DeHacker

Code Security Assessment

ZERONE

May25th, 2023

Contents

Summary

DeHacker's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.
Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow/underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service/logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Issue Categories

Every issue in this report was assigned a severity level from the following:

Critical severity issues

A vulnerability that can disrupt the contract functioning in a number of scenarios or creates a risk that the contract may be broken.

Major severity issues

A vulnerability that affects the desired outcome when using a contract or provides the opportunity to use a contract in an unintended way.

Medium severity issues

A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.

Minor severity issues

A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.

Informational

A vulnerability that has informational character but is not affecting any of the code.

Overview

Project Summary

Project Name	ZERONE
Platform	ARBITRUM
website	zeronedex.com
Type	Dex
Language	Solidity

Vulnerability Summary

Vulnerability Level	**Total**	**Pending**	**Declined**	**Acknowledged**	**Partially Resolved**	**Resolved**
Critical	0	0	0	0	0	0
Major	1	0	0	0	0	1
Medium	1	0	0	0	0	1
Minor	1	0	0	0	0	1
Informational	2	0	0	0	0	2
Discussion	0	0	0	0	0	0

Audit scope

ID	File	SHA256 Checksum
TFC	PerpetualTradeAgent.sol	b2f4eb75079a8bee85da4ac321ffbea7929bf14893da2cd146a40a64160674ef
ODS	SLDPerpetual.sol	3f8c6d2cc16e09e24567d9a933931678a2e3897db02fbc57c10135e354117bbc
FPO	PerpetualPrivatePool.sol	f7225388c1d69d57e6251c9fda50cbbf9e05131e5adb81e5aa0422402f048162

Findings

ID	Category	Severity	Status
GLOBAL-01	Centralization Related Risks	Major	Solved
TFC-01	Logical Issue	Medium	Solved
FPO-01	Duplicate Writes	Minor	Solved
ODS-01	Redundant Code	Informational	Solved
ODS-02	costly-loop	Informational	Solved

Major

GLOBAL-01 | Centralization Related Risks

Category	Severity	Location	Status
Centralization Related Risks	Major	Global	Solved

Description

The owner of the PerpetualTradeAgent contract has the following permissions:
function setAddressParam()
The owner of the SLDPerpetual contract has the following permissions:
function setParam()
The owner will have the ability to influence the operation results of the protocol.

Recommendation

This finding describes the level of decentralization of the project, and it is recommended to strengthen security and improve the degree of decentralization from the following aspects:
It is recommended that privileged addresses use multi-signature wallet addresses.
For modification operations that affect protocol operation stability and key business parameters, it is recommended to implement time locks.

Medium

TFC-01 | Logical Issue

Category	Severity	Location	Status
Logical Issue	Medium	PerpetualTradeAgent.sol Line 636	Solved

Description

Line 636

```
function calDealPNL(
    ContractType _type,
    uint256 _currentPrice,
    uint256 _dealPrice,
    uint256 _amount,
    uint256 _willClose
) public pure returns (uint256 profit, uint256 loss) {
    if (_type == ContractType.LONG) {
        if (_currentPrice >= _dealPrice) {
            profit =
_amount.mul(_willClose).div(_amount).mul(_currentPrice.sub(_dealPrice)).div(PRI
CE_DECIMALS);
            loss = 0;
        } else {
            profit = 0;
            loss =
_amount.mul(_willClose).div(_amount).mul(_dealPrice.sub(_currentPrice)).div(PRI
CE_DECIMALS);
        }
    } else {
        if (_currentPrice <= _dealPrice) {
            //                  profit = _amount
            //                  .mul(PRICE_DECIMALS)
            //                  .div(_willClose)
            //                  .mul(_amount)
            //                  .mul(_dealPrice.sub(_currentPrice))
            //                  .div(PRICE_DECIMALS)
            //                  .div(PRICE_DECIMALS);
            profit =
_amount.mul(_willClose).div(_amount).mul(_dealPrice.sub(_currentPrice)).div(PRI
CE_DECIMALS);
```

```
            loss = 0;
        } else {
            profit = 0;
            //                  loss = _amount
            //                  .mul(PRICE_DECIMALS)
            //                  .div(_willClose)
            //                  .mul(_amount)
            //                  .mul(_currentPrice.sub(_dealPrice))
            //                  .div(PRICE_DECIMALS)
            //                  .div(PRICE_DECIMALS);
            loss =
_amount.mul(_willClose).div(_amount).mul(_currentPrice.sub(_dealPrice)).div(PRI
CE_DECIMALS);
        }
    }

    require(profit == 0 || loss == 0, "PNL error");
}
```

Calculation of `profit` and `loss`: `_amount.mul(_willClose).div(_amount)`. There is redundancy in the mathematical operation here. Or is there a logical issue?

Recommendation

Optimize the calculation formula and pay attention to the order of multiplication and division operations to prevent loss of precision.

Minor

FPO-01 | Duplicate Writes

Category	Severity	Location	Status
Duplicate Writes	Minor	PerpetualPrivatePool.sol Line 242	Solved

Description

Line 242

```
function close(
    uint256 _id,
    uint8 _type,
    uint256 _profit,
    uint256 _loss,
    uint256 _amount,
    uint256 _price,
    bool _transform
) public onlyKeeper returns (uint256 makerProfit, uint256 makerLoss, bool isLiquidated,
bool isAgreement) {
    ...
            makerProfit = 0;
            makerLoss = _profit;
            makerDeal.locked = false;
            {
                makerProfit =
_profit.sub(makerDeal.marginAmount.add(makerDeal.maintenanceMargin).add(account.available
Amount));
                uint256 riskFundBalance = getRiskFundAmount();
                if (makerProfit > riskFundBalance) {
                    _profit = _profit.sub(makerProfit).add(riskFundBalance);
                    makerLoss = _profit;
                    makerProfit = riskFundBalance;
                }
                if (makerProfit > 0) {
                    _safeTransferFrom(tokenAddress, riskFundAddr, address(this),
makerProfit);
                }
                makerProfit = 0;
            }


    ...
}
```

Repeatedly assign values to the `makerProfit` variable, and in either case, `makerProfit` is eventually assigned a value of `0`.

Recommendation

Remove duplicate writes to the `makerProfit` variable. If `makerProfit` is always `0` in this scenario, there is no need to repeatedly assign the `makerProfit` variable, use temporary variables to store the value during calculation and make conditional judgments.

Informational

ODS-01 | Redundant Code

Category	Severity	Location	Status
Redundant Code	Informational	SLDPerpetual.sol Line 363	Solved

Description

Line 363

```
function updateUserAsset(address _taker, int256 amount, int256 available,
uint256 margin, uint256 locked) public onlyKeeper {
    AccountInfo storage account = userAccount[_taker];
    if (amount >= 0) {
        account.depositAmount = uint256(amount);
    }
    if (available >= 0) {
        account.availableAmount = uint256(available);
    }
    if (margin >= 0) {
        account.marginAmount = uint256(margin);
    }
    if (locked >= 0) {
        account.orderLocked = uint256(locked);
    }
}
```

`margin`, `locked` are of type `uint256` and are always greater than 0, so checking if they are greater than 0 is redundant.

Recommendation

Remove the conditional statement here.

ODS-02 | costly-loop

Category	Severity	Location	Status
Redundant Code	Informational	SLDPerpetual.sol Line 313	Solved

Description

Line 313

```
function triggerOrders(uint256[] memory _orderIDs) public {
    for (uint256 i = 0; i < _orderIDs.length; i++) {
        ...
        if (result.flag) {
            ...
            localCaller = msg.sender;
        }
    }
}
```

Line 68

```
modifier onlyKeeper() {
    require(keeperMap[msg.sender] || msg.sender == localCaller, "caller is not
perpetual keeper");
    _;
    localCaller = address(0x0);
}
```

Line 346

```
function cancelOrders(uint256[] memory _orderIDs) public {
    for (uint256 i = 0; i < _orderIDs.length; i++) {
        localCaller = msg.sender;
        cancelOrder(_orderIDs[i], msg.sender, false);
    }
}
```

```
function cancelOrder(uint256 _orderID, address payable _taker, bool _status)
public onlyKeeper {
    uint256 locked = orderBook.cancelOrder(_orderID, _taker, _status);
    if (locked > 0) {
        AccountInfo storage account = userAccount[_taker];
        account.availableAmount = account.availableAmount.add(locked);
        account.orderLocked = account.orderLocked.sub(locked);
    }
}
```

Setting localCaller multiple times in a loop results in unnecessary gas waste.

Recommendation

Optimized the setting method for localCaller to reduce unnecessary gas waste.

Disclaimer

Appendix

Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Coding Style

Coding Style findings usually do not affect the generated bytecode but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block. timestamp works.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux “sha256sum" command against the target file.

About

DeHacker is a team of auditors and white hat hackers who perform security audits and assessments. With decades of experience in security and distributed systems, our experts focus on the ins and outs of system security. Our services follow clear and prudent industry standards. Whether it's reviewing the smallest modifications or a new platform, we'll provide an in-depth security survey at every stage of your company's project. We provide comprehensive vulnerability reports and identify structural inefficiencies in smart contract code, combining high-end security research with a real-world attacker mindset to reduce risk and harden code.

BLOCKCHAIINS

Ethereum	Cosmos
Eos	Substrate

TECH STACK

Python	Solidity
Rust	c++

CONTACTS

https://dehacker.io

https://twitter.com/dehackerio

https://github.com/dehacker/audits_public

https://t.me/dehackerio

https://blog.dehacker.io/

DeHacker

May 2023