QuillAudits

AUDIT REPORT

March, 2025

For

Spheron

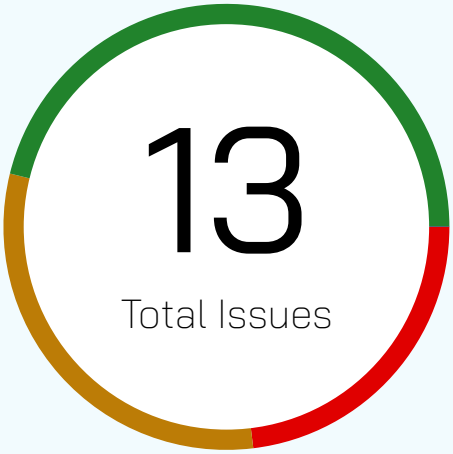
Table of Content

Executive Summary

Project name	spheronFdn
Overview	This is a lending pool for miners, enabling users to deposit mining tokens in exchange for pTokens. Implements borrowing and repayment mechanisms for agents, with interest accrual, liquidation processes, and distribution of rewards to token holders.
Project URL	https://www.spheron.network/
Audit Scope	The scope of this Audit was to analyze the spheronFdn Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/spheronFdn/airdrop-staking-contracts
Contracts in Scope	https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/AUDIT.md
Commit Hash	64445ea61904d771678f4fee394a36f382bdaef8
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	7th March 2025 - 16th March 2025
Updated Code Received	26th March 2025
Review 2	26th March 2025

Fixed In	https://github.com/spheronFdn/airdrop-staking-contracts/pull/3

Number of Issues per Severity



High	3 (23.08%)
Medium	4 (30.77%)
Low	6 (46.15%)
Informational	0 (0.00%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**3**	**4**	**3**	0
Acknowledged	0	0	**3**	0
Partially Resolved	0	0	0	0

Checked Vulnerabilities

- ☑ Access Management
- ☑ Arbitrary write to storage
- ☑ Centralization of control
- ☑ Ether theft
- ☑ Improper or missing events
- ☑ Logical issues and flaws
- ☑ Arithmetic Computations Correctness
- ☑ Race conditions/front running
- ☑ SWC Registry
- ☑ Re-entrancy
- ☑ Timestamp Dependence
- ☑ Gas Limit and Loops
- ☑ Exception Disorder
- ☑ Gasless Send
- ☑ Use of tx.origin
- ☑ Malicious libraries
- ☑ Compiler version not fixed
- ☑ Address hardcoded
- ☑ Divide before multiply
- ☑ Integer overflow/underflow
- ☑ ERC's conformance
- ☑ Dangerous strict equalities
- ☑ Tautology or contradiction
- ☑ Return values of low-level calls
- ☑ Missing Zero Address Validation
- ☑ Private modifier
- ☑ Revert/require functions
- ☑ Multiple Sends
- ☑ Using suicide
- ☑ Using delegatecall
- ☑ Upgradeable safety
- ☑ Using throw

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools And Platforms Used For Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

Security vulnerabilities identified that must be resolved and are currently unresolved.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/ impact of the security issue, but are not completely resolved.

High Severity Issues

Liquidation doesn't accrue interest

Resolved

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/core/Delegation-Pool.sol#L312-L321

Function

startLiquidation()

Description

accureInterest() should be called whenever totalDebt is being used because it updates the totalDebt. Now debtOf(agent) uses totalDebt to calculate the debtOf(agent), which means totalDebt should be updated before calling debtOf(agent) in startLiquidation(). ie missing accureIntest()

```solidity
 function startLiquidation(address agent) external nonReentrant whenNotPaused {
//@audit this should also call accureIntest()
 if (msg.sender != core) revert CallerNotCore();

@>  uint256 liquidatingAmount = debtOf(agent);
 removeDebt(agent, liquidatingAmount);
 totalLiquidatingAmount += liquidatingAmount;
 liquidatingAmounts[agent] += liquidatingAmount;
 invariantCheck();
 emit LiquidationStarted(agent, liquidatingAmount);
}
```

Wrong hashing implementation allows user to execute EXPIRED signature

Resolved

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/core/Delegation-Pool.sol#L312-L321

Function

startLiquidation()

Description

In the bootstrapOffchainCollateral(), you'll notice it checks if (block.timestamp > deadline) revert ExpiredSignature(deadline); for expired signature however also note this deadline is a user argument, the issue here is that the deadline was not included when generating the hash to verify, so even if the signature signed by the core signers is expired, user can just pass in a future time which will always pass since the deadline won't get validated.

