

Audit Report for

Whirl Finance

January 2024

Version 1.2

Email	Telephone
audits@extropy.io	+44 1865338228

Scope

The code is taken from repo <https://github.com/WhirlProtocol/whirl>

The initial audit was at commit 245bb14489221a84087098b12ef2c1b361676268

The final audit was of branch refactorDataFeed at commit
096c52eec86841acdba7ef31b566e0a1e7852f27

Contracts in scope

This only only looked at the contracts in the mining directory.

MiningPoolManager.sol

RandomStrike.sol

QuantumRNG.sol

RefManager.sol

Whirl.sol

WhirlMine.sol

WhirlMineHelpers.sol

WhirlTreasury.sol

byteCalc.js

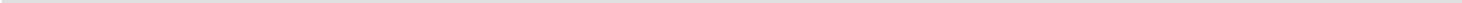
Using This Report

To facilitate the dissemination of the information within this report throughout your organisation, this document has been divided into the following clearly marked and separable sections.

- Executive Summary
- Management level, strategic overview of the assessment and the risks posed to the business
- Technical Summary
- An overview of the assessment from a more technical perspective, including a defined scope and any caveats which may apply
- Technical Findings
- Detailed discussion (including evidence and recommendations) for each individual security issue which was identified
- Audit process and tools used

Document Version Control

Data Classification	Client Confidential
Client Name	Whirl
Document Title	Whirl Audit
Author	Extropy Audit Team



Executive Summary

Extropy was contracted to conduct an initial code review and vulnerability assessment of part of the project, the scope being the contracts in the mining directory only.

The initial assessment was carried out between 4th and 11th December 2023.

After significant refactoring, the code was reassessed between 9th and 16th January 2024.

Assessment Summary

The most important issues found in the initial report, those relating to randomness and the possibility of withdrawal of funds, have been resolved by re factoring. Random values are now provided by an off chain oracle. Funds previously held in SupporterPool are planned to be held in a wallet. All other issues have been resolved as described in the following section.

Issue Count

Stage	Critical	Medium	Low	Informational
Initial Report	0	5	7	8
Final Report	0	0	0	0

Technical Findings

The remainder of this document is technical in nature and provides additional detail about the items already discussed, for the purposes of remediation and risk assessment.

Medium Risk

M1. Access Control in clearExcess function

Risk	Medium
Affects	WhirlTreasury.sol

The clearExcess function can currently be called by any user, which might lead to its inappropriate or malicious triggering. This open access could be exploited by an attacker aware of a pending large deposit or aware of market conditions favourable to triggering this function for personal gain. Unauthorised triggering of clearExcess could lead to suboptimal conversion rates for the treasury, resulting in financial losses or undesired market impacts. In worst-case scenarios, it could also facilitate manipulative schemes, especially when coupled with other vulnerabilities like price manipulation in the liquidity pool.

Recommendation

- Restrict access to clearExcess by implementing an access control mechanism, such as allowing only the contract owner or a designated governance body to trigger it.
- Introduce multi-signature requirements for added security, where more than one authorised account must approve the execution of clearExcess.
- Implement a timelock mechanism to provide a buffer period during which stakeholders can review and react to the pending execution of clearExcess.

Outcome

Resolved as recommended

M2. Underflow possible in function sqrt(uint256 x)

Risk	Medium
Affects	WhirlTreasury.sol

Calling this function with certain values causes an underflow and panic.

- e.g input = 766920509, expected = 27693, actual = 27693329684239

This would also impact the following functions

```
function calculateRealignment() internal -> unsafe method it uses unsafe sqrt()
function
function realignWhirlPrice() -> _ethToSpend references unsafe method
calculateRealignment()
function executeSwap() no handling of underflow panics for uint256 totalBalance
``
```

Recommendation

- Adjusting by the requisite number of decimal places (9)
- Use a Maths library such as <https://github.com/PaulRBerg/prb-math>

Outcome

Resolved with the use of the recommended library.

M3. Unchecked return value

Risk	Medium
Affects	SupporterPool.sol

There is an unchecked return value Line 106. This would allow the later updates to proceed even if the external call failed.

Impact :

This could leave the contract in an indeterminate state.

