// Private Smart Contract Security Assessment 07.22.2024 - 07.26.2024

Maha's Core

Zerolend

/ DRAFT /

HALBORN

Maha's Core - Zerolend

Prepared by: HALBORN

Last Updated 07/29/2024

Date of Engagement by: July 22nd, 2024 - July 26th, 2024

Summary

0% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	0	1	4

TABLE OF CONTENTS

1. Introduction

Zerolend engaged Halborn to conduct a security assessment on their smart contracts beginning on July 22nd and ending on July 26th. The security assessment was scoped to the smart contracts provided in the GitHub repository [mahaxyz](https://github.com/mahaxyz/), commit hashes, and further details can be found in the Scope section of this report.

The scope included in this assessment includes the core of the MAHA protocol which is focused on the creation of ZAI token USDz, a stable coin with the following characteristics:

- Fully pegged to USD fiat by minting 1:1 to already pegged USDC token, used as collateral, and assuring peg with a custom Peg Stability Module.
- USDz is a **LayerZero**'s OFT, which means that it will be able to be sent (minted in)to all blockchains supported by LayerZero.
- ZAI can be staked in a dedicated **Morpho Blue** trustless lending market.

2. Assessment Summary

The team at Halborn was provided 1 week for the engagement and assigned one full-time security engineer to check the security of the smart contract. The security engineer is a blockchain and smart-contract security experts with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

3. Test Approach And Methodology

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could led to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Static Analysis of security for scoped contract, and imported functions. (`Slither`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: mahaxyz

(b) Assessed Commit ID: f87db9e

(c) Items in scope:

- contracts/core/ZaiStablecoin.sol
- contracts/core/direct-deposit/Constants.sol
- contracts/core/direct-deposit/DDHub.sol
- contracts/core/direct-deposit/plans/DDOperatorPlan.sol
- contracts/core/direct-deposit/pools/DDBase.sol
- contracts/core/direct-deposit/pools/DDMetaMorpho.sol
- contracts/core/psm/PegStabilityModule.sol
- contracts/interfaces/IZaiStablecoin.sol
- contracts/interfaces/core/IDDHub.sol
- contracts/interfaces/core/IDDPlan.sol
- contracts/interfaces/core/IDDPool.sol
- contracts/interfaces/core/IPegStabilityModule.sol
- contracts/interfaces/events/DDEventsLib.sol
- contracts/interfaces/events/PSMEventsLib.sol
- contracts/interfaces/events/ZAIEventsLib.sol

Out-of-Scope: Third party dependencies, Economic attacks

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	1	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-02 - USE OF CONSTANT VALUE TO ABSORB ROUNDING ERRORS	LOW	-
HAL-04 - LACK OF ZERO ADDRESS CHECKS	INFORMATIONAL	-
HAL-01 - UNUSED IMPORTS	INFORMATIONAL	-
HAL-03 - UNUSED FUNCTION ARGUMENTS	INFORMATIONAL	-
HAL-05 - REDUNDANT CHECK IN BALANCE AND DEBT VERIFICATION	INFORMATIONAL	-

7. FINDINGS & TECH DETAILS

7.1 (HAL-02) USE OF CONSTANT VALUE TO ABSORB ROUNDING ERRORS

// LOW

Description

Utilizing the constant WAD (10^18) to absorb rounding errors is inappropriate for tokens like USDC, which operate with 6 decimal places (10^6). Applying WAD to USDC can lead to significant inaccuracies and out-of-range errors, as the precision mismatch can result in values that exceed the token's operational limits.

```
146     // It's adding up `WAD` due possible rounding errors
147     if (!info.isLive || !info.plan.active() || currentAssets + Constants.
148       toWithdraw = maxWithdraw;
149     } else {
150       uint256 maxDebt = info.debtCeiling; //vat.Line();
151       uint256 debt = info.debt;
152 ...
```

Therefore, when working with USDC, it's being created a range to absorb rounding errors of 10^12 USDC, which is undoubtedly an exaggerated range.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Creating/using a dynamic value to fit better the "rounding error" range is recommendable. This can be easily achievable by calling `token.decimals()` or storing this value in the pool info.

7.2 (HAL-04) LACK OF ZERO ADDRESS CHECKS

// INFORMATIONAL

Description

The `initialize` function in the `PegStabilityModule` lacks validation checks for zero addresses, potentially leading to critical issues. Initializing contracts with zero addresses can result in unexpected behavior or security vulnerabilities, as zero addresses may represent invalid or unintentional destinations. Implementing checks to ensure addresses are not zero is essential to maintain contract integrity, prevent errors, and enhance overall security.
Besides, there are no zero checks in the following functions either:

- `mint`
- `redeem`
- `updateFeeDestination`

Regarding the contract `DDHub.sol`:

- `initialize`
- `setFeeCollector`

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:H/D:N/Y:N/R:F/S:U (1.9)

Recommendation

Consider including pertinent address checks.