Recommendation

Just as in updateOffChainCollateralValueBySignature(), deadline should be included in the hash to be verified

```
bytes32 hash = _hashTypedDataV4(
 keccak256(
 abi.encode(
 keccak256(
"UpdateOffChainCollateralValue(address agent,uint256 value,uint256 deadline,uint256 nonce)"
),
 agent,
 value,
 deadline, <@
 nonce
)
)
);
```

Signature replay protection bypassed also due to wrong hashing implementation

Resolved

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/core/Delegation-Pool.sol#L312-L321

Function

startLiquidation()

Description

In verifySignature, the check if (nonce != core.signatureNonces(agent)) revert InvalidNonce(); was to help prevent replays as the purpose of nonce logic is to prevent signature replay however this nonce is also a user argument, what that means is user specifies this value and not the protocol. The issue is the nonce specified by the user is not included when generating the hash to be verified allowing user to specify a different nonces but on the same signature.

Recommendation

Nonce should be included in the hash to be verified:

```
bytes32 hash = _hashTypedDataV4(
 keccak256(
 abi.encode(
 keccak256(
"BootstrapOffchainCollateral(address agent,uint256 securityDepositAmount,uint256 borrowingAmount,bytes deployData)"
),
 agent,
 securityDepositAmount,
 borrowingAmount,
 deployData
) // nonce should be included here
)
);
```

Medium Severity Issues

Honest users can be griefed in withdrawRequest.

Resolved

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/ThawingPool.sol#L70-L94

Function

withdrawRequest , withdraw

Description

User can initiate withdrawRequest on behalf of other user by passing _to, and this will increase the unlockTime of withdrawRequest unlockTime is checked in withdraw().

```solidity
 function withdrawRequest(uint256 _amount, address _to) external whenNotPaused {
 require(_to != address(0), "Invalid recipient");
 require(_amount > 0, "Invalid amount");
 WithdrawRequest storage request = withdrawRequests[_to];
 pToken.transferFrom(msg.sender, address(this), _amount);
 uint256 shares = pToken.convertToShares(_amount);
 request.shares += shares;
//@audit this can be used to grief other users by initiating a withdrawRequest and below line will increase the unlockTime by vestingTime
@>    request.unlockTime = block.timestamp + vestingTime;
 emit WithdrawRequested(_to, shares);
}
function withdraw() external whenNotPaused {
 address _to = msg.sender;
 WithdrawRequest storage request = withdrawRequests[_to];
 require(request.shares > 0, "No withdraw request found");
@>   require(block.timestamp >= request.unlockTime, "Vesting time not reached");
 uint256 amount = pToken.convertToAssets(request.shares);
 delete withdrawRequests[_to];
 pToken.approve(address(delegationPool), amount);
 delegationPool.withdraw(amount, _to);
 emit Withdraw(_to, amount);
}
```

Withdraw doesn't enforce slippage protection

Resolved

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/Thawing-Pool.sol#L70-L94

Function

withdrawRequest , withdraw

Description

There should be slippage protection for users otherwise while withdrawing they could lose significant amount if exchange rate is high/low

```solidity
 function withdrawRequest(uint256 _amount, address _to) external whenNotPaused {
 require(_to != address(0), "Invalid recipient");
 require(_amount > 0, "Invalid amount");
 WithdrawRequest storage request = withdrawRequests[_to];
 pToken.transferFrom(msg.sender, address(this), _amount);
@>  uint256 shares = pToken.convertToShares(_amount);
 request.shares += shares;
 request.unlockTime = block.timestamp + vestingTime;
 emit WithdrawRequested(_to, shares);
}

 function withdraw() external whenNotPaused {
 address _to = msg.sender;
 WithdrawRequest storage request = withdrawRequests[_to];
 require(request.shares > 0, "No withdraw request found");
 require(block.timestamp >= request.unlockTime, "Vesting time not reached");
 uint256 amount = pToken.convertToAssets(request.shares);
 delete withdrawRequests[_to];
 pToken.approve(address(delegationPool), amount);
 delegationPool.withdraw(amount, _to);
 emit Withdraw(_to, amount);
}
```

UpdateFeeRate doesn’t accrue old fee before update

Resolved

Description

DelegationPool.sol.updateFee() doesn't accrue old fee before updating it to the new one

```solidity
 function updateFeeRate(uint256 _newFeeRate) external onlyOwner {
 feeRate = _newFeeRate;
 emit FeeRateUpdated(_newFeeRate);
}
```

as the updateFeeRate function doesn't accrue fees before the update, there will be problem in distributeLiquidationSurplus and accrueInterest functions as both distribute pTokens w.r.t fee. Although distributeLiquidationSurplus won't be affected much since it is only called when liquidation is called but accrued interest will be affected since it is dependent on the fee.

```solidity
 function distributeLiquidationSurplus(uint256 amount) internal {
 uint256 fee = (amount * feeRate) / 10_000;
 pToken.mint(feeReceiver, fee);
 pToken.distribute(amount - fee);
}

 function accrueInterest() public {
 uint256 timeElapsed = block.timestamp - lastAccrueTimeStamp;
 if (timeElapsed == 0) return;
 uint256 totalInterestIncrement = (
 interestRateModel.getBorrowRate(pToken.totalSupply(), totalDebt + totalLiquidatingAmount) *
totalDebt  * timeElapsed ) / SECONDS_PER_YEAR / 1e18;
 totalDebt += totalInterestIncrement;
 uint256 fee = (totalInterestIncrement * feeRate) / 10_000;
 lastAccrueTimeStamp = block.timestamp;
 pToken.mint(feeReceiver, fee);
 pToken.distribute(totalInterestIncrement - fee);//@audit if fee is increased then less pTokens will
be distributed here
 invariantCheck();
}
```

The correct way would be to collect the fee first and then update the fee, or call the accrueInterest() before updating the fee.

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/core/Delegation-Pool.sol#L168-L171
https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/core/Delegation-Pool.sol#L379-L404

Function

updateFeeRate , distributeLiquidationSurplus , accrueInterest

DOS due to check implemented in wrong function causes changeAgentOwner to always revert

Resolved

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/Thawing-Pool.sol#L70-L94

Function

withdrawRequest , withdraw

Description

The changeAgentOwner() function allows owner to change it's agent owner, it has the isAgentCall modifier what this means is that only the Agent can call this function and not the owner, also meaning the owner would have to call this through the agent contract which is the intended design. However, an issue arises in the corelogic.changeAgentOwner, notice the check if (agentDetail.owner != msg.sender) revert CallerNotAgentOwner(); this means caller must be the owner itself and not the agent, this is a conflict check to the isAgentCall modifier causes the entire function to always revert.

Recommendation

remove if (agentDetail.owner != msg.sender) revert CallerNotAgentOwner(); from corel-ogic.changeAgentOwner()

Low Severity Issues

updateCollateralConfig can trigger sudden liquidations.

Acknowledged

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/core/Core.sol#L361

Function

updateCollateralConfig()

Description

```solidity
function updateCollateralConfig(CollateralConfig memory _collateralConfig) external onlyOwner {
_updateCollateralConfig(_collateralConfig);
}
 function _updateCollateralConfig(CollateralConfig memory _collateralConfig) internal {
 if (!CoreLogic.validateCollateralConfig(_collateralConfig, PARAM_PRECISION)) revert InvalidCollateralConfig();
 slashingBuffer = _collateralConfig.slashingBuffer;
 isSlashingBufferAbsoluteValue = _collateralConfig.isSlashingBufferAbsoluteValue;
 liquidationThreshold = _collateralConfig.liquidationThreshold;
 collateralFactor = _collateralConfig.collateralFactor;
 emit CollateralConfigUpdated(_collateralConfig);
}
```

The above function. is called by owner to update the collateral config, It updates certain important states including liquidationThreshold , the problem is when liquidationThreshold is updated it can trigger sudden liquidation for healthy positions too

