May 15, 2025

QuantDAO

Smart Contract Security Assessment

Prepared for
asdfjkl;
QuantitiveDAO

Prepared by
0xMilenov
Visualisa

Contents

1 About Visualisa

Visualiza is a security firm founded by 0xMilenov, a software engineer with 5 years of experience in Web3 development and competitive auditing.

After building and securing multiple blockchain projects, Visualiza was created to bring developer insight into the world of security researching. Our audits combine real-world building experience with deep technical review to help protocols avoid costly vulnerabilities.

While our main focus is smart contract audits, we also explore broader areas of security research. Learn more about us at visualisa.xyz or reach out on X.

More of our work can be found here.

2 Disclaimer

This report is based on a time-boxed security review of the provided code and related documentation. While every effort has been made to identify vulnerabilities, the findings may not cover all possible issues.

Visualiza does not guarantee the complete security of the project and takes no responsibility for any direct or indirect losses resulting from the use of this report.

3 Risk Classification

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

4 Protocol Summary

QuantDAO offers a decentralized staking platform where users can lock their QD tokens to receive sQD, a transferable token representing their share in the staking vault. Stakers earn rewards that compound over time and gain governance rights within the QuantDAO ecosystem. The platform emphasizes flexibility, allowing users to unstake at any time without penalties.

5 Audit Scope

Summary of the audit scope and any specific inclusions/exceptions.

The audit covered the following components of the QuantDAO protocol:

- QD.sol
- sQD.sol

Project Name	QuantDAO
Repository	contracts
Commit	dfe84bc7dcfe...
Audit Timeline	May 5th - May 15th
Methods	Manual Review

Critical Risk	2
High Risk	1
Medium Risk	0
Low Risk	0
Informational	2
Gas Optimizations	5
Total Issues	10

Description	Status
[C-1] Forgotten nonReentrant modifier in withdraw() function enables contract draining	Resolved
[C-2] Forgotten nonReentrant modifier in notifyRewardAmount() allows excessQD tokens to be drained	Resolved
[H-1] Bypass of anti-bot tax due to `launchBlock == 0` condition in `getCurrentTaxRate()`	Resolved
[I-1] Missing rescue function for accidentally sent tokens in sQD.sol	Resolved
[I-2] Misleading function name for appreciation ratio calculation	Resolved
[G-1] Redundant mint function implementation increases contract size and maintenance burden	Resolved
[G-2] Redundant redeem function implementation increases contract complexity	Resolved
[G-3] Redundant call to `_absorbPendingRewards()` in updateBuckets() function	Resolved
[G-4] Redundant return values in getRedeemableBalances() function	Resolved
[G-5] Redundant reward updates in notifyRewardAmount() leading to gas inefficiency	Resolved

6 Findings

6.1 Critical Risk

6.1.1 Forgotten nonReentrant modifier in withdraw() function enables contract draining

Context: sQD.sol#L216-L245

Description: The `sQD` contract implements a staking mechanism with automatic reward distribution, where users can deposit QD tokens to receive sQD shares and withdraw their assets. The `withdraw()` function is a critical component that handles asset withdrawals and updates the contract's state.

```
 function withdraw(
    uint256 assets,
    address receiver,
    address owner_
) public override whenNotPaused updateReward returns (uint256) {
    if (assets == 0) revert ZA();

    // Get maximum that can be withdrawn
    uint256 maxWithdrawable = maxWithdraw(owner_);

    // Enforce limit to prevent excessive withdrawals
    if (assets > maxWithdrawable) revert EW();

    // Calculate shares to burn (based on current exchange rate)
    uint256 shares = convertToShares(assets);

    // Make sure shares don't exceed user's balance
    if (shares > balanceOf(owner_)) {
        shares = balanceOf(owner_);
    }

    // Process withdrawal
    _withdraw(_msgSender(), receiver, owner_, assets, shares);

    // Update buckets after withdrawal
    principalQD = principalQD > assets ? principalQD - assets : 0;
    _recalculateBalances();

    return assets;
}
```

The function performs several important operations:

1. Validates withdrawal amount and limits
2. Calculates shares to burn based on current exchange rate
3. Processes the withdrawal through the parent contract's `_withdraw()`
4. **Updates the contract's principal balance and recalculates all buckets after the withdraw**

The issue lies in the lack of reentrancy protection on the `withdraw()` function. While the contract inherits from `ReentrancyGuard`, the modifier is not applied to this specific function. This is particularly concerning because the function:

- Makes external calls through `_withdraw()`
- **Updates critical state variables (`principalQD`) after the `_withdraw`**
- Interacts with the reward distribution system through `updateReward` modifier

Note

The contract already implements `ReentrancyGuard` and uses it in other functions like `sweepAll()`, indicating awareness of reentrancy risks.

Proof of Concept:

1. User deposits a large amount of QD tokens to receive sQD shares
2. User calls `withdraw()` with a valid amount
3. During the execution of `_withdraw()`, the user's contract receives the QD tokens
4. The user's contract can reenter `withdraw()` before the state updates are completed
5. This allows the user to:
    - Withdraw more assets than they should be entitled to
    - Manipulate the reward distribution system
    - Potentially drain the contract's assets

Recommendation: Add the `nonReentrant` modifier to the `withdraw()` function:

```
function withdraw(
    uint256 assets,
    address receiver,
    address owner_
)
    public
    override
    nonReentrant
    whenNotPaused
    updateReward
    returns (uint256)
{
    // ... existing code ...
}
```

6.1.2 Forgotten nonReentrant modifier in notifyRewardAmount() allows excessQD tokens to be drained

Context: sQD.sol#L298-L394

Description: The `sQD` contract implements a staking mechanism for QuantDAO tokens with automatic reward distribution. The contract uses a bucket model to track principal, pending rewards, and excess QD tokens. The `notifyRewardAmount()` function is a critical component that manages reward distribution cycles and caller incentives.

The function allows both the owner and public users (after a minimum block window) to trigger reward distribution. When called by a public user, it calculates and transfers a caller reward before updating the contract's state. This implementation violates the checks-effects-interactions pattern and lacks reentrancy protection.

```
function notifyRewardAmount() external {

// .... existing code ... //

if (callerReward > 0) {
    IERC20(asset()).safeTransfer(msg.sender, callerReward);
    // Reduce available rewards
    totalNewRewards = totalNewRewards - callerReward;

    emit CallerRewarded(msg.sender, callerReward);
}
```

The vulnerability stems from the following issues in `notifyRewardAmount()`:

1. Missing `nonReentrant` modifier
2. External call (`safeTransfer`) before state updates
3. Violation of checks-effects-interactions pattern

Note

The contract inherits from `ReentrancyGuard` but fails to utilize its protection in this critical function.

Proof of Concept:

1. Alice deploys a malicious contract that implements a fallback function to reenter `notifyRewardAmount()`
2. Alice waits for the public notification window to open
3. Alice calls `notifyRewardAmount()` from her malicious contract
4. The function calculates and transfers the caller reward to Alice's contract
5. Before state updates occur, Alice's fallback function reenters `notifyRewardAmount()`
6. The function processes the same rewards again, transferring another caller reward
7. This process can be repeated until all excess QD tokens are drained

The attack is possible because:

- The reward transfer happens before state updates
- The function lacks reentrancy protection
- The contract's state remains unchanged between reentrant calls

Recommendation:

1. Add the `nonReentrant` modifier to `notifyRewardAmount()`
2. Restructure the function to follow the checks-effects-interactions pattern:

```
function notifyRewardAmount() external nonReentrant {
```

6.2 High Risk

6.2.1 Bypass of anti-bot tax due to `launchBlock == 0` condition in `getCurrentTaxRate()`

Context: QD.sol#L171-L201

Description: The `QuantDAO` contract implements a decaying tax mechanism designed to protect against bot activity during launch. The contract uses high initial tax rates (80%) that gradually decrease to lower final rates (3%) over a specified period to deter front-running and sniping bots.

Here we can see that we use `getCurrentTaxRate` only when we have a pool and a trade.

```
if (mainPool != address(0)) {
    if (recipient == mainPool && isPool[recipient]) {
        taxRate = getCurrentTaxRate(INITIAL_SELL_TAX, FINAL_SELL_TAX);

        if (stakingContract != address(0)) {
            taxDestination = stakingContract;
        }
    } else if (sender == mainPool && isPool[sender]) {
        taxRate = getCurrentTaxRate(INITIAL_BUY_TAX, FINAL_BUY_TAX);
    }
}
```

However, there is a vulnerability in the tax calculation logic that allows bots to bypass the high initial tax rates.

The core issue is in the `getCurrentTaxRate()` function, which mistakenly returns the lower final tax rate (3%) when `launchBlock == 0`, effectively creating a window of opportunity for bots to purchase tokens at reduced tax rates before trading officially starts.

```
function getCurrentTaxRate(
    uint256 initialRate,
    uint256 finalRate
) public view returns (uint256) {
    // If trading hasn't started or if we're beyond the decay period, return the final rate
    if (
        launchBlock == 0 || block.number >= launchBlock + TAX_DECAY_BLOCKS
    ) {
        return finalRate;
    }
```

This vulnerability undermines one of the primary security mechanisms of the token. The high initial tax rate (80%) is specifically designed as an anti-bot measure during the critical pool creation and early trading phases. By allowing transactions with only a 3% tax before `launchBlock` is set, the contract exposes itself to the exact predatory trading behavior it was designed to prevent.

Recommendation: Two changes are needed to properly secure the contract against early bot purchases:

1. Remove the `launchBlock == 0` condition from `getCurrentTaxRate()` function
2. Add a trading status check in the `_transferWithTax()` function to block pool transactions until trading is officially started.

```
// In getCurrentTaxRate function, modify the condition to:
if (block.number >= launchBlock + TAX_DECAY_BLOCKS) {
    return finalRate;
}

// In _transferWithTax function, modify the pool check to:
if (launchBlock != 0 && mainPool != address(0)) {
    // Existing buy/sell logic
}
```

6.3 Informational

6.3.1 Missing rescue function for accidentally sent tokens in sQD.sol

Context: sQD.sol#L688

Description: The contract lacks a `rescueToken()` function to recover accidentally sent ERC20 tokens. This is a common pattern in staking contracts to allow the owner to rescue tokens that were sent to the contract by mistake. Without this function, any tokens sent to the contract address cannot be recovered.

The contract should implement a `rescueToken()` function that allows the owner to withdraw any ERC20 tokens that were accidentally sent to the contract. This function should:

1. Only be callable by the owner
2. Not allow withdrawal of the staking token (`QD`) or the staking receipt token (`sQD`)
3. Transfer the specified amount of tokens to the owner

Here's a recommended implementation:

```
function rescueToken(IERC20 token, uint256 amount) external onlyOwner {
    require(token != QD, "Cannot rescue staking token");
    require(token != IERC20(address(this)), "Cannot rescue staking receipt token");
    token.safeTransfer(owner(), amount);
}
```

6.3.2 Misleading function name for appreciation ratio calculation

Description: The function `getAppreciationRatio()` has a misleading name as it returns the current appreciation ratio rather than the final ratio after the reward cycle ends. This can lead to confusion for developers and users who might expect the function to return the final ratio.

The function calculates the current ratio by taking into account both the principal amount and any rewards that **would be absorbed** up to the current block. However, the name suggests it returns the final ratio after the reward cycle completes, which is not the case.

Recommendation: Rename the function to `getCurrentAppreciationRatio()` to accurately reflect that it returns the current ratio rather than the final ratio after the reward cycle ends.

6.4 Gas Optimization

6.4.1 Redundant mint function implementation increases contract size and maintenance burden

Context: sQD.sol#L194-L214

Description: The contract implements a `mint()` function that is functionally identical to the `deposit()` function. Both functions perform the same operations: checking for zero amounts, recording stake blocks, processing the deposit/mint, and updating the principal and buckets. This redundancy increases the contract size unnecessarily and creates additional maintenance overhead.

The `mint()` function is an override of the ERC4626 standard, but since it provides no additional functionality beyond `deposit()`, it should be removed to reduce contract complexity and gas costs.

Recommendation: Remove the `mint()` function and use `deposit()` exclusively for all deposit operations. If the ERC4626 standard requires the `mint()` function to be present, consider implementing it as a simple wrapper that calls `deposit()` internally.

6.4.2 Redundant redeem function implementation increases contract complexity

Context: sQD.sol#L255-L276

Description: The contract implements a `redeem()` function that is functionally identical to the `withdraw()` function. Both functions perform the same core operations: checking balances, calculating assets/shares, processing the withdrawal, and updating the principal and buckets. The only difference is that `redeem()` takes shares as input while `withdraw()` takes assets as input, but they ultimately perform the same operation.

This redundancy increases the contract size and creates additional maintenance overhead. The `redeem()` function is an override of the ERC4626 standard, but since it provides no additional functionality beyond `withdraw()`, it should be removed to reduce contract complexity and gas costs.

Recommendation: Remove the `redeem()` function and use `withdraw()` exclusively for all withdrawal operations. If the ERC4626 standard requires the `redeem()` function to be present, consider implementing it as a simple wrapper that calls `withdraw()` internally after converting shares to assets.

6.4.3 Redundant call to `_absorbPendingRewards()` in updateBuckets() function

Context: sQD.sol#L652-L655

Description: The `updateBuckets()` function makes a redundant call to `_absorbPendingRewards()` before calling `_recalculateBalances()`. This is unnecessary since `_recalculateBalances()` already calls `_absorbPendingRewards()` as its first operation. This redundancy could lead to slightly higher gas costs and potential confusion for developers maintaining the code.

Recommendation: Remove the explicit call to `_absorbPendingRewards()` in `updateBuckets()` since it's already handled within `_recalculateBalances()`:

```
function updateBuckets() external {
    _recalculateBalances();
}
```

6.4.4 Redundant return values in getRedeemableBalances() function

Context: sQD.sol#L642-L649

Description: The `getRedeemableBalances()` function returns two identical values (`qdBalance` and `totalValue`) which is redundant since they represent the same underlying value. This creates unnecessary gas overhead and potential confusion for developers integrating with the contract.

The function calculates both values using the same `convertToAssets()` call, making the second value completely redundant. This is likely a remnant from an earlier implementation where these values may have been different.

Recommendation: Simplify the function to return only one value since both represent the same underlying asset value:

```
function getRedeemableBalances(address user) external view returns (uint256) {
    uint256 userShares = balanceOf(user);
    return convertToAssets(userShares);
}
```

6.4.5 Redundant reward updates in notifyRewardAmount() leading to gas inefficiency

Context: sQD.sol#L298-L383

Description: The `notifyRewardAmount()` function contains two redundant calls that lead to unnecessary gas consumption. First, it calls `_updateReward()` which internally calls `_absorbPendingRewards()`, followed immediately by `_recalculateBalances()` which also calls `_absorbPendingRewards()`. Second, it makes a final call to `_recalculateBalances()` after all changes are complete, which is unnecessary since all state updates have already been made.

The first redundancy stems from `_updateReward()` being essentially a wrapper that calls `_absorbPendingRewards()` and updates `lastUpdateBlock`, while `_recalculateBalances()` already handles reward absorption as its first step.

The second redundancy occurs because the final `_recalculateBalances()` call will:

1. Skip `_absorbPendingRewards()` since we just absorbed all rewards and reset the period
2. Calculate `excessQD` which will be 0 as we just set it
3. Emit a `BucketsUpdated` event with values we already have

Recommenadtion:

1. Remove the `_updateReward()` call
2. Keep the first `_recalculateBalances()` call to get accurate `excessQD` before changes
3. Remove the final `_recalculateBalances()` call as it's redundant
4. Update `lastUpdateBlock` along with other block tracking variables at the end of the function

This optimization will reduce gas costs while maintaining the same functionality and accuracy of reward calculations.