EMDX Audit

EMDX protocol

December 2021

By CoinFabrik

Introduction

CoinFabrik was asked to audit the contracts for the EMDX project. First we will provide a summary of our discoveries and then we will show the details of our findings.

Summary

The contracts audited are from the EMDX repository at https://github.com/emdx-dex/perpetual-protocol/. The audit is based on the commit `ca34a5e9e254caca5056ab5002a3fdb0b4f3b004`.
Fixes were made and rechecked based on the commits:

1. `ee9bb77d8a96ec409970d795d6eb8b40acba5446` (typos)
2. `4f9f71afc7853a9bd991dc25e9757f2d4627c05f` (ME-01)
3. `4dc179ff37796d6d59d5a0220e176144260f7d6e` (MI-01)

Contracts

The audited contracts are:

- `src/ChainlinkL1.sol: Chainlink Oracle interface`
- `src/L2PriceFeed.sol: Asset price feed`
- `src/ClearingHouse.sol: Core perpetual protocol contract`
- `src/InsuranceFund.sol: Insurance fund`
- `src/interface/Iark.sol: Ark interface`
- `src/Ark.sol: Minter replacement to cover Insurance fund collateral depletion.`

Contracts are a fork of the Perpetual protocol (https://github.com/perpetual-protocol/perpetual-protocol), except for IArk.sol and Ark.sol.

Analyses

The following analyses were performed:

- Misuse of the different call methods
- Integer overflow errors
- Division by zero errors
- Outdated version of Solidity compiler
- Front running attacks
- Reentrancy attacks
- Misuse of block timestamps
- Softlock denial of service attacks
- Functions with excessive gas cost
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Failure to use a withdrawal pattern
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures

Findings and Fixes

ID	Title	Severity	Status
ME-01	Denial of Service in Ark.withdrawForLoss()	Medium	Fixed
MI-01	Invalid calculation in Ark.withdrawForLoss()	Minor	Fixed
MI-02	Oracle manipulation on ChainlinkL1.updateLatestRoundData()	Minor	Mitigated
MI-03	Solidity Compiler Version	Minor	Acknowledged
EN-01	Uninitialized Contract address	Enhancement	Mitigated

Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of but can be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.
- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

SEVERITY	EXPLOITABLE	ROADBLOCK	TO BE FIXED
Critical	Yes	Yes	Immediately
Medium	In the near future	Yes	As soon as possible
Minor	Unlikely	No	Eventually
Enhancement	No	No	Eventually

Issues Found by Severity

Critical Severity

No issues found

Medium Severity

ME-01 Denial of Service in `Ark.withdrawForLoss()`

In the function `Ark.withdrawForLoss()` it is possible to withdraw more than the total balance of the token. This should be prevented by the require statement at line 46 of `Ark.sol`:

```
require(_balanceOf(_quoteToken, address(this)).toUint() >= _amount.toUint(),
"insufficient funds");
```

But in the next step in `Ark.sol`, line 50, the amount to withdraw is incremented by one to compensate for the method DecimalERC20 that rounds down in the conversion from decimal to uint256. But if the token has 18 or more decimals, the conversion to uint256 is done at the line 103 in `DecimalERC20.sol`:

```
return _decimal.toUint().mul(10**(tokenDecimals.sub(18)));
```

We can see that in this case the conversion is simply a multiplication, so no rounding down is done. Subsequently, by increasing the amount by one at line 50 in Ark.sol, the amount to withdraw is now bigger than the balance, and the `_transfer()` will fail.

Recommendation

The assumption that Decimal to uint256 conversion always rounds down the value is incorrect, as this only happens with tokens with less than 18 decimals. Adjust the code to reflect this condition or require only tokens with less than 18 decimals.

Solution

The issue was fixed by the development team by compensating the amount only when necessary.

Minor Severity

MI-01 Invalid calculation in `Ark.withdrawForLoss()`

When calling `Ark.withdrawForLoss()` it is possible to operate with different ERC20 tokens that are passed as parameter `_quotetoken`.

There are two issues when operating with different tokens:

1) At line 57 in `Ark.sol` a log of the transfers is calculated in this way:

```
cumulativeAmount = withdrawnTokenHistory[len -
1].cumulativeAmount.addD(_amount);
```

This calculation doesn't consider that _amount might represent amounts of different tokens. This issue is not serious because the variable `withdrawnTokenHistory` is currently unused.

2) At line 64 in `Ark.sol` we can see the transfer and associated event being generated:

```
_transfer(_quoteToken, _msgSender(), _amount);
emit WithdrawnForLoss(_msgSender(), _amount.toUint());
```

We can see that the transfer specifies the token type, but the event does not, so any listener of this event has no way to know which token was withdrawn.

Recommendation

Refactor the code to support operation with different tokens, or limit the contract to operate on a single token type.

Solution

The issue was fixed by the development team by supporting different tokens in the event and in the token history.

MI-02 Oracle manipulation on `ChainlinkL1.updateLatestRoundData()`

In ChainlinkL1.sol line 118 we can see a decimal conversion calculation:

```
return _price.mul(TOKEN_DIGIT).div(10**uint256(_decimals));
```

This exponentiation operation will overflow in any token with more than 77

decimals, causing an exception in the `UpdateLatestRoundData()` function.
The same overflow was detected in the following contracts but are out of the scope of this audit, and impact was not analyzed:

- `DecimalMath.sol`, line 13
- `DecimalERC20.sol`, line 94
- `DecimalERC20.sol`, line 103

Recommendation
Require a limit or a range for token decimals accepted by this contract.

Solution
The issue was mitigated by the fact that the dev team won't use tokens with more than 18 decimal places.

MI-03 Solidity Compiler Version

All audited files use the pragma `solidity 0.6.9;` statement. This solidity version is outdated and may introduce bugs. See https://swcregistry.io/docs/SWC-103.

Recommendation

Update the solidity compiler to the latest revision (0.8.10 at the current time). Also, when updating to 0.8 take into account the new semantics for safe math operations.

Solution
The issue was acknowledged by the dev team, but it was decided to keep using the compiler because refactoring it is too costly at this time.

Enhancements

EN-01 Uninitialized Contract address

At line 86 in `L2PriceFeed.sol`, the modifier onlyChainlink() restricts the call to chainlinkContract, previously set calling `ChainlinkL1.setChainlink()`. However, it is not verified or assured in any way that this function was called, as a result the chainlinkContract can be uninitialized, preventing this function from updating the prices until the `setChainlink()` function is called.

Recommendation
Initialize the `chainlinkContract` address in the initializer of the contract, so it never can be used uninitialized.

Solution

The issue was contemplated in the deployment script and thus mitigated.

Other considerations

Documentation

Documentation is inconsistent throughout the source code. The core contract `ClearingHouse.sol` has good documentation of functions and arguments, but it's not as detailed in `Ark.sol, ChainlinkL1.sol, InsuranceFund.sol` and `L2PriceFeed.sol`.

Centralization

The contract `ChainlinkL1.sol` allows the owner to change the price oracle, and in the case of `Ark.sol` it allows the owner of the contract to withdraw all funds using the `claimTokens()` function. Also, all contracts are upgradeable by the owner.

Users of the EMDX protocol need to trust the owner of the contract because of this fact.

Conclusion

We found 1 medium-severity issue and 3 minimal-severity issues. Also we suggested two enhancements to the code. A medium-severity issue and a minor-severity issue were fixed. A minor-severity issue and an enhancement were mitigated. Finally, a minor-severity issue was acknowledged but a fix was not considered necessary by the development team.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the EMDX project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.