Recommendation

Check the return value of the call and revert if necessary.

Outcome

Resolved as recommended

Low Risk

L1. Potential Issues with `_writeCheckpoint` Function

Risk	Low
Affects	Whirl.sol

The `_writeCheckpoint` function assumes chronological writing of checkpoints, potentially causing discrepancies if a checkpoint is missed.

The contract uses a checkpoint system to record vote balances at specific block numbers. This is a common approach in governance tokens to track voting power changes over time.

The implementation assumes that checkpoints are written in a chronological order (based on block numbers). This is evident from the logic in `_writeCheckpoint`, which either updates the last checkpoint or adds a new one based on the current block number.

The contract does not explicitly handle scenarios where a checkpoint might be missed. This means that if there is a significant gap between actions that change a user's voting power, some intermediate states might not be checkpointed.

For governance decisions, this means the contract relies on the latest checkpointed state if a block number is specified for which there is no direct checkpoint. This could potentially lead to discrepancies if there's a significant change in voting power that wasn't checkpointed due to inactivity.

Recommendation

Review and potentially refactor the `_writeCheckpoint` function to handle out-of-order or missed checkpoints. Consider implementing mechanisms to retroactively fill missed checkpoints without affecting vote integrity.

Consider Historical Activity: If the token holders are highly active and frequently change their delegations or balances, the current implementation might suffice. However, if there are long periods of inactivity, it could be worth reconsidering this approach.

Outcome

Resolved by refactoring

L2. Insufficient Input Validation in `createPool` Function

Risk	Low
Affects	MiningPoolManager.sol

The `createPool` function is crucial for establishing new mining pools but currently lacks comprehensive input validations, such as uniqueness and specific format constraints.

Exploitation Example:

An attacker could create pools with identical or misleading names, causing confusion among users and potentially diverting rewards. Additionally, names with special characters or inconsistent formats could disrupt user interfaces or integrations.

Impact:

- **Duplicate Pool Names:** Allows the creation of multiple pools with the same name, leading to operational confusion and potential fund misallocation.
- **Inconsistent Name Formats:** Causes challenges in user interfaces, potentially leading to user errors or exploitation.

Recommendation

Consider adding the following checks to `poolNameCheck` modifier:

- **Uniqueness Check:** Verify that new pool names do not exist in `poolIdByName`.
- **Format Validation:** Enforce consistent name formats, including character restrictions and case sensitivity rules.
- **Input Sanitisation:** Remove or escape special characters to prevent injection attacks.

Outcome

Resolved the uniqueness already existed, and validation and sanitisation were added.

L3. Missing Require Checks in `payStrike()`

Risk	Low
Affects	WhirlMine.sol

`payStrike()` Line 393, call may revert, no require checks on call reverting before proceeding to L395-398

Recommendation

Check for errors before proceeding

Outcome

Resolved : the return value is checked in a require.

L4. Reentrancy possible

Risk	Low
Affects	WhirlMine.sol
	WhirlReferralRegistry.sol

In `payWhirlWithReferrer()` in WhirlMine reentrancy is possible if `_blokMiner` is a carefully crafted malicious contract.

In WhirlReferralRegistry.sol with function `register(uint256 referrerCode)` line 73 `payable(referrer).transfer(reward)` if referrer is a contract, it can execute re-entrancy callbacks

Recommendation

Add a reentrancy guard

Outcome

Resolved with the addition of reentrancy checks

L5. Handling of deactivatePool

Risk	Low
Affects	MiningPoolManager.sol

deactivatePool lacks a comprehensive process for handling current pool members and their rewards.

An abrupt deactivation could leave miners in limbo, uncertain about their contributions and rewards.

Impact:

- **Member Uncertainty:** Miners in a deactivated pool might lose access to their contributions and rewards.
- **Operational Disruption:** Lack of a clear deactivation process disrupts mining operations.

Recommendation

- **Notification Mechanism:** Inform miners about deactivation and provide a clear exit process.
- **Rewards Distribution:** Ensure accrued rewards are handled before deactivation.
- **Entry Prevention:** Stop new miners from joining a pool marked for deactivation.

We recommend a design where state is only held in one place. Also for optimisation a struct could be used to hold the requisite values.

Outcome

This issue is dropped as the handling of this is by design

L6. Use of Treasury Funds for Purchasing WHIRL

Risk	Low
Affects	WhirlTreasury.sol

In the WhirlTreasury contract, the method for utilising treasury funds to acquire WHIRL tokens is not transparent and might lack sufficient controls to prevent the rapid depletion of treasury funds or acquisition of WHIRL at unfavourable rates. This process is particularly evident in functions like clearExcess and realignWhirlPrice.

Functions like clearExcess and realignWhirlPrice in WhirlTreasury.sol demonstrate the use of treasury funds to purchase WHIRL. However, there are no explicit controls or limits detailed in these functions to prevent potential financial mismanagement. Such unregulated use of treasury funds could lead to severe financial repercussions for the WHIRL ecosystem. It could destabilise the WHIRL market, potentially causing inflated token prices and subsequent crashes, leading to a loss of trust among investors and users.

Exploitation Path

- **Rapid Depletion of Treasury Funds:** Without proper limits, an attacker (or even a well-intentioned user) could trigger these functions repeatedly, leading to a rapid depletion of treasury funds.
- **Purchasing at Unfavourable Rates:** In the absence of adequate market analysis and rate checks, the contract could end up purchasing WHIRL at inflated prices, especially during market volatility.
- **Artificial Price Inflation:** Large-scale purchases of WHIRL using treasury funds could artificially inflate the token's market price, leading to market instability and potential long-term devaluation.

Recommendation

- **Implement Spending Limits and Frequency Caps:** Introduce daily or weekly limits on the amount of WHIRL that can be purchased using treasury funds. Also, implement frequency caps to prevent repeated triggering of purchasing functions in a short time frame.
- **Market Analysis Mechanism:** Before executing purchases, incorporate a mechanism to analyse current market conditions. This could include checking the average price over a certain period to avoid buying at peak prices.
- **Dynamic Pricing Strategy:** Use a dynamic pricing strategy that adjusts the purchase rate based on market conditions, ensuring that treasury funds are used effectively without causing market disruption.
- **Governance Approval for Large Purchases:** For substantial treasury fund usage, require governance approval, ensuring that significant financial decisions are made transparently and with community consent.

Outcome

This issue is dropped after discussion with the development team.

L7. Swap Execution and Slippage Control in clearExcess

Risk	Low
Affects	WhirlTreasury.sol

The clearExcess function in WhirlTreasury is designed to convert excess treasury ETH to WHIRL tokens. However, its current implementation might allow for market manipulation, particularly if large amounts of ETH are involved. Specifically, the function's reliance on a single liquidity pool and the fixed slippage parameter (acceptableSlippage) could be exploited.

Impact

An attacker could use a flash loan to temporarily manipulate the price of WHIRL in the liquidity pool, triggering clearExcess to swap ETH for WHIRL at an inflated rate. After the swap, the attacker could sell the acquired WHIRL tokens at a higher rate in other transactions within the same block, and then repay the flash loan.

Recommendation

- Implement dynamic slippage control based on current market conditions.
- Introduce a time-weighted average price (TWAP) mechanism to determine the swap rate, reducing vulnerability to short-term price manipulation.
- Split large swaps into multiple smaller transactions to minimise market impact.
- Utilise external price oracles for additional price verification.
- Implement circuit breakers that pause swapping in case of detected abnormal activities, such as sudden large price movements.

Outcome

Resolved, with the addition of access control and mitigation against flash loans.

Informational Issues

I1. Hardcoded Domain Separator in Constructor

Risk	Info
Affects	Whirl.sol

The current implementation recomputes the domain separator each time in `delegateBySig()`. Hardcoding the domain separator in the constructor would optimize gas usage by avoiding repeated computations.

Increased gas costs for transactions involving `delegateBySig()`, potentially deterring users from participating in governance due to higher costs.

Recommendation

Hardcode the domain separator in the constructor to optimize gas usage. This change would reduce the transaction costs for users when delegating voting power, thereby encouraging more active participation in governance.

Outcome

Resolved by removing functionality from this contract.

I2. `_safe32()` called by `_writeCheckpoint()` assumes that the `block.number` will not grow beyond 2^{32} .