7.3 (HAL-01) UNUSED IMPORTS

// INFORMATIONAL

Description

Having unused imports in an Ethereum contract increases its bytecode size, leading to higher deployment costs and slower transaction processing.

```
import "../../../lib/forge-std/src/console.sol";
```

This unnecessary bloating not only wastes resources but also poses a security risk, as it increases the attack surface and complexity of the contract. Additionally, including unused imports can make the contract code harder to read and maintain, potentially leading to errors or vulnerabilities. Therefore, it's crucial to remove unused imports to optimize contract efficiency, reduce deployment costs, and mitigate security risks.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is recommended to delete the unused import.

7.4 (HAL-03) UNUSED FUNCTION ARGUMENTS

// INFORMATIONAL

Description

In the `DDOperatorPlan` contract we find that `getTargetAssets` has a not used argument.

```
36    function getTargetAssets(uint256) external view override returns (uint2
37      if (enabled == 0) return 0;
38      return targetAssets;
39    }
```

Besides, there's a call in contract DDHUB.sol that can be cleaned too:

```
152         uint256 targetAssets = info.plan.getTargetAssets(currentAssets);
```

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Unused arguments should be cleaned up from the code if they have no purpose. Clearing these arguments will increase the readability of the code.

7.5 (HAL-05) REDUNDANT CHECK IN BALANCE AND DEBT VERIFICATION

// INFORMATIONAL

Description

The provided code contains a redundant check that can be streamlined for efficiency. The `require` statement verifies that the balance plus 1 wei is greater than or equal to the debt, followed by an `if` statement that returns if the balance is less than or equal to the debt. The `require` statement makes the subsequent `if` check unnecessary because if the condition in `require` fails, the function execution is halted.

```
201 | require(balanceBefore + 1 wei >= info.debt, "invaraint balance >= debt");
202 |
203 | // calculate fees
204 | if (balanceBefore <= info.debt) return;
```

Score

AO:A/AC:L/AX:H/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

The require statement can be refined to remove redundancy, and the if check can be eliminated.

```
require(balanceBefore > info.debt, "invariant balance > debt");
```

8. AUTOMATED TESTING

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the scoped contracts. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their abi and binary formats, Slither was run on the all-scoped contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Slither Results

ERC20 Specific

```
# Check ZaiStablecoin

## Check functions
[✓] totalSupply() is present
        [✓] totalSupply() -> (uint256) (correct return type)
        [✓] totalSupply() is view
[✓] balanceOf(address) is present
        [✓] balanceOf(address) -> (uint256) (correct return type)
        [✓] balanceOf(address) is view
[✓] transfer(address,uint256) is present
        [✓] transfer(address,uint256) -> (bool) (correct return type)
        [✓] Transfer(address,address,uint256) is emitted
[✓] transferFrom(address,address,uint256) is present
        [✓] transferFrom(address,address,uint256) -> (bool) (correct return type)
        [✓] Transfer(address,address,uint256) is emitted
[✓] approve(address,uint256) is present
        [✓] approve(address,uint256) -> (bool) (correct return type)
        [✓] Approval(address,address,uint256) is emitted
[✓] allowance(address,address) is present
        [✓] allowance(address,address) -> (uint256) (correct return type)
        [✓] allowance(address,address) is view
[✓] name() is present
        [✓] name() -> (string) (correct return type)
        [✓] name() is view
[✓] symbol() is present
        [✓] symbol() -> (string) (correct return type)
        [✓] symbol() is view
[✓] decimals() is present
        [✓] decimals() -> (uint8) (correct return type)
        [✓] decimals() is view

## Check events
[✓] Transfer(address,address,uint256) is present
        [✓] parameter 0 is indexed
        [✓] parameter 1 is indexed
[✓] Approval(address,address,uint256) is present
        [✓] parameter 0 is indexed
        [✓] parameter 1 is indexed


        [ ] ZaiStablecoin is not protected for the ERC20 approval race condition
```

As `USDz` inherits `ERC20permit`, approvals aren't expected to be used, otherwise "ERC20 appoval race condition" should be taken into account.

Core Analysis

```
DDHub._sweepFees(IDDPool) (src/core/direct-deposit/DDHub.sol#197-217) ignores return value by zai.transfer(feeCollector,f
ees) (src/core/direct-deposit/DDