AQTIS Liquid Staking Tokens

Smart Contract Security Assessment

April 04, 2024

ABSTRACT

Dedaub was commissioned to perform a security audit of **AQTIS's Liquid Staking Tokens**, specifically **qETH**, **QRT** and **QSD**. This protocol allows users to invest in a specific quant investment portfolio that would then give them rewards for holding the tokens. The diversity of the Liquid Staking Tokens available allow users to pick an investment strategy that satisfies their risk tolerance.

BACKGROUND

AQTIS has developed three Liquid Staking Tokens (LSTs): **qETH**, **QRT**, and **QSD**, each tailored to different levels of volatility tolerance amongst investors, the investments made are put towards the quant strategies developed by AQTIS, which will be returned towards investors through a rewards system utilizing AQTIS smart contracts and a uniswap V3 pool for it's new LSTs and the existing uniswap V2 pool for it's AQTIS tokens.

qETH mirrors the value of Ethereum on a 1:1 basis, offering investors a direct exposure to Ethereum's market movements coupled with a yield generated from the broader AQTIS ecosystem. It provides a 10% annual yield, divided into 7.5% from Ethereum and 2.5% from AQTIS tokens, catering to those seeking exposure to Ethereum's market with an additional yield component.

QSD, standing for Quant State Dollar, is characterized by its fixed annual return rate (APR) of 15%, despite a dynamic pricing mechanism. It yields a specific $0.15 USD per token, with returns made up of 12.5% in USDC and 2.5% in AQTIS tokens.

QRT, or Quant Reserve Token, features a yield that adjusts based on the token's market demand, offering a combined annual yield of 17.5%—allocated across ETH, USDC, and AQTIS tokens.

SETTING & CAVEATS

This audit report mainly covers the contracts of the **at-the-time private** repository https://github.com/AQTISOfficial/aqtis-smart-contracts/ of the Protocol at commit `9397cc67c3df80fe6f19a6ec6e91e255c0cf50c1`.

2 auditors worked on the codebase for 4 days on the following contracts:

```
contracts/
├── Compose.sol
├── lst/
│   ├── AbstractLST.sol
│   ├── Qeth.sol
│   ├── Qrt.sol
│   ├── Qsd.sol
│   └── rewards/
│       ├── AbstractRewards.sol
│       ├── QethRewards.sol
│       ├── QrtRewards.sol
│       └── QsdRewards.sol
├── prices/
│   ├── TokenPriceCalculator.sol
│   └── UniswapV2PriceOracle.sol
├── rewards/
│   ├── BatchClaim.sol
│   ├── ClaimVault.sol
│   └── MintingBonus.sol
├── token/
│   └── PseudoCappedERC20.sol
└── trade/
    └── LSTSwap.sol
```

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often

trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

PROTOCOL-LEVEL CONSIDERATIONS

ID	Description	STATUS
P1	The protocol depends on external uniswap pools which are promised to be maintained by the AQTIS team.	**INFO**
The protocol doesn't provide on-chain guarantees that rewards will be available for a set amount of time, and neither withdrawals other than the promise that the pool will be watched and kept at a reasonable state (the component responsible for this is the Ecosystem Liquidity aggregator, which is not part of this audit). In the case of a liquidation event the last users to exit will very likely lose funds due to the nature of uniswap pools.		
P2	The uniswap v2 oracle and daily claim functionality depends on the AQTIS team to call the contracts daily.	**INFO**
It is unclear at this point in time what mechanism the team will be utilizing to perform these actions daily, however it is important that these mechanisms are reliable and predictable as if these actions aren't performed, users will lose out on rewards (Although they might be negligible if only a single day is missed) and users might have to deal with a stale price of AQTIS.		

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

ID	Description	STATUS
C1	The reward accounting of LST tokens is manipulable	**RESOLVED**

On top of the typical ERC20 balance accounting, LST transfer and minting operations involve some protocol-introduced reward accounting. This is implemented in `AbstractRewards::_updateRecord` which is invoked inside each LST's overridden `ERC20::_update` function:

`AbstractRewards::_updateRecord`

```
function _updateRecord(address user, uint256 value, Update updateType) internal {
        _beforeUpdate(user, value, updateType);
        _updateSupply(user, value, updateType);

        if (user == address(0) || isContract(user)) {
        return;
        }

        // If user is not in the list, add them
        if (!_users.contains(user)) {
        _users.add(user);
        }

        UserRecord storage record = _userRecords[user];
        uint256 timeElapsed = 0;

        // First entry check
        if (record.lastUpdateTime == 0) {
        record.cumulativeBalance = 0;
        record.cumCirculatingSupplyLastClaim = _supply.cumulativeCirculatingSupply;
        record.lastClaimTime = block.timestamp;
        } else {
        timeElapsed = block.timestamp - record.lastUpdateTime;
```

```
        record.cumulativeBalance += record.userBalance * timeElapsed;
        }

        // Update balance and last update time
        if (updateType == Update.FROM) {
        record.userBalance -= value;
        } else {
        record.userBalance += value;
        }
        record.lastUpdateTime = block.timestamp;
}
```

The protocol skips all reward accounting for users that are considered to be smart contracts (i.e., addresses for which `isContract` returns true)

`AbstractRewards::isContract`

```
function isContract(address addr) internal view returns (bool) {
        if (whitelistedContracts[addr]) {
                return false;
        }
        uint size;
        assembly {
                size := extcodesize(addr)
        }
        return size > 0;
}
```

Although this is done with the clear intent of preventing liquidity pools (or other non-whitelisted 3rd party contracts) from accruing rewards , `isContract(X)` is false when:

- X has not been deployed, this means that interacting with an about-to-be deployed address will cause rewarding accounting to take place as normal.
- X is under construction, this enables one to run code from inside a contract's constructor and having reward accounting enabled for that contract momentarily.

One can use the first point from above to set the reward accounting of about-to-be deployed contract arbitrarily high:

1. Assume that a contract `X` receives a huge LST flashloan of `N` tokens from the corresponding Balancer/Uniswap pool
2. `X` transfers all borrowed funds to an address `Y` that will be soon deployed at a pre-computed address via `CREATE2`. At this point, `AbstractRewards::_updateRecord` will see that `isContract(Y)` is false, so it will happily create a `_userRecord` for `Y` during the large transfer of tokens
3. `Y` is deployed via `CREATE2` to the predicted address
4. Assume that `Y` has a function that simply transfers all LST funds back to `X`, named `Y::refund()` . Here's where everything comes together; when we invoke `Y::refund()` the transfer method of the LST will not touch the `_userRecord` entry of `Y`, since `Y` is now initialized and `isContract(Y)` will return true. Token balances will be updated as they should, so `X` will receive back the funds and repay the flash loan with no problems.

We have now successfully polluted the `_userRecord` of Y with a completely inflated balance (even though `Y` holds no tokens). This can have devastating consequences: `AbstractRewards::_getTWAB(Y)` will return `N` (the originally flash-loaned amount) when invoked at a subsequent block than the one that `Y` got created.

At this point, we can use this to directly attack `ClaimVault` and claim rewards as if `Y` had `N` tokens, since rewards will be claimed successfully. The above attack can be atomically set up with many `Y`-like contracts, and thus we can claim rewards multiple times from the `ClaimVault` contract and drain its entire balance.

Flash Loans are not necessary, an LST holder can also pull the attack on its own via direct transfers. However, flash loans make the attack even more practical, since no LST balance is required at all to pull this out.

We additionally note another vector of reward accounting manipulation, although with a lower severity than the manipulation described above. This time we are targeting the accounting inside `AbstractRewards::_updateSupply`:

`AbstractRewards::_updateSupply:78`

```
if (updateType == Update.FROM && (user == address(0) || isContract(user))) {
            _supply.currentCirculatingSupply += value;
      } else if (updateType == Update.TO && (user == address(0) ||
isContract(user))) {
            _supply.currentCirculatingSupply -= value;
}
```

1. Assume that an attacker transfers `N` LST tokens to a pre-calculated address and then deploys a contract `X` on it (`_supply.currentCirculatingSupply` has not changed since `isContract(X)` was false when the transfer took place)
2. Assume that `X` transfers the LST tokens at another pre-calculated address. Since `isContract(X)` is now true, `_supply.currentCirculatingSupply` is now increased by `N` .
3. After the transfer, `X` uses `CREATE2` to deploy a copy of itself at the pre-computed address `Y` of the previous step
4. Step 2 is repeated many times

This artificially inflates the `_supply.currentCirculatingSupply`. `X` may be coded in a way so that funds are returned to the attacker after a finite number of steps so the attacker ultimately manipulates the reward accounting for free. This manipulation is not by itself profitable for the attacker, but it could cause `QrtReward::getRewardsFor` and `MintingBonus::_claimableBonusRewards` to yield lower rewards for the rest of the protocol users.

In conclusion, even though the design decision of disabling reward distribution for smart contracts is a fine decision on its own, it currently poses a security liability. It is recommended that this part of the protocol is re-implemented and/or re-designed.

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	`ClaimVault::_claim` may unexpectedly revert, raising DOS concerns	**RESOLVED**

If `maxClaim` is larger than `remainingRewards` in `MintingBonus`, `claimVault::_claim` will keep reverting till the bonus program finishes, since `remainingRewards` will underflow and revert due to Solidity's 0.8 integration of safeMath resulting in a DoS.

`MintingBonus::_claimableBonusRewards:L57`

```
// @audit this should be the other way round cause if remaining < maxClaim, then
it fails
if (bonusAmount > rewardsSettings[lst].maxClaim) {
     bonusAmount = rewardsSettings[lst].maxClaim;
} else if (bonusAmount > rewardsSettings[lst].remainingRewards) {
     bonusAmount = rewardsSettings[lst].remainingRewards;
}
```

LOW SEVERITY:

ID	Description	STATUS
L1	The chainlink oracles aren't checked for staleness in `TokenPriceCalculator`	**RESOLVED**

In order to prevent the protocol from consuming stale prices in the event of a price feed's downtime, answers coming from chainlink data feeds should be checked for staleness. Reference

`TokenPriceCalculator::getLatestEthPrice:L56`

```
/// @notice Returns ETH Price in USD (8 decimals)
function getLatestEthPrice() public view returns (int) {
    (,int price,,,) = priceFeedEth.latestRoundData();
    return price;
}

// @audit Returns USDC price in USD (8 decimals) *FIX
/// @notice Returns USD Price in USD (8 decimals)
function getLatestUsdPrice() public view returns (int) {
    (,int price,,,) = priceFeedUsd.latestRoundData();
    return price;
}
```

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be

considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	The owner of the contracts is considered trusted	**OPEN**
The protocol has some centralization risks, with some owner entities considered trusted. There are numerous instances in which the owner of a contract is able to change the core configuration of the contracts (e.g., `ClaimVault::setUsdcPair`) or even pull funds out of the contract (e.g., `ClaimVault::withdrawERC20`)		

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	`lastBuyTime` and `DailyMintTokens` are unused in `AbstractLST`	**RESOLVED**

`AbstractLST::L29,L36`

```
mapping(address => uint256) public lastBuyTime; // @audit isn't really
used anywhere. Might not be worth the gas cost, since an event is emitted
as well.

event DailyMintTokens(uint256 amount, uint256 newTotalSupply); // @audit
unused
```

A2	`buyActive` and `whiteListActive` are unnecessarily set to false in the constructor	**RESOLVED**

`AbstractLST::constructor:L47`

```
constructor(
        string memory _name,
        string memory _symbol,
        uint256 _totalMaxSupply
    ) PseudoCappedERC20(_name, _symbol, _totalMaxSupply) Ownable(msg.sender) {
        // set defaults
        // @audit this is a waste of gas since booleans are false by default, Also
might want to
        // consider setting the distribution address in the constructor
        buyActive = false;
        whitelistActive = false;
    }
```

A3	`updateWhitelist` might be more suitable as a function that runs in batches	**ACKNOWLEDGED**

`AbstractLST::updateWhitelist:L83`

```
// @audit it might be ideal to actually have a function that sets the whitelist in
bulk
function updateWhitelist(address _addr, bool _whitelisted) external onlyOwner {
    whitelist[_addr] = _whitelisted;
    emit WhitelistUpdated(_addr, _whitelisted);
}
```

A4	Use of events is inconsistent throughout the protocol	**RESOLVED**

Although `AbstractLST` emits a wealth of events, there are no events emitted for any changes in the `claimVault` or any bonus rewards that are set. The events should be reconsidered protocol wide.

A5	`_users` is unused in `AbstractRewards`	**RESOLVED**

Although written to, `_users` is never read from in the protocol, and it isn't exposed publicly either since it's internal.

`AbstractRewards:L51`

```
EnumerableSet.AddressSet internal _users; // @audit this variable isn't read anywhere and not exposed publicly.
```

A6	Redundant check in `AbstractRewards` for `timeDifference`	**RESOLVED**

The check is redundant since if timeDifference == 0,
`(record.cumulativeBalance + (record.userBalance * timeDifference)) / claimTime`
==
`(record.cumulativeBalance + (record.userBalance * 0)) / claimTime`
==
`record.cumulativeBalance / claimTime`
`AbstractRewards::_getTWAB:L145`

```
if (timeDifference > 0) {
    return (record.cumulativeBalance + (record.userBalance * timeDifference)) / claimTime;
} else {
    return record.cumulativeBalance / claimTime;
}
```

A7	`claimCooldown` being 1 day might be problematic if batch claim deviates a bit.	**RESOLVED**

This would result in certain users/all users subscribed to the batch daily claim to potentially getting their rewards a day late. Suggested fix is to lower this to half a day.

`ClaimVault::_claim:L67`

```
// @audit is there actually a reason for the claimCooldown to be 1 day? Since this
could potentially end up reverting for the
// batch claimer, if it deviates a bit from the exact 1 day mark.
require(lastClaimTime[user] + claimCooldown < block.timestamp, "ClaimVault:
Cooldown not expired");
```

A8	Incorrect comment in `TokenPriceCalculator`	**RESOLVED**

`TokenPriceCalculator::getLatestUsdPrice:L62`

```
// @audit Returns USDC price in USD (8 decimals) *FIX
/// @notice Returns USD Price in USD (8 decimals)
function getLatestUsdPrice() public view returns (int) {
    (,int price,,,) = priceFeedUsd.latestRoundData();
    return price;
}
```

A9	Unused import `CallbackValidation` in `LSTSwap`	**RESOLVED**

`TokenPriceCalculator::getLatestUsdPrice:L62`

```
import {CallbackValidation} from
"../external/uniswap/CallbackValidation.sol"; // @audit unused
```

A10	Test coverage	**INFO**

Although the project already has quite good coverage of tests it does not achieve 100% test coverage. It is recommended that before launching a further investment of testing aiming to cover BatchClaim, ClaimVault and MintingBonus especially is done. Fuzz tests are also recommended as they could catch further edge case logic issues.

A11	Unused struct member	**RESOLVED**

The `amountOutMin` member of the `LSTSwap::SwapCallbackData` is not used anywhere

LSTSwap:42

```
struct SwapCallbackData {
        address tokenIn;
        address tokenOut;
        uint256 amountOutMin;
        uint256 amountInMax;
}
```

A12	Potentially unbounded iteration	**ACKNOWLEDGED**

Since anybody may enable "auto-claiming" of rewards, `BatchClaim::BatchClaim` might cause the execution to run out of gas when `_enabledUsers` become large enough

BatchClaim:63

```
function batchClaim() external onlyScheduler {
        address[] memory lsts = claimVault.getLSTs();

        for (uint256 i = 0; i < _enabledUsers.length(); i++) {
                address user = _enabledUsers.at(i);
                for (uint256 j = 0; j < lsts.length; j++) {
                claimRewardsFor(lsts[j], user);
                }
        }
}
```

This might not pose a practical security issue, since the scheduler will be able to claim rewards for a subset of users at a time by calling `BatchClaim::multiClaim` or `BatchClaim::multiClaimLST`.

A13	Unused internal functions	**RESOLVED**

Both `LSTSwap::_setTimeWeightedAveragePeriod` and `LSTSwap::_setMinOutFractionQ64` are not used anywhere inside the codebase

LSTSwap:215

```
function _setTimeWeightedAveragePeriod(uint24 period) internal {
	timeWeightedAveragePeriod = period;
}

function _setMinOutFractionQ64(uint256 fraction) internal {
	minOutFractionQ64 = fraction;
}
```

A14	Superfluous check	RESOLVED

`Compose::minAmountBuy` can never be set to 0:

Compose:35

```
function distributeFunds(
	uint256 _percentageQeth,
	uint256 _percentageQsd,
	uint256 _percentageQrt
) external payable nonReentrant {
	require(msg.value > 0, "No ETH sent");
	...
	require(
		msg.value >= minAmountBuy,
		"Amount sent is less than minimum amount"
	);
```

So the first check involving `msg.value` inside `Compose::distributeFunds` is unnecessary, since the one that follows is logically stronger:

Compose:64

```
function distributeFunds(
	uint256 _percentageQeth,
	uint256 _percentageQsd,
	uint256 _percentageQrt
) external payable nonReentrant {
	require(msg.value > 0, "No ETH sent");
	...
```

```
        require(
                msg.value >= minAmountBuy,
                "Amount sent is less than minimum amount"
        );
```

asd

A15	Unused argument	RESOLVED

The `payer` argument of `LSTSwap::pay` is unused:

`LSTSwap:196`

```
function pay(address token, address payer, address receiver, uint256 amount)
private {
        ILST(token).mint(amount);
        TransferHelper.safeTransfer(token, receiver, amount);
}
```

A16	Compiler bugs	INFO

The code is compiled with Solidity `0.8.23`, which has no known issues at the time of this report.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Security Suite.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.