Risk	Low
Affects	Whirl.sol

`_safe32()` called by `_writeCheckpoint()` assumes that the `block.number` will not grow beyond 2^{32} . Based on those assumptions, if blocks reach production of 7,000,000 blocks per day, calls to this function from `_delegate()`, `_moveDelegates()` and `_writeCheckpoint()` could start failing due to Denial of Service in about 1.5 years of 7mil block per day

If this function fails it will also impact `WhirlHelpers.sol` and `RandomStrike.sol`

Outcome

Resolved by removing functionality from this contract.

I3. View Function with OnlyOwner Modifier

Risk	Info
Affects	Whirl.sol

The `getMinter` function is restricted to the contract owner, an unusual practice for a view function. This could obscure information from regular users, potentially impacting transparency

Recommendation

Make the `getMinter` function public to align with standard practices for view functions. If there are specific reasons for restricting access (e.g., security concerns), document these clearly. Alternatively, manage access control off-chain in the application layer, if necessary.

Outcome

Resolved by removal of the modifier

I4. Max Supply Set But Never Used

Risk	Info
Affects	Whirl.sol

The contract defines a `_maxSupply` constant but does not utilize it, leading to potential misunderstandings about the token's supply mechanics.

```
uint256 private constant _maxSupply = type(uint256).max; // << NO MAX SUPPLY
```

Recommendation

Clarify the purpose of `_maxSupply` . If it's meant to enforce a maximum supply limit, integrate its use in minting functions. If it's not required, consider removing it to avoid confusion.

Outcome

Resolved by removal of MaxSupply

15. Optimisation of `_moveDelegates`

Risk	Info
Affects	Whirl.sol

The `_moveDelegates` function can be optimized for gas efficiency by reducing storage reads and writes.

Recommendation

Implement the provided optimised version of `_moveDelegates` to reduce gas costs. This involves reducing storage operations and using memory variables for intermediate calculations.

Here's an optimised version of the `_moveDelegates` function:

```
function _moveDelegates(
    address _srcRep,
    address _dstRep,
    uint256 _amount
) internal {
    if (_srcRep != _dstRep && _amount > 0) {
        _updateDelegateVotes(_srcRep, _amount, false); // Decrease old
representative's votes
        _updateDelegateVotes(_dstRep, _amount, true); // Increase new
representative's votes
    }
}

function _updateDelegateVotes(
    address _delegate,
    uint256 _amount,
    bool _increase
) internal {
    if (_delegate != address(0)) {
        uint32 delegateNum = numCheckpoints[_delegate];
        uint256 delegateOld = delegateNum > 0 ? checkpoints[_delegate]
[delegateNum - 1].votes : 0;
        uint256 delegateNew = _increase ? delegateOld + _amount : delegateOld -
_amount;

        if (delegateNew != delegateOld) { // Only update if there's a change
            _writeCheckpoint(_delegate, delegateNum, delegateOld, delegateNew);
        }
    }
}
```

The function now checks if there is an actual change in the votes before making a write operation to the storage. This is a significant optimisation because storage operations are expensive in terms of gas.

The updated function handles potential edge cases where the vote balance does not change, avoiding unnecessary storage writes. This is particularly beneficial for contracts like Whirl, which might have frequent delegate changes.

The values `srcRepOld`, `srcRepNew`, `dstRepOld`, and `dstRepNew` are calculated once and stored in memory. This reduces the number of storage accesses.

The function checks if the new and old values are different before calling `_writeCheckpoint`. This avoids unnecessary storage writes when the balance doesn't change.

This approach avoids duplicating the logic for updating votes, making the function easier to read and maintain.

Outcome

Resolved by removal.

16. Use of Kill Switches

Risk	Low
Affects	MiningPoolManager.sol

`setKillSwitches` provides emergency control but centralises power, potentially undermining decentralisation.

Central control over key functions can erode trust in the decentralised system.

Recommendation

Ensure kill switches are governed transparently, possibly involving community consensus. Clearly document the conditions and procedures for activating kill switches.

Outcome

Resolved, the development team advise that these switches will be documented as short term use only and will be removed in discussion with the community.

17. Pull vs Push Withdrawal Mechanism

Risk	Low
Affects	MiningPoolManager.sol

The `withdrawPoolRewards` and `withdrawOwnerRewards` functions in the `MiningPoolManager` contract currently utilise a push mechanism for distributing rewards. Transitioning to a pull mechanism can enhance security and gas efficiency.

Impact:

- Gas Inefficiency:** Push mechanisms can lead to higher gas costs, especially when iterating over multiple beneficiaries.

- **Scalability Concerns:** As the number of pool members increases, the push approach becomes less scalable due to gas limits.

Recommendation

Here's an example of how the `withdrawPoolRewards` function could be refactored to a pull mechanism:

```
function claimPoolRewards(uint256 _blokTime) public nonReentrant {
    uint256 _minerPoolId = minerInBlokForPool[_blokTime][msg.sender];
    require(_minerPoolId != 0, "You did not participate in mining that blok with a pool.");
    require(!claimedForBlok[_blokTime][msg.sender], "You have already claimed for this blok.");

    MiningPool storage _pool = poolById[_minerPoolId];
    uint256 _totalMinerRewards = poolRewardsThisBlok[_blokTime][_minerPoolId];
    uint256 _ownerRewards = _totalMinerRewards * _pool.ownerFee / 10000;
    uint256 _commonRewards = _totalMinerRewards - _ownerRewards;

    uint256 _minerMiningPower = miningUnitsInPools[_blokTime][msg.sender];
    uint256 _totalMiningPower = poolTotalForBlok[_blokTime][_minerPoolId];

    uint256 _rewards = _commonRewards * _minerMiningPower / _totalMiningPower;

    // Instead of transferring immediately, record the claimable amount
    membershipRewards[msg.sender] += _rewards;
    claimedForBlok[_blokTime][msg.sender] = true;

    emit MemberWithdrawal(msg.sender, _blokTime, _rewards);
}

function withdrawClaimedRewards() public nonReentrant {
    uint256 rewards = membershipRewards[msg.sender];
    require(rewards > 0, "No rewards to withdraw");

    membershipRewards[msg.sender] = 0;
    require(IERC20(whirl).transfer(msg.sender, rewards), "Withdrawal failed");

    emit RewardsWithdrawn(msg.sender, rewards);
}
```

The original function performs a token transfer (`IERC20(whirl).transfer`) within each withdrawal call. This process is gas-intensive as it interacts with the ERC20 token contract on each call. In contrast, the modified version separates the claim process from the actual transfer of tokens. The claim function (`claimPoolRewards`) merely records the claimable amount without initiating a token transfer. The actual transfer occurs later in a separate function (`withdrawClaimedRewards`), reducing the immediate transaction costs.

As the number of users grows, the original approach would become increasingly gas-inefficient because each withdrawal would involve a separate token transfer. The modified approach scales

better because users independently claim and withdraw their rewards, distributing the gas costs over time and across multiple transactions.

Outcome

This issue has been dropped after discussion with the development team.

18. Loop Optimisation in `getAllMiningPools`

Risk	Low
Affects	MiningPoolManager.sol

The `getAllMiningPools` function has potential gas inefficiencies due to loops. This is only mentioned as the function has a public modifier, although it could be made external as it is not called anywhere in the contract.

As the number of pools grows, this function becomes more expensive to execute.

Recommendation

Either change the modifier to external rather than public or :

- **Pagination:** Implement pagination to limit the number of pools returned in a single call.
- **On-Demand Retrieval:** Allow querying of individual pools instead of returning all at once.

Outcome

Resolved : visibility changed to external

Appendix A

General Audit Goals

We audit the code in accordance with the following criteria:

Sound Architecture

This audit includes assessments of the overall architecture and design choices. Given the subjective nature of these assessments, it will be up to the development team to determine whether any changes should be made.

Smart Contract Best Practices

This audit will evaluate whether the codebase follows the current established best practices for smart contract development.

Code Correctness

This audit will evaluate whether the code does what it is intended to do.

Code Quality

This audit will evaluate whether the code has been written in a way that ensures readability and maintainability.

Security

This audit will look for any exploitable security vulnerabilities, or other potential threats to the users.

Although we have commented on the application design, issues of cryptoeconomics, game theory and suitability for business purposes as they relate to this project are beyond the scope of this audit.