If we take a look at coreLogic.startLiquidation() it has a check and when liquidationThreshold is changed the check can be bypassed suddenly and healthy positions can become unhealthy and become prone to sudden liquidation.

```solidity
 if (fullRepayTokenAmount < (collateralValue * liquidationThreshold) / paramPrecision) {
 revert CannotLiquidateHealthyAgent();
}
```

Hardcoded claimDeadLine will lead to loss of vesting assets for users

Resolved

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/AirdropPool.sol#L61

Function

claimDeadline

Description

In AirdropPool.sol the constructor sets the claimDeadline as claimDeadline = _launchTime + (8 * 30 days); which is hardcoded as 240 days from the launchTime, However the function claimVestingAssets has a check :

```
require(block.timestamp <= claimDeadline, "Claim period has ended");
```

Which means the claim can only be made when the claimDeadline is not passed but the claim deadline is hardcoded to 240 days in the constructor, and for the periods of more than 240 days vesting schedule users won't be able to claim vesting assets via claimVestingAssets function.

```solidity
function claimVestingAssets() external whenNotPaused {
 UserInfo storage user = claimInfo[msg.sender];
 require(user.claimed, "Not claimed");
 require(block.timestamp <= claimDeadline, "Claim period has ended");
 require(user.totalShares > user.claimedShares, "No shares to claim");
 uint256 claimableShares = _claimAfterLockTime();
 user.lastClaimTime = block.timestamp;
 pToken.transfer(msg.sender, pToken.convertToAssets(claimableShares));
 emit AirdropClaimed(msg.sender, pToken.convertToAssets()
```

weth address is hardcoded

Acknowledged

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/core/utils/Token-Wrapper.sol#L24

Function

weth

Description

IWETH constant weth = IWETH(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2); // need to manually update for each chain//@audit (L) hardcoded address for weth, WETH has diff address on diff chains

Wrong event emitted in the repay function

Resolved

Description

The repay() function in delegation pool emits wrong event

```diff
- emit Repaid(agent, payingAmount, payingAmount);
+ emit Repaid(agent, tokenAmount, payingAmount);
```

Recommendation

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/core/Delegation-Pool.sol#L303

Teams Comment

repay

claimPenaltyAssets lacks onlyOwner modifier

Acknowledged

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/Airdrop-Pool.sol#L108-L114

Function

claimPenaltyAssets()

Description

The above function should only be called by owner but it lacks modifier onlyOwner so anyone can initiate claiming of penaltyAssets, although the assets are being transferred to the owner so this is a low issue. And emit PenaltyAssetsDistributed(msg.sender, penaltyAssets); event is emitted with msg.sender which is wrong if anyone else apart from the owner calls this function.

```solidity
function claimPenaltyAssets() external {
 require(penaltyShares > 0, "No penalty shares");
 uint256 penaltyAssets = pToken.convertToAssets(penaltyShares);
 penaltyShares = 0;
 pToken.transfer(owner(), penaltyAssets);
 emit PenaltyAssetsDistributed(msg.sender, penaltyAssets);
}
```

Wrong getAgentAddress() check in registerMiner()

Resolved

Path

https://github.com/spheronFdn/airdrop-staking-contracts/blob/main/src/Airdrop-Pool.sol#L108-L114

Function

claimPenaltyAssets()

Description

The check if (getAgentAddress(minerId, _msgSender()) != address(0)) revert AgentAlreadyCreated(); in registerMiner() is implemented wrongly, notice getAgentAddress(minerId, _msgSender()), it passes minerId first then _msgSender() , However this is the getAgentAddress() function:
function getAgentAddress(address _owner, address _minerId) public view returns (address) { <@ note owner first then minerId
return minerAgentsByOwner[_owner][_minerId];
}
As we can see the parameters where passed in wrongly leading to wrong value returned

Recommendation

diff
- if (getAgentAddress(minerId, _msgSender()) != address(0)) revert AgentAlreadyCreated();
+ if (getAgentAddress(_msgSender(), minerId) != address(0)) revert AgentAlreadyCreated();

Functional Tests

Some of the tests performed are mentioned below:

- AccrueInterest should be called when needed
- Withdraw can be dossed
- Old fee not collected when fee is updated
- Repayment without paying interest.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of SpheronFdn. We performed our audit according to the procedure described above.

Issues of High, medium and low severity were found.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

March, 2025

For

Spheron

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com