Security Audit Report

Cypher Protocol

v1.4

July 30, 2025

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by CypherD Wallet Inc to perform a security audit of Cypher Protocol.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities that could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/CypherD-IO/CypherProtocol
Commit	`0d6847e970056d7de8184b9bc2a67fe1d5f6b17a`
Scope	Only the following Solidity contracts were in scope: • `src/VotingEscrow.sol` • `src/Election.sol` • `src/RewardDistributor.sol` • `src/DistributionModule.sol`
Fixes verified at commit	`30fdfee7edddc55ba6ca6631941f82e2815fc001`

	Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.
Update	https://github.com/CypherD-IO/CypherProtocol/pull/16 reviewed at commit `1505e293bc0c94f69f9b973a8dfe51104841c4ef`, audit-fixes branch at `30fdfee7edddc55ba6ca6631941f82e2815fc001`.
Fixes verified at commit	`bb82650a7400a3ed5baaf108dbf499ae49f75e74`

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Cypher is a crypto card platform that allows users to spend cryptocurrency for real-world expenses. $CYPR tokens are used to incentivize spending and referrals, with a merchant voting system where users lock $CYPR tokens to gain voting power. This voting power directs incentives to merchants, enhancing user engagement and merchant participation.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium-High	-
Test coverage	High	`forge coverage` reports a line test coverage of `98.40%` (`432/439`). Note that we encountered "stack too deep" errors and had to apply small changes to the `Election` contract before determining the test coverage.

Summary of Findings

No	Description	Severity	Status
1	Malicious users can inflate `votesForCandidateInPeriod` by exploiting the `merge` function in the `VotingEscrow` contract, leading to improper reward distribution	Critical	Resolved
2	Users cannot vote upon locking due to an invalid vote power calculation	Major	Resolved
3	Malicious users can add a large number of items to the `votedCandidates` and `votedWeights` arrays, potentially causing a Denial-of-Service of the `refreshVotesFor` function	Major	Resolved
4	Missing NFT flash loan protection allows voting and returning the NFT within the same block	Minor	Acknowledged
5	Users can end up losing one epoch's worth of rewards	Minor	Acknowledged
6	The `totalSupplyAt` function does not add the indefinite amount to the final bias, resulting in an incorrect total supply	Minor	Resolved
7	The `withdraw` function in `VotingEscrow` could end up sending the `CYPR` tokens to the invalid address	Informational	Resolved
8	Add `nonReentrant` modifier to the `refreshVotesFor` function	Informational	Acknowledged
9	Cache array length outside of loops	Informational	Resolved
10	Cache the `votedCandidates` array used in the `refreshVotesFor` function in memory to reduce repeated storage reads	Informational	Resolved
11	Unused `MAX_LOCK_DURATION` constant	Informational	Resolved
12	Missing maximum capacity for `tokensOwnedBy` in the `VotingEscrow` contract could lead to reverts	Informational	Resolved
13	Use `super` for nested inheritance function overriding	Informational	Acknowledged
14	Cache storage variables when used multiple times	Informational	Resolved

15	Using `ERC721`'s `_mint` function might mint tokens to addresses that do not support NFTs	**Informational**	**Acknowledged**
16	Bribes for periods without votes should be recoverable	**Informational**	**Acknowledged**
17	Voting with an empty `candidates` array should result in an error	**Informational**	**Resolved**
18	Inaccurate `EmmissionSchedules` due to integer division	**Informational**	**Acknowledged**
19	Safe module needs to be registered	**Informational**	**Acknowledged**
20	Inability to change Merkle roots and lack of sufficient reward checks in the `RewardDistributor` contract	**Informational**	**Acknowledged**

Detailed Findings

1. Malicious users can inflate `votesForCandidateInPeriod` by exploiting the `merge` function in the `VotingEscrow` contract, leading to improper reward distribution

Severity: Critical

The Cypher protocol enables users to lock up their tokens and utilize them as voting power, where each lock enables the user to place a vote once per epoch. However, the `merge` function in `src/VotingEscrow.sol:157-186` allows users to merge tokens that have already voted in an epoch, which means that they can use the same vote power multiple times. Even though there is no direct way to utilize more bribe rewards (as the from token is burned), a malicious user can inflate `votesForCandidateInPeriod`, which serves as a denominator when calculating the bribe rewards, leading to improper reward distribution.

Recommendation

We recommend adding a check to ensure that the tokens being merged have not already voted in the current epoch.

Status: Resolved

2. Users cannot vote upon locking due to an invalid vote power calculation

Severity: Major

The current voting mechanism in `src/Election.sol:108` calculates the vote power of a user based on the balance of the user's tokens at the time of the period start by calling `ve.balanceOfAt(tokenId, periodStart)`. By doing so, if a user joins in the middle of the period, they will not have any voting power and will be unable to vote until the next period. This is a major issue as voting power is at its highest the moment they lock, and it starts decreasing each day.

Additionally, in `src/VotingEscrow.sol:411`, the `points[1].ts > timestamp` check is done before the `points[lastEpoch].ts <= timestamp` check, which could lead to invalid balance returns.

Recommendation

We recommend using the current `block.timestamp` to calculate the vote power and move `points[1].ts > timestamp` check after `points[lastEpoch].ts <= timestamp` check.

Status: Resolved

3. Malicious users can add a large number of items to the `votedCandidates` and `votedWeights` arrays, potentially causing a Denial-of-Service of the `refreshVotesFor` function

Severity: Major

The current voting mechanism in `src/Election.sol:121-153` does not limit the number of candidates a user can vote for, and also allows for duplicate entries to be added. While this does not affect voting power allocation or bribe claims, it will enable users to potentially Denial-of-Service the vote refreshing mechanism via the `refreshVotesFor` function implemented in `src/Election.sol:156-190`.

A malicious user can call the `vote` function with thousands of duplicate candidate entries, filling up the `votedCandidates` and `votedWeights` arrays for this specific token. When this token is added to the batch of tokens for vote refreshing, the filled arrays will be iterated multiple times, eventually consuming all available gas and causing the function to revert due to an out-of-gas error.

The above allows a single user to perform a denial-of-service (DoS) attack on the vote refreshing functionality, preventing other users' votes from being refreshed if they are in the same batch (depending on how the vote refresh off-chain keeper builds the batches).

A test case is provided in Appendix 1.

Recommendation

We recommend disallowing duplicate candidate voting and potentially limiting the number of candidates a single user can vote for.

Status: Resolved

4. Missing NFT flash loan protection allows voting and returning the NFT within the same block

Severity: Minor

In `src/VotingEscrow.sol:222-230`, the `balanceOfAt` function calculates the veNFT's effective voting power at a specified timestamp for voting purposes. However, flash loan

protection is absent, enabling users to borrow a veNFT to vote and return it within the same block.

Recommendation

We recommend tracking the change of ownership of the veNFT at a given block and ensuring that it is not equal to the current `block.number` when calculating the effective voting power in the `balanceOfAt` function.

Status: Acknowledged

The client acknowledges this finding, noting that there will be no liquidity for these NFT tokens. The NFTs are used only for voting.

5. Users can end up losing one epoch's worth of rewards

Severity: Minor

Bribe rewards are naturally lagging with one epoch, meaning that they are claimable up to the previous epoch. The `mint` and `merge` functions in `src/VotingEscrow.sol:105` and `src/VotingEscrow.sol:168`, respectively, burn users' tokens, meaning that if a user votes in an epoch, they need to wait for the entire epoch to end, claim their rewards for the past epoch, and then merge or withdraw. If a user forgets to claim before merging or withdrawing, they will not be able to get the rewards.

Recommendation

We recommend properly documenting that users must claim their eligible bribes before withdrawing. Another solution is to claim for the user automatically.

Status: Acknowledged

The client acknowledges this finding, noting that the frontend will make sure that users claim any pending rewards before merging or withdrawing.

6. The `totalSupplyAt` function does not add the indefinite amount to the final bias, resulting in an incorrect total supply

Severity: Minor

The final return statement in `src/VotingEscrow.sol:216` does not add the indefinite amount to the final bias, resulting in an incorrect total supply. Any locks that are locked indefinitely will not be counted towards the total supply.

Recommendation

We recommend adding the point's `indefinite` amount to the final `bias`.

Status: Resolved

7. The `withdraw` function in `VotingEscrow` could end up sending the `CYPR` tokens to the invalid address

Severity: Informational

Users can approve external keepers to operate with their locked positions. Because of this, when users call the `withdraw` function, in `src/VotingEscrow.sol:111`, the `CYPR` tokens will be sent to the keeper's address, instead of the user's address. Similarly, this can also happen when bribes are claimed in `src/Election.sol:181`.

Recommendation

We recommend sending the `CYPR` and bribe tokens to the owner of the `tokenId`, instead of the caller.

Status: Resolved

8. Add `nonReentrant` modifier to the `refreshVotesFor` function

Severity: Informational

Although we did not identify an actual reentrancy risk, for an extra layer of safety, add a `nonReentrant` modifier to the vote refresh `refreshVotesFor function` implemented `src/Election.sol:156-190`.

Recommendation

We recommend adding a `nonReentrant` modifier to the vote refresh `refreshVotesFor` function.

Status: Acknowledged

9. Cache array length outside of loops

Severity: Informational

When using arrays in for loops, for example, in `src/DistributionModule.sol:136`, it is best to cache the length before starting the for loop, as this will reduce the storage memory reads, thus reducing gas costs.

Recommendation

We recommend storing emission schedules in a local variable, using it for both the length and the inner schedules within the for loop.

Status: Resolved

10. Cache the `votedCandidates` array used in the `refreshVotesFor` function in memory to reduce repeated storage reads

Severity: Informational

The `votedCandidates` array that is used in the `refreshVotesFor` function in `src/Election.sol:156-190` is iterated twice, causing repeated storage access, which raises the gas costs for executing the function. Although the length of the array, i.e., the number of candidates, is cached in memory, reading the entire array into memory can optimize gas costs.

Recommendation

We recommend caching the whole `votedCandidates[tokenId]` array into memory.

Status: Resolved

11. Unused `MAX_LOCK_DURATION` constant

Severity: Informational

The `MAX_LOCK_DURATION` constant in `src/Election.sol:17` is unused.

Recommendation

We recommend removing `MAX_LOCK_DURATION`.

Status: Resolved

12. Missing maximum capacity for `tokensOwnedBy` in the `VotingEscrow` contract could lead to reverts

Severity: Informational

The `tokensOwnedBy` function in `src/VotingEscrow.sol:227-233` is missing a maximum for the total number of tokens to be iterated. Depending on how and where this function is called, it can lead to an out-of-gas error that potentially impacts off-chain services.

Recommendation

We recommend adding a paging functionality where tokens are grouped in pages with limits, instead of being returned all at once.

Status: Resolved

13. Use `super` for nested inheritance function overriding

Severity: Informational

The new addition of the `ERC721Enumerable` package in `src/VotingEscrow.sol` requires some of its methods to be overridden for correct usage. When referencing the parent, it is best practice to use `super` instead of directly calling the package. Using `super` will ensure that the proper method resolution order is maintained even in the event of any inheritance changes.

Recommendation

We recommend using `super` for the overridden functions in `VotingEscrow.sol:218,275,284`.

Status: Acknowledged

14. Cache storage variables when used multiple times

Severity: Informational

Storage loads are significantly more expensive than memory loads. Therefore, storage values that are read multiple times should be cached to avoid multiple storage loads.

Recommendation

We recommend storing `lastEmissionTime` in `src/DistributionModule.sol:131` in a local variable and using it throughout.

Status: Resolved

15. Using `ERC721`'s `_mint` function might mint tokens to addresses that do not support NFTs

Severity: Informational

The `_mint` function can mint `ERC721` tokens to addresses, specifically, wallets or contracts that don't support `ERC721` tokens. This means that when users lock their tokens in `src/VotingEscrow.sol:251`, they may unexpectedly receive an NFT, which would lock their funds inside the protocol.

Recommendation

We recommend using `_safeMint` instead of `_mint`.

Status: Acknowledged

The client acknowledges this finding, noting that users not utilizing the official frontend should be sophisticated enough to understand when they are receiving an NFT.

16. Bribes for periods without votes should be recoverable

Severity: Informational

In `src/Election.sol:187-197`, the `addBribe` function enables anyone to add a bribe for a specified `candidate` in the current voting period. However, if no votes are recorded for this period, the bribes cannot be recovered and will stay locked in the contract.

Recommendation

We recommend adding functionality that allows, for example, the contract owner to withdraw unclaimed bribes for previous periods.

Status: Acknowledged

The client acknowledges this finding, noting that they are aware of the issue and will ensure that at least one vote is cast during the period before closing the election for the epoch.

17. Voting with an empty `candidates` array should result in an error

Severity: Informational

In `src/Election.sol:102-130`, a user that is authorized to vote for the given `tokenId` can vote for specific `candidates` via the `vote` function. The `lastVoteTime` variable prevents duplicate voting within a period, triggering an "AlreadyVoted" error if a second attempt is made in the same period.

However, if a user calls `vote` with an empty `candidates` array, the transaction succeeds without recording a vote, yet it still updates `lastVoteTime`. This prevents the user from voting again until the next period. Consequently, a user who mistakenly submits an empty vote must wait for the subsequent voting period to cast their intended vote.

Recommendation

We recommend checking whether at least one candidate is provided and erroring if it is empty.

Status: Resolved

18. Inaccurate `EmmissionSchedules` due to integer division

Severity: Informational

In `src/DistributionModule.sol:115`, the amount of tokens allocated for a given `EmmissionSchedule` is calculated. The number of tokens intended for each schedule is known and hard-coded in the contract's construction. However, the design does not account for the integer division rounding in Solidity.

For example, the first `EmmissionSchedule` contains 5 million tokens to be distributed in the span of 13 weeks. However, the 5 million tokens do not divide evenly by the number representing 13 weeks, which makes the `EmmissionSchedule` distribute less than 5 million tokens. A similar calculation error is associated with other schedules defined in the `DistributionModule` contract.

Recommendation

We recommend either changing the `EmmissionSchedules` so that they reflect an actual intended token distribution. This can be achieved by either adjusting the schedules to prevent integer division from affecting the token amount or by updating the documentation to accurately reflect the actual number of tokens that will be distributed.

Status: Acknowledged

The client acknowledges this finding, noting that only a tiny, negligible amount is lost. It is not a practical consequence worth fixing. The total error is robustly quantified in the tests at less than 10^{-15} of a token.

19. Safe module needs to be registered

Severity: Informational

In `src/DistributionModule.sol:183`, the `DistributionModule` contract uses the Safe's `execTransactionFromModule` functionality to transfer the tokens. It must be noted that the `execTransactionFromModule` function will only be able to complete successfully

if the Module (i.e., the `DistributionModule` contract) is registered within Safe's `ModuleManager`.

There is no check within the `DistributionModule` contract that ensures it is already registered prior to attempting execution in the `ModuleManager`. As there is no direct security impact and the transaction will be reverted if attempted without prior registration, we decided to report this as an informational issue.

Recommendation

We recommend either implementing a check in the `DistributionModule` contract to ensure its registration within `ModuleManager` or implementing a business-level process that ensures such registration is performed before attempting to transfer the tokens.

Status: Acknowledged

The client acknowledges this finding, noting that the module gets registered by the deployment script. Adding a check just increases gas costs for users post-registration, and does not change the functionality pre-registration in a meaningful way.

20. Inability to change Merkle roots and lack of sufficient reward checks in the `RewardDistributor` contract

Severity: Informational

The `RewardDistributor` contract implements the `addRoot` function in `src/RewardDistributor.sol:26-31`. This function is responsible for adding Merkle roots, which are then used to authorize the actual reward distribution using Merkle proofs.

It should be noted that there is no functionality that would allow changing or removing the root. This could lead to potential distribution issues if the root is added by accident or if it is malformed or compromised.

Furthermore, the `RewardDistributor` contract does not ensure that it owns enough tokens to cover the whole distribution.

Recommendation

We recommend implementing an administrative function that allows modifying the Merkle root, possibly by invalidating one, should an issue with it be identified.

Furthermore, we recommend implementing a check within the contract to ensure ownership of a sufficient amount of tokens to cover the token distribution. Alternatively, an off-chain process can be created to monitor the token balance and trigger a top-up when needed.

Status: Acknowledged

The client acknowledges this finding, noting that this is a conscious decision to prevent any kind of tampering with the Merkle root once added.

Appendix: Test Cases

1. Test case for "Malicious users can add a large number of items to the votedCandidates and votedWeights arrays, potentially causing a Denial-of-Service of the refreshVotesFor function"

```
function testDoSRefresh() public {
        _warpToNextVotePeriodStart();

        cypher.approve(address(ve), 200e18);
        uint256 id = ve.createLock(200e18, MAX_LOCK_DURATION);
        ve.lockIndefinite(id);

        cypher.approve(address(ve), 200e18);
        uint256 id2 = ve.createLock(200e18, MAX_LOCK_DURATION);
        ve.lockIndefinite(id2);

        bytes32[] memory candidates = new bytes32[](10000);
        for (uint256 i = 0; i < 10000; i++) {
            candidates[i] = CANDIDATE1;
        }

        election.enableCandidate(CANDIDATE1);
        uint256[] memory weights = new uint256[](10000);
        for (uint256 i = 0; i < 10000; i++) {
            weights[i] = 1;
        }

        election.vote(id, candidates, weights);
        election.vote(id2, candidates, weights);

        _warpToNextVotePeriodStart();

        election.authorizeVoteRefresher(address(this));
        uint256[] memory ids = new uint256[](2);
        ids[0] = id;
        ids[1] = id2;
        vm.expectRevert();
        election.refreshVotesFor(ids);
    }
```