f3e172c45861d17a0
structs/BankStructs.sol	57aa4b4b77790bc5ba23ff8874073ca7ea91978bfe06127e84db50f682733807
protocols/euler/IEulerEToken.sol	93da0ea05a742013dad670ea5a61e3a3b3b17d0acf919ad10c1fd2b6948a0e5c
protocols/euler/IEulerDistributor.sol	b705a4840edf011f9e23dfbabaf87565edbc3056ce41d109325a79e16b8290fa
protocols/benqi/IQiErc20.sol	21d6a4097c33f613d0a2b3440143a42f14f3225b3a3fe335aa98c8273740653a
protocols/benqi/IComptrollerRewards.sol	253be8e08d8edf778146058bab3ea4603ba379bd6599bdc5273b3f88019ba5e5
interfaces/IBankToken.sol	d254223644b14b54f971c7963556c1adc095c6aed091e7d6385320540d142f6a
interfaces/IBankStaking.sol	2f86c55f24aad4572059dede0e564998e1ad3637ac86eb293e0c8e02b15c4160
interfaces/IBankMinting.sol	8a318b522b8e0d04ad8960d47b73508970b29359e77dc1eb17e7fba9e4bd528c

Overview

BankToken

The `BankToken` contract is a mintable and burnable ERC20 token used as the system's staking token. It follows the OpenZeppelin upgradeable ERC20 implementation and defines a custom decimal value of six to match assets like USDC. Access control is enforced using the `AccessControlUpgradeable` module. Only accounts with the `MINTER_ROLE` can mint or burn tokens. The contract also features a blacklist system to prevent transfers to or from malicious or compromised addresses. Minting and burning operations are expected to be executed by trusted minting contracts, and ownership controls allow the system administrator to manage minters and blacklist entries.

BankStaking

The `BankStaking` abstract contract manages the staking logic, reward accounting, and user interactions. It tracks how much of the `BankToken` each user has staked and calculates rewards based on interest accrued by the underlying lending protocol. Interest is periodically calculated and distributed via an `accRewardPerShare` system, which ensures fair proportional distribution. The contract also supports reward claiming, compounding, and unminting back to the original base token. Additionally, it includes mechanisms for delayed unminting to support liquidity safety, reward fee handling, and dust tracking for fractional rewards that fall below the minimum claim threshold.

BankMinting

The `BankMinting` abstract contract manages the logic for minting and unminting tokens, acting as the bridge between user deposits and the underlying yield-bearing protocol. When a user deposits a base token, the minting contract converts it into a protocol token, mints an equal amount of `BankToken`, and deposits the funds into the selected protocol. Conversely, when a user unstakes or requests a withdrawal, the contract burns their `BankToken` and retrieves the equivalent base token, accounting for any fees. The contract supports both instant unminting and delayed unminting, where funds become available after a defined time window. It is designed to be protocol-agnostic, requiring derived contracts to implement protocol-specific logic for deposits, withdrawals, and value calculations.

EulerStaking

The `EulerStaking` contract extends the `BankStaking` abstract contract and provides the integration with the Euler protocol. It calculates the total underlying value of assets deposited in Euler by converting the eToken balance into its equivalent base token value using the current exchange rate. It also provides view functions to preview redemption values and determine the amount of liquidity available for withdrawals.

EulerMinting

The `EulerMinting` contract extends the `BankMinting` abstract contract and implements the specific logic for interacting with Euler’s eTokens. When a user deposits base tokens, the contract calls the `deposit()` function on Euler's eToken and mints the corresponding `BankToken` . On withdrawal, it calls `withdraw()` on the eToken to redeem the underlying base token. The expected value is calculated using `convertToAssets` , which reflects the protocol's internal exchange rate. Additionally, the contract can claim protocol-level rewards distributed by Euler's distributor, which are forwarded to the designated fee receiver.

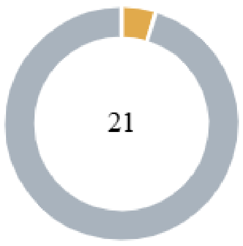
BenqiStaking

The `BenqiStaking` contract is a protocol-specific implementation of the `BankStaking` base contract tailored for integration with the Benqi protocol. It calculates the total protocol value by retrieving the balance of qiTokens held by the minting contract and multiplying it by the current exchange rate. The rest of the staking logic, including user balance management, reward distribution, and fee handling, remains identical to the base staking contract, ensuring consistent user experience across supported protocols.

BenqiMinting

The `BenqiMinting` contract implements the `BankMinting` abstract contract for the Benqi protocol. It supplies user-deposited base tokens into Benqi by calling `mint()` on the qiToken contract and retrieves them using `redeemUnderlying()` . The expected value of deposited tokens is calculated using the current exchange rate, obtained via `exchangeRateCurrent()`. The contract includes error-handled interactions with Benqi to ensure safe deposit and withdrawal operations. It also supports claiming protocol rewards via the `claimReward()` function, using Benqi’s reward distribution system. Rewards are sent to the specified recipient, and the minting logic remains consistent with the platform's modular design.

Findings Breakdown



●	Critical	0
●	Medium	1
●	Minor / Informative	20

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	1	0	0
●	Minor / Informative	0	20	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🟠	PUF	Possible Undercollateralization Freeze	Acknowledged
⚪	MVN	Misleading Variables Naming	Acknowledged
⚪	IPRD	Inconsistent Pending Reward Dust	Acknowledged
⚪	AVV	Accumulated Variable Vulnerabilities	Acknowledged
⚪	BC	Blacklists Addresses	Acknowledged
⚪	CCR	Contract Centralization Risk	Acknowledged
⚪	DDC	Delegator Dependency Concern	Acknowledged
⚪	EDA	Early Deposit Advantage	Acknowledged
⚪	ETWC	Exact Tokens Withdrawal Concern	Acknowledged
⚪	IRP	Internal Reentrance Protection	Acknowledged
⚪	MT	Mints Tokens	Acknowledged
⚪	PMO	Possible MEV Opportunities	Acknowledged
⚪	PICD	Potential Interest Cap Discrepancy	Acknowledged
⚪	PSR	Potential Stale Rewards	Acknowledged

●	PSVD	Potential Symmetry Value Discrepancy	Acknowledged
●	PTAI	Potential Transfer Amount Inconsistency	Acknowledged
●	SDC	Stop Delayed Claim	Acknowledged
●	UT	Unburnable Tokens	Acknowledged
●	UTPD	Unverified Third Party Dependencies	Acknowledged
●	VIBM	Volatile Interest Based Minting	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged

PUF - Possible Undercollateralization Freeze

Criticality	Medium
Location	BankStaking.sol#L192,296,325,346,544
Status	Acknowledged

Description

During critical protocol actions, the `BankStaking` contract verifies whether the system is adequately collateralized. This is done by comparing the current amount of underlying assets with the total amount of `bankTokens` staked in the contract. If the current assets are insufficient, these actions will revert. This can occur under normal circumstances during the use of the external protocols, where the value of shares may decrease due to processes such as sudden market downturns or unexpected liquidation events. Consequently, users will be unable to perform functions within the protocol, such as withdrawing tokens, claiming rewards, or staking and unstaking tokens. Additionally, malicious actors may target the contract and utilize the "freeze on undercollateralisation" rule imposing real economic losses through denial-of-service, liquidation bonuses and trading gains.

```
function _updatePool() internal {
        require(checkSolvency(), "System is undercollateralized!");
        ...
}
...
require(checkSolvency(), "System is undercollateralized after");
...
function checkSolvency() public returns (bool) {
        uint256 liability = getTotalStakePlusInterest();
        uint256 getUnderlying = _getUnderlying();
        return getUnderlying >= liability;
}
```

Recommendation

It is recommended to implement a more flexible approach to handle undercollateralization scenarios in the `BankStaking` contract. Instead of fully reverting critical actions when the system is undercollateralized, consider introducing a grace period or alternative mechanism that allows users to still interact with the protocol, such as limiting certain functions or applying penalties.

Team Update

The team has acknowledged that this is not a security issue and states:

The strict solvency check is an intentional security mechanism to ensure all circulating bankToken is fully backed by underlying assets at all times. This design prevents withdrawals or claims if the system becomes undercollateralized, protecting remaining depositors during market stress events and avoiding unfair fund distribution.

No grace period or partial withdrawal mechanism will be implemented, as this could weaken solvency guarantees and increase complexity. The freeze-on-undercollateralization rule will remain to prioritize depositor safety over availability in distressed conditions.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	BankStaking.sol#L209
Status	Acknowledged

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

Specifically, `maxDailyInterest` does not represent a daily basis, as `interest` updates can occur multiple times in a day. Additionally, the protocol may not have been used for more than one day.

```
uint256 maxDailyInterest = (totalStakedAmount * maxInterestBps) /
BPS_DENOMINATOR;
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

Team Update

The team has acknowledged that this is not a security issue and states:

The variable name maxDailyInterest originates from the interest calculation method, which is based on a daily rate even though updates may occur multiple times per day. This naming choice does not affect system logic or security. Given the minimal impact and clear internal understanding of its meaning, no changes will be made.

IPRD - Inconsistent Pending Reward Dust

Criticality	Minor / Informative
Location	BankStaking.sol#L371
Status	Acknowledged

Description

The staking contract tracks unclaimable fractional rewards using a `pendingRewardDust` field, which accumulates when rewards are too small to be claimed due to the `MIN_AMOUNT` threshold. However, when a user withdraws some of their stake, this dust value is retained without being scaled down proportionally to the remaining stake. This creates a discrepancy between the user's actual share of rewards and what is tracked, allowing a user to earn dust on a large stake, withdraw most of their position, and later claim the entire dust amount despite only holding a smaller share.

```
function _withdraw(address _account, uint256 _amount, bool
_isMintToken) internal returns (uint256) {
        ...
        (uint256 rewardToPay, uint256 fee, uint256 scaled) =
_calcRewardParts(_account);
        ...
        user.amount -= _amount;
        user.pendingRewardDust = (user.amount == 0) ? 0 : (scaled %
rewardDecimalsDivisor);
        user.rewardDebt = user.amount * accRewardPerShare /
decimalFactor;
        ...
}
```

Recommendation

To ensure accuracy, the `pendingRewardDust` should be adjusted when a user withdraws part of their stake, scaling it according to the ratio of their remaining balance.

Team Update

The team has acknowledged that this is not a security issue and states:

If the user withdraws their entire stake, pendingRewardDust is now reset to 0. Otherwise, the dust is retained until the reward becomes claimable.

AVV - Accumulated Variable Vulnerabilities

Criticality	Minor / Informative
Location	BankStaking.sol#L222,229 BankMinting.sol#L108,126,164 BankToken.sol#L68
Status	Acknowledged

Description

The contract is using variables to accumulate values. The contract could lead to an overflow when the total value of a variable exceeds the maximum value that can be stored in that variable's data type. This can happen when an accumulated value is updated repeatedly over time, and the value grows beyond the maximum value that can be represented by the data type.

Additionally, there are certain variables that may alway decrease creating a possible underflow.

```
accRewardPerShare += add;
totalAccruedInterest += interest;
```

```
bankToken.mint(msg.sender, amount);
totalMintedAmount += amount;
userMintedAmount[account] += int256(amount);
...
uint256 prevUnMinted = totalUnMintedAmount;
totalUnMintedAmount += amount;
userMintedAmount[account] -= int256(amount);
```

```
function mint(address to, uint256 amount) external
onlyRole(MINTER_ROLE) {
    _mint(to, amount);
}
```

Recommendation

The team is advised to carefully investigate the usage of the variables that accumulate value positively or negatively. A suggestion is to add checks to the code to ensure that the value of a variable does not exceed the maximum or minimum value that can be stored in its data type.

Team Update

The team has acknowledged that this is not a security issue and states:

Solidity 0.8.x and above provides built-in overflow/underflow checks, so additional protection is unnecessary.

BC - Blacklists Addresses

Criticality	Minor / Informative
Location	BankToken.sol#L81,97
Status	Acknowledged

Description

The contract owner has the authority to stop addresses from transactions. The owner may take advantage of it by calling the `blacklistAddress` function.

```
function addBlacklist(address account) external onlyOwner {
    require(!_blacklist[account], "Already blacklisted");
    _blacklist[account] = true;
    emit Blacklisted(account);
}
function _beforeTokenTransfer(address from, address to, uint256 amount)
internal override {
    require(!_blacklist[from], "Sender is blacklisted");
    require(!_blacklist[to], "Recipient is blacklisted");
    super._beforeTokenTransfer(from, to, amount);
}
```

Recommendation

The team should carefully manage the private keys of the owner’s account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

Team Update

The team has acknowledged that this is not a security issue and states:

Admin privileges will be restricted and securely managed with a hardware wallet after deployment.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	BankToken.sol#L37,56,61,68,72,81,87 TransparentUpgradeableProxy.sol#L42,46,50,54,59,64,77,83 BankMinting.sol#L69,81,87,75,103,123,158,178 BankStaking.sol#L126,135,142,149,156,163,174,180,252,269,309,366,410,436 BenqiMinting.sol#L39,45 EulerMinting.sol#L34,44
Status	Acknowledged

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function transferOwnership(address newOwner) public override onlyOwner
function grantMinterRole(address account) external onlyOwner
function revokeMinterRole(address account) external onlyOwner
function mint(address to, uint256 amount) external
onlyRole(MINTER_ROLE)
function burn(address from, uint256 amount) external
onlyRole(MINTER_ROLE)
function addBlacklist(address account) external onlyOwner
function removeBlacklist(address account) external onlyOwner
function admin() external ifAdmin returns (address admin_)
function nextAdmin() external ifAdmin returns (address nextAdmin_)
function implementation() external ifAdmin returns (address
implementation_)
function changeAdmin(address newAdmin) external virtual ifAdmin
function changeNextAdmin(address newNextAdmin) external ifAdmin
function receiveAdmin() external {
    address next = _nextAdmin();
    require(msg.sender == next);
    ...
}
function upgradeTo(string calldata version, address newImplementation)
external ifAdmin
function upgradeToAndCall(string calldata version, address
newImplementation, bytes calldata data) external payable ifAdmin
```

```
function setStakingContract(address _stakingContract) external
onlyOwner
function setUnmintFeeBps(uint256 _bps) external onlyOwner
function setUnmintFeeReceiver(address _receiver) external onlyOwner
function setDelayedUnmintEnabled(bool enabled) external onlyOwner
function mint(address account, uint256 amount) external override
nonReentrant onlyStakingContract
function unmint(address account, uint256 amount, uint256 reward)
external override nonReentrant onlyStakingContract returns (uint256)
function requestDelayedUnmint(address account, uint256 amount) external
override nonReentrant onlyStakingContract
function claimDelayedUnmintAt(address account, uint256 idx) external
override nonReentrant onlyStakingContract returns (uint256)
```

```
function setMaxInterestBps(uint256 _bps) external onlyOwner
function distributePendingInterest() external onlyOwner
function setMintingContract(address _mintingContract) public onlyOwner
function setPairPoolFeeReceiver(address _pairPoolFeeReceiver) public
onlyOwner
function setBuybackFeeReceiver(address _buybackFeeReceiver) public
onlyOwner
function setRewardFeePercent(uint256 _rewardFeePercent) public
onlyOwner
function addDelegator(address _account) public onlyOwner
function removeDelegator(address _account) public onlyOwner
function mintForUser(address _account, uint256 _amount) external
onlyDelegator nonReentrant
function depositForUser(address _account, uint256 _amount, bool
_isMintToken) external onlyDelegator nonReentrant
function unmintForUser(address _account, uint256 _amount) external
onlyDelegator nonReentrant
function withdrawForUser(address _account, uint256 _amount, bool
_isMintToken) external onlyDelegator nonReentrant
function claimRewardForUser(address _account, bool _isMintToken)
external onlyDelegator nonReentrant returns (uint256)
function compoundRewardForUser(address _account) external onlyDelegator
nonReentrant returns (uint256)
```

```
function setProtocolToken(address _ptoken) external onlyOwner
function setRewardManager(address _rewardManager) external override
onlyOwner
```

```
function setProtocolToken(address _ptoken) external onlyOwner
function setRewardManager(address _rewardManager) external override
onlyOwner
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

Team Update

The team has acknowledged that this is not a security issue and states:

Admin privileges will be restricted and securely managed with a hardware wallet after deployment.

DDC - Delegator Dependency Concern

Criticality	Minor / Informative
Location	BankStaking.sol#L252,269,309,366,410,436
Status	Acknowledged

Description

The functions that have the `onlyDelegator` modifier allow certain addresses to control the users' tokens and make actions for them. While this provides flexibility it introduces a centralization risk. For example a delegator could withdraw tokens for a user without their concern. This will effectively burn the user's `bankTokens`.

```
function mintForUser(address _account, uint256 _amount) external
onlyDelegator nonReentrant
function depositForUser(address _account, uint256 _amount, bool
_isMintToken) external onlyDelegator nonReentrant
function unmintForUser(address _account, uint256 _amount) external
onlyDelegator nonReentrant
function withdrawForUser(address _account, uint256 _amount, bool
_isMintToken) external onlyDelegator nonReentrant
function claimRewardForUser(address _account, bool _isMintToken)
external onlyDelegator nonReentrant returns (uint256)
function compoundRewardForUser(address _account) external onlyDelegator
nonReentrant returns (uint256)
```

Recommendation

The team should carefully manage the private keys of the delegator's accounts as well as the owner's that is able to set new delegators. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

Team Update

The team has acknowledged that this is not a security issue and states:

Delegator can only execute functions identical to those the user can execute directly.

EDA - Early Deposit Advantage

Criticality	Minor / Informative
Location	BankStaking.sol#L220
Status	Acknowledged

Description

`_updatePool` checks if `totalStakedAmount` is greater than 0 and if it is it updates the `accRewardPerShare`. However it is possible that users use the `mint` function first. As the `totalStakedAmount` is not being updated the `accRewardPerShare` will stay 0. The first user that deposits will receive rewards based on the entire amount of tokens held as underlying in the external protocol as `accRewardPerShare` will be 0 and updated only after the deposit.

```
if (totalStakedAmount > 0) {
    uint256 add = interest * ACC_REWARD_PER_SHARE_SCALE /
totalStakedAmount;
    accRewardPerShare += add;
}
```

Recommendation

It is recommended that the team handles the potential of users minting their tokens first instead of depositing.

Team Update

The team has acknowledged that this is not a security issue and states:

MEV or early deposit risks are not an issue, since team handles all initial deposits.

ETWC - Exact Tokens Withdrawal Concern

Criticality	Minor / Informative
Location	BankMinting.sol#L142,187 BankStaking.sol#L342,376
Status	Acknowledged

Description

The contract uses `_withdrawFromProtocol` to receive the exact amount of tokens requested. However, if the amount returned is not equal to the requested amount, the transaction reverts. While in normal cases this will ensure that the contract will receive the exact amount asked and the returned amount will be equal to the requested, it is possible that external applications may fail or be altered. This will result in users not being able to claim their tokens.

```
uint256 pre = IERC20(mintToken).balanceOf(address(this));
_withdrawFromProtocol(amount);
uint256 post = IERC20(mintToken).balanceOf(address(this));
require(post - pre == amount, "redeemed != expected");
```

The case is similar for `BankStaking` strict restrictions on unmint and withdraw functions.

```
require(lastUnderlying >= amount, "insufficient underlying");
if (_isMintToken) require(getWithdrawableAmount() >= _amount,
"insufficient liquidity");
```

Recommendation

It is recommended that the team handles the potential of external interactions not working as expected. Additionally the team should allow users to choose if they want to receive less value than the expected. This could be achieved by implementing a threshold option for the users.

Team Update

The team has acknowledged that this is not a security issue and states:

1:1 redemption required; tx reverts if not exact.

IRP - Internal Reentrance Protection

Criticality	Minor / Informative
Location	BenqiStaking.sol#L31 BenqiMinting.sol#L52,65 BankStaking.sol#L259,274,318,371,414,440 BankMinting.sol#L134 EulerMinting.sol#L50,61
Status	Acknowledged

Description

The contract has a lot of internal functions that can be called through the rest of its functionality. While external parties cannot create reentrancies due to the non-reentrant modifier on external functions it is always a good practice to also protect the contract's internal functions.

```
function _getUnderlying() internal override returns (uint256)
function _depositToProtocol(uint256 amount) internal override returns
(uint256)
function _withdrawFromProtocol(uint256 amount) internal override
function _unmintInstant(address account, uint256 amount) internal
returns (uint256)
function _mint(address _account, uint256 _amount) internal
function _deposit(address _account, uint256 _amount, bool _isMintToken)
internal
function _unmintInstant(address _account, uint256 _amount, uint256
_reward) internal returns (uint256)
function _withdraw(address _account, uint256 _amount, bool
_isMintToken) internal returns (uint256)
function _claimReward(address _account, bool _isMintToken) internal
returns (uint256)
function _compoundReward(address _account) internal returns (uint256)
```

Recommendation

The team could implement some custom modifier to protect the contract and its internal functions from potential reentrances.

Team Update

The team has acknowledged that this is not a security issue and states:

Reentrancy protection is applied to all public/external functions. Internal functions are only called by these protected functions.

MT - Mints Tokens

Criticality	Minor / Informative
Location	BankToken.sol#L68
Status	Acknowledged

Description

Addresses that have `MINTER_ROLE` have the authority to mint tokens. They may take advantage of it by calling the `mint` function. Additionally, the owner is able to set any address as `MINTER_ROLE`. As a result, the contract tokens will be highly inflated.

```
function mint(address to, uint256 amount) external
onlyRole(MINTER_ROLE) {
    _mint(to, amount);
}
```

Recommendation

The team should carefully manage the private keys of the owner's account and of the addresses that are set as `MINTER_ROLE`. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate threats but it is non-reversible.

Team Update

The team has acknowledged that this is not a security issue and states:

Only the contract itself can mint; the owner who can grant the mint role will be managed with a hardware wallet.

PMO - Possible MEV Opportunities

Criticality	Minor / Informative
Location	BankStaking.sol#L143
Status	Acknowledged

Description

During `updatePool` the protocol checks the value of its shares based on the `mintToken`, in order to calculate the interest and to subsequently increase the rewards. However it is possible that users enter the protocol before a significant interest update to earn a disproportionate fraction of the rewards. This could be unfair for existing depositors.

```
function _updatePool() internal {
    require(checkSolvency(), "System is undercollateralized!");
    uint256 underlying = _getUnderlying();
    if (underlying <= lastUnderlying) {
        if (underlying < lastUnderlying) {
            emit LossOccurred(lastUnderlying, underlying);
        }
        return;
    }
    uint256 interest = underlying - lastUnderlying;
    uint256 maxDailyInterest = (totalStakedAmount * maxInterestBps) /
BPS_DENOMINATOR;
    if (interest > maxDailyInterest) {
        uint256 excess = interest - maxDailyInterest;
        pendingInterest += excess;
        emit ExcessInterestCapped(interest, maxDailyInterest, excess);
        interest = maxDailyInterest;
    }
    if (totalStakedAmount > 0) {
        uint256 add = interest * ACC_REWARD_PER_SHARE_SCALE /
totalStakedAmount;
        accRewardPerShare += add;
    }
    ...
}
```

Recommendation

The contract should be fair in the case of price distribution. This could be achieved by implementing a time based system where not only the amount of deposited tokens is used to calculate users' rewards but also the duration of each deposit.

Team Update

The team has acknowledged that this is not a security issue and states:

MEV or early deposit risks are not an issue, since team handles all initial deposits.

PICD - Potential Interest Cap Discrepancy

Criticality	Minor / Informative
Location	BankStaking.sol#L180,188
Status	Acknowledged

Description

The `BankStaking` contract performs pool updates every time a critical action is triggered. The `_updatePool` function mints tokens based on the interest calculated by subtracting the last recorded underlying amount from the current underlying amount. The `lastUnderlying` value is then updated to reflect the current underlying amount, ensuring that each `bankToken` is backed by one `mintToken`. However, if the calculated interest exceeds a defined `maxDailyInterest`, fewer bankTokens are minted, but the `lastUnderlying` is still updated to the current underlying amount. This creates a discrepancy between the number of `bankTokens` and the actual minted `mintTokens`, resulting in fewer rewards for users. Additionally, the excess interest, which is the difference between the calculated interest and the `maxDailyInterest`, becomes unusable and can only be reset by the contract owner.

```
function distributePendingInterest() external onlyOwner {
    require(pendingInterest > 0, "No pending interest");
    emit PendingInterestDistributed(pendingInterest);
    pendingInterest = 0;
}
function _updatePool() internal {
    ...
    uint256 interest = underlying - lastUnderlying;
    uint256 maxDailyInterest = (totalStakedAmount * maxInterestBps) /
BPS_DENOMINATOR;
    if (interest > maxDailyInterest) {
        uint256 excess = interest - maxDailyInterest;
        pendingInterest += excess;
        emit ExcessInterestCapped(interest, maxDailyInterest, excess);
        interest = maxDailyInterest;
    }
    if (totalStakedAmount > 0) {
        uint256 add = interest * ACC_REWARD_PER_SHARE_SCALE /
totalStakedAmount;
        accRewardPerShare += add;
    }
    lastUnderlying = underlying;
    ...
    totalAccruedInterest += interest;
    IBankToken(bankToken).mint(address(this), interest);
    ...
}
```

Recommendation

It is recommended to modify the `BankStaking` contract to ensure consistency between the minted `bankTokens` and the calculated interest, even when the `maxDailyInterest` cap is exceeded.

Team Update

The team has acknowledged that this is not a security issue and states:

A maximum interest cap (maxDailyInterest) is intentionally defined to mitigate abnormal interest spikes that could occur due to issues in the external protocols where funds are deposited. If the calculated interest exceeds this cap, the excess is stored in pendingInterest for potential manual distribution by the owner, ensuring that user rewards are not permanently lost.

The 0.1% cap is based on observed protocol volatility and operational experience, and any larger spike is treated as a non-normal event. In such cases, capping prevents over-minting of bankToken and preserves collateralization. If the event is temporary, the capped amount remains fully backed, and subsequent pool updates and solvency checks will detect the recovery, allowing safe resumption or manual distribution of the pending amount.

This mechanism is a deliberate safeguard to prioritize solvency and user fund safety over immediate, full interest distribution during abnormal conditions.

PSR - Potential Stale Rewards

Criticality	Minor / Informative
Location	BankStaking.sol#L196
Status	Acknowledged

Description

`updatePool` checks if the underlying tokens are greater than the `lastUnderlying` storage variable in the contract. If it is, the `interest` is calculated and minted and the accumulated rewards are updated. However if users withdraw or claim rewards it is possible that `lastUnderlying` will be greater than the actual underlying. This will result in rewards not being updated until the amount of underlying tokens is updated and becomes greater than `lastUnderlying`. Additionally, it is possible that value per share can decrease over time. This means that `lastUnderlying` will not be updated and the contract will not provide rewards. A possible scenario could be that collateral placed in the external dependencies by borrowers could crash. If this happens the external protocol may suffer loss due to a bad debt. As a result the underlying will be lower than `lastUnderlying` and rewards will not be provided.

```
if (underlying <= lastUnderlying) {
    if (underlying < lastUnderlying) {
        emit LossOccurred(lastUnderlying, underlying);
    }
    return;
}
```

Recommendation

The contract should ensure that users receive their rewards for the time they deposit their tokens in the protocol. Additionally, the contract should handle the potential of value per share being reduced.

Team Update

The team has acknowledged that this is not a security issue and states:

Interest is only generated when the underlying asset value increases compared to lastUnderlying. If the value remains the same or decreases, no interest is accrued, and therefore no rewards are distributed. This is intentional, as rewards must be backed by real yield from the underlying protocols. The system does not issue rewards without actual asset growth, ensuring all distributed rewards are fully supported by realized gains.

PSVD - Potential Symmetry Value Discrepancy

Criticality	Minor / Informative
Location	BankToken.sol BankStaking.sol BankMinting.sol
Status	Acknowledged

Description

The protocol uses `BankToken` as the currency minted when users deposit or use the `mintToken`. However, if the same `BankToken` is utilized across different protocol configurations, there is a risk of value discrepancy. Since the price of the token may vary depending on the underlying minting asset, this can lead to potential value loss. For instance, if mintTokens used are `USDT` in one configuration and `AVAX` in another, and both configurations use the same `BankToken`, users could deposit `BankToken` earned through `USDT` into a protocol configuration based on `AVAX`. This mismatch could cause an imbalance in token values across the two implementations.

Recommendation

It is recommended that each implementation is using a separate `BankToken` to avoid the potential loss of value.

Team Update

The team has acknowledged that this is not a security issue and states:

For deployment, bUSDT and bAVAX contracts will be deployed for each asset (e.g., USDT, AVAX), so each token with the same value will be managed separately.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	BankStaking.sol#L260 BankMinting.sol#L106,151,195
Status	Acknowledged

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
TransferHelper.safeTransferFrom(mintToken, _account, address(this),
_amount);
```

```
TransferHelper.safeTransferFrom(mintToken, msg.sender, address(this),
amount);
if (payout > 0) TransferHelper.safeTransfer(mintToken, msg.sender,
payout);
TransferHelper.safeTransfer(mintToken, account, amount);
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

`Actual Transferred Amount = Balance After Transfer - Balance Before Transfer`

Team Update

The team has acknowledged that this is not a security issue and states:

Only standard ERC-20 tokens such as USDT, USDC, WETH, BTC.b, etc. will be used.

SDC - Stop Delayed Claim

Criticality	Minor / Informative
Location	BankMinting.sol#L178
Status	Acknowledged

Description

The contract allows users to claim their delayed unmints after a certain period of time. However delayed claiming can be stopped by the owner by using the `setDelayedUnmintEnabled` function. This will result in users that already requested a delayed unmint to not be able to claim their tokens.

```
function claimDelayedUnmintAt(address account, uint256 idx) external
override nonReentrant onlyStakingContract returns (uint256) {
    require(delayedUnmintEnabled, "Delayed unmint is not enabled");
    ...
}
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

Team Update

The team has acknowledged that this is not a security issue and states:

Delayed unmint functionality is disabled by default and not used for public launch. After deployment, the owner account will be managed securely via a hardware wallet.

UT - Unburnable Tokens

Criticality	Minor / Informative
Location	BankToken.sol#L81,97
Status	Acknowledged

Description

`_beforeTokenTransfer` checks if an address is blacklisted. If it then the transaction reverts. However `_burn` function also uses `_beforeTokenTransfer` meaning that tokens owned by blacklisted addresses cannot be burned.

```
function addBlacklist(address account) external onlyOwner {
    require(!_blacklist[account], "Already blacklisted");
    _blacklist[account] = true;
    emit Blacklisted(account);
}
function _beforeTokenTransfer(address from, address to, uint256 amount)
internal override {
    require(!_blacklist[from], "Sender is blacklisted");
    require(!_blacklist[to], "Recipient is blacklisted");
    super._beforeTokenTransfer(from, to, amount);
}
```

Recommendation

It is recommended that the team checks if blacklisted addresses should be able to get their tokens burned.

Team Update

The team has acknowledged that this is not a security issue and states:

Burning or confiscating assets from blacklisted accounts may introduce additional risks, so the account will only be frozen. Unfreezing will require further explanation or review.

UTPD - Unverified Third Party Dependencies

Criticality	Minor / Informative
Location	BenqiStaking.sol#L8 BenqiMinting.sol#L11,12 EulerMinting.sol#L10,11 EulerStaking.sol#L8 BankStaking.sol#L42 BankMinting.sol#L30
Status	Acknowledged

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

Additionally, most of the external interactions are with contracts that are upgradable. This may hinder the contract's functionality as the functions used may change.

```
IQiErc20 public protocolToken;
```

```
IQiErc20 public protocolToken;
IComptrollerRewards public rewardManager;
```

```
IEulerEToken public protocolToken;
IEulerDistributor public rewardManager;
```

```
IEulerEToken public protocolToken;
```

```
address public override mintToken;
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization. The team should also ensure that any contract interacting with the protocol is properly verified by the relevant protocol authority.

Team Update

The team has acknowledged that this is not a security issue and states:

Euler and Benqi are already audited and stable, but implemented setProtocolToken as a separate function for future flexibility if needed

VIBM - Volatile Interest Based Minting

Criticality	Critical
Location	BankStaking.sol#L188
Status	Acknowledged

Description

The `_updatePool` function calculates the value of the shares it holds based on the `mintToken`. It compares the last recorded underlying value (`lastUnderlying`) with the current underlying value. If the current value is higher, the difference — representing accrued interest — is used to estimate rewards. The `lastUnderlying` is then updated to match the current value, and new `bankTokens` are minted in proportion to the calculated interest.This approach can be used to create a scenario where each `bankToken` is backed up by one `mintToken`. This mechanism effectively pegs each `bankToken` to one `mintToken` based on internal accounting. However, since `bankTokens` are standard ERC-20 tokens, they can be freely traded on secondary markets. This creates a potential vulnerability as the market price of `bankTokens` may deviate from the internal valuation maintained by the contract. Despite such discrepancies,the contract continues to mint bankTokens at a fixed internal rate, ignoring their actual market value, possibly creating arbitrage opportunities.

```
function _updatePool() internal {
    ...
    uint256 interest = underlying - lastUnderlying;
    uint256 maxDailyInterest = (totalStakedAmount * maxInterestBps) /
BPS_DENOMINATOR;
    if (interest > maxDailyInterest) {
        uint256 excess = interest - maxDailyInterest;
        pendingInterest += excess;
        emit ExcessInterestCapped(interest, maxDailyInterest, excess);
        interest = maxDailyInterest;
    }
    if (totalStakedAmount > 0) {
        uint256 add = interest * ACC_REWARD_PER_SHARE_SCALE /
totalStakedAmount;
        accRewardPerShare += add;
    }
    lastUnderlying = underlying;
    ...
    totalAccruedInterest += interest;
    IBankToken(bankToken).mint(address(this), interest);
    ...
}
```

Recommendation

The team should ensure that the contracts enable defensive mechanisms to protect against potential manipulations. The team should additionally consider the risks associated with the `BankStaking` contract's interaction with secondary markets. The ability to mint `bankTokens` based on a 1:1 ratio with `mintTokens` can create opportunities for profit in scenarios where the ratio in external markets deviates.

Team Update

The team has acknowledged that this is not a security issue and states:

bankToken is strictly backed and redeemable 1:1 for mintToken via the protocol's unmint/withdraw paths, subject to liquidity and solvency guards. We deliberately mint rewards against realized underlying growth—not secondary-market prices. If an external market price deviates, arbitrage (buy discounted bankToken → redeem for mintToken) drives convergence back to parity.

Risk controls already in place: strict solvency checks (isSolvent/checkSolvency), per-tx liquidity checks (getWithdrawableAmount()), interest-spike caps with pendingInterest, and pausing. Given these mechanisms, we do not couple minting to secondary-market moves; pricing divergence does not create unbacked issuance, and parity is economically enforced by redemption. No change planned.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	EulerStaking.sol#L11,12,13,14,15,16,43 EulerMinting.sol#L14,15,16,17,18,34,44 BenqiStaking.sol#L11,12,13,14,15,16,47 BenqiMinting.sol#L15,16,17,18,19,39,45 BankStaking.sol#L76,77,78,79,80,81,126,135,142,149,156,163,170,174,245,252,266,269,301,309,330,363,366,407,410,436,498,515,519 BankMinting.sol#L43,69,75,81
Status	Acknowledged

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address _mintingContract
address _bankToken
address _mintToken
address _pairPoolFeeReceiver
address _buybackFeeReceiver
address _protocolToken
uint256 _qty
address _pToken
address _rewardManager
address _feeReceiver
address _ptoken

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

Team Update

The team has acknowledged that this is not a security issue and states:

Some internal variable names intentionally use a leading underscore (_) for clarity and to differentiate constructor or initializer parameters from storage variables. This naming style is consistent within the codebase and does not affect functionality, security, or maintainability. Given the minimal impact on readability for the current team and established internal conventions, no changes will be made.

Function Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IVersion	Interface			
	version	External		-
TransparentUpgradeableProxy	Implementation	ERC1967Proxy		
		Public	Payable	ERC1967Proxy
	admin	External	✓	ifAdmin
	nextAdmin	External	✓	ifAdmin
	implementation	External	✓	ifAdmin
	changeAdmin	External	✓	ifAdmin
	changeNextAdmin	External	✓	ifAdmin
	receiveAdmin	External	✓	-
	_changeNextAdmin	Private	✓	
	upgradeTo	External	✓	ifAdmin
	upgradeToAndCall	External	Payable	ifAdmin
	_nextAdmin	Internal		
	_beforeFallback	Internal	✓	
TransferHelper	Library			
	safeApprove	Internal	✓	
	safeTransfer	Internal	✓	

	safeTransferFrom	Internal	✓	
	safeTransferETH	Internal	✓	
StandardProxy	Implementation	Transparent Upgradeable Proxy		
		Public	✓	TransparentUpgradeableProxy
	getAdmin	Public		-
	getImplementation	Public		-
	getNextAdmin	Public		-
		External	Payable	-
EulerStaking	Implementation	BankStaking		
	initialize	Public	✓	initializer
	version	External		-
	getProtocolToken	External		-
	_getUnderlying	Internal		
	getUnderlyingView	Public		-
	getWithdrawableAmount	Public		-
	previewRedeem	External		-
EulerMinting	Implementation	BankMinting		
	initialize	Public	✓	initializer
	version	External		-
	protocolType	External		-
	setProtocolToken	External	✓	onlyOwner

	_protocolToken	Internal		
	setRewardManager	External	✓	onlyOwner
	_depositToProtocol	Internal	✓	
	_withdrawFromProtocol	Internal	✓	
	_expectedValue	Internal		
	pendingProtocolReward	Public		-
	claimProtocolReward	External	✓	nonReentrant
BenqiStaking	Implementation	BankStaking		
	initialize	Public	✓	initializer
	version	External		-
	getProtocolToken	External		-
	_getUnderlying	Internal	✓	
	getUnderlyingView	Public		-
	getWithdrawableAmount	Public		-
	previewRedeem	External		-
	_expectedValue	Internal		
BenqiMinting	Implementation	BankMinting		
	initialize	Public	✓	initializer
	version	External		-
	protocolType	External		-
	_protocolToken	Internal		
	setProtocolToken	External	✓	onlyOwner
	setRewardManager	External	✓	onlyOwner

	_depositToProtocol	Internal	✓	
	_withdrawFromProtocol	Internal	✓	
	_expectedValue	Internal	✓	
	pendingProtocolReward	Public		-
	claimProtocolReward	External	✓	-
BankToken	Implementation	Initializable, ERC20Upgradeable, OwnableUpgradeable, AccessControlUpgradeable		
	initialize	Public	✓	initializer
	version	External		-
	decimals	Public		-
	transferOwnership	Public	✓	onlyOwner
	renounceOwnership	Public	✓	onlyOwner
	isMinter	Public		-
	grantMinterRole	External	✓	onlyOwner
	revokeMinterRole	External	✓	onlyOwner
	mint	External	✓	onlyRole
	burn	External	✓	onlyRole
	addBlacklist	External	✓	onlyOwner
	removeBlacklist	External	✓	onlyOwner
	isBlacklisted	Public		-
	_beforeTokenTransfer	Internal	✓	

BankStaking	Implementation	IBankStaking, OwnableUpgradeable, ReentrancyGuardUpgradeable		
	__BankStaking_init	Internal	✓	onlyInitializing
	_getUnderlying	Internal	✓	
	getUnderlyingView	Public		-
	getProtocolToken	External		-
	getWithdrawableAmount	Public		-
	previewRedeem	External		-
	setMintingContract	Public	✓	onlyOwner
	setPairPoolFeeReceiver	Public	✓	onlyOwner
	setBuybackFeeReceiver	Public	✓	onlyOwner
	setRewardFeePercent	Public	✓	onlyOwner
	addDelegator	Public	✓	onlyOwner
	removeDelegator	Public	✓	onlyOwner
	isDelegator	Public		-
	setMaxInterestBps	External	✓	onlyOwner
	distributePendingInterest	External	✓	onlyOwner
	updatePool	External	✓	nonReentrant
	_updatePool	Internal	✓	
	mint	External	✓	nonReentrant
	mintForUser	External	✓	onlyDelegator nonReentrant
	_mint	Internal	✓	
	deposit	External	✓	nonReentrant

	depositForUser	External	✓	onlyDelegator nonReentrant
	_deposit	Internal	✓	
	unmint	External	✓	nonReentrant
	unmintForUser	External	✓	onlyDelegator nonReentrant
	_unmintInstant	Internal	✓	
	requestDelayedUnmint	External	✓	nonReentrant
	getDelayedUnmints	External		-
	claimDelayedUnmintAt	External	✓	nonReentrant
	withdraw	External	✓	nonReentrant
	withdrawForUser	External	✓	onlyDelegator nonReentrant
	_withdraw	Internal	✓	
	claimReward	External	✓	nonReentrant
	claimRewardForUser	External	✓	onlyDelegator nonReentrant
	_claimReward	Internal	✓	
	compoundReward	External	✓	nonReentrant
	compoundRewardForUser	External	✓	onlyDelegator nonReentrant
	_compoundReward	Internal	✓	
	_distributeRewardAndFees	Internal	✓	
	_calcRewardParts	Internal		
	pendingReward	External		-
	getUserInfo	External		-
	getStakingAmount	External		-
	userLength	External		-

	getUnclaimedInterest	Public		-
	getLogicalTotal	Public		-
	getTotalStakePlusInterest	Public		-
	checkSolvency	Public	✓	-
BankMinting	Implementation	IBankMinting, Initializable, OwnableUpgradeable, ReentrancyGuardUpgradeable		
	__BankMinting_init	Internal	✓	onlyInitializing
	_depositToProtocol	Internal	✓	
	_withdrawFromProtocol	Internal	✓	
	_expectedValue	Internal	✓	
	_protocolToken	Internal		
	protocolType	External		-
	setRewardManager	External	✓	-
	setStakingContract	External	✓	onlyOwner
	setUnmintFeeBps	External	✓	onlyOwner
	setUnmintFeeReceiver	External	✓	onlyOwner
	setDelayedUnmintEnabled	External	✓	onlyOwner
	hasMinterRole	External		-
	getCurrentMintedAmount	Public		-
	mint	External	✓	nonReentrant onlyStakingContract
	unmint	External	✓	nonReentrant onlyStakingContract

	_unmintInstant	Internal	✓	
	requestDelayedUnmint	External	✓	nonReentrant onlyStakingContract
	getDelayedUnmints	External		-
	claimDelayedUnmintAt	External	✓	nonReentrant onlyStakingContract
BankStructs	Library			
IEulerEToken	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	approve	External	✓	-
	transferFrom	External	✓	-
	decimals	External		-
	asset	External		-
	cash	External		-
	totalAssets	External		-
	convertToAssets	External		-
	convertToShares	External		-
	deposit	External	✓	-
	mint	External	✓	-
	withdraw	External	✓	-
	redeem	External	✓	-

	accumulatedFees	External		-
	feeReceiver	External		-
	interestRate	External		-
IEulerDistributor	Interface			
	claim	External	✓	-
	getMerkleRoot	External		-
	claimed	External		-
	claimWithRecipient	External	✓	-
IQiErc20	Interface	IERC20		
	balanceOf	External		-
	decimals	External		-
	getCash	External		-
	exchangeRateStored	External		-
	exchangeRateCurrent	External	✓	-
	mint	External	✓	-
	redeem	External	✓	-
	redeemUnderlying	External	✓	-
	borrow	External	✓	-
	repayBorrow	External	✓	-
	totalBorrows	External		-
	borrowIndex	External		-
	balanceOfUnderlying	External	✓	-

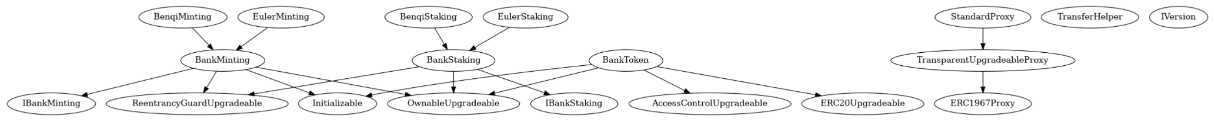
	borrowBalanceCurrent	External	✓	-
	borrowBalanceStored	External		-
	getAccountSnapshot	External		-
IComptrollerRewards	Interface			
	claimReward	External	✓	-
	rewardAccrued	External		-
	enterMarkets	External	✓	-
	exitMarket	External	✓	-
	mintAllowed	External	✓	-
	redeemAllowed	External	✓	-
	borrowAllowed	External	✓	-
	claimReward	External	✓	-
	markets	External		-
	getClaimableRewards	External		-
	rewardSupplyState	External		-
	supplyRewardSpeeds	External		-
	borrowRewardSpeeds	External		-
	rewardBorrowState	External		-
	rewardSupplierIndex	External		-
	rewardBorrowerIndex	External		-
IBankToken	Interface	IERC20Metadata		
	isMinter	External		-

	isBlacklisted	External		-
	grantMinterRole	External	✓	-
	revokeMinterRole	External	✓	-
	mint	External	✓	-
	burn	External	✓	-
	addBlacklist	External	✓	-
	removeBlacklist	External	✓	-
IBankStaking	Interface			
	mint	External	✓	-
	mintForUser	External	✓	-
	unmint	External	✓	-
	unmintForUser	External	✓	-
	deposit	External	✓	-
	withdraw	External	✓	-
	claimReward	External	✓	-
	claimRewardForUser	External	✓	-
	compoundReward	External	✓	-
	compoundRewardForUser	External	✓	-
	updatePool	External	✓	-
	pendingReward	External		-
	mintingContract	External		-
	mintToken	External		-
	bankToken	External		-
	getProtocolToken	External		-

	getWithdrawableAmount	External		-
	getUnderlyingView	External		-
	previewRedeem	External		-
	interestPerSecond	External		-
	totalStakedAmount	External		-
	totalAccruedInterest	External		-
	totalDistributedInterest	External		-
	getUnclaimedInterest	External		-
	getLogicalTotal	External		-
	userLength	External		-
	getUserInfo	External		-
	getStakingAmount	External		-
	getTotalStakePlusInterest	External		-
IBankMinting	Interface			
	mint	External	✓	-
	unmint	External	✓	-
	bankToken	External		-
	mintToken	External		-
	protocolType	External		-
	getCurrentMintedAmount	External		-
	totalMintedAmount	External		-
	totalUnMintedAmount	External		-
	hasMinterRole	External		-
	requestDelayedUnmint	External	✓	-

	getDelayedUnmints	External		-
	claimDelayedUnmintAt	External	✓	-

Inheritance Graph



Summary

Aqua Bank contract implements a token, financial and staking mechanism. This audit investigates security issues, business logic concerns and potential improvements. The team has acknowledged all the findings.

Disclaimer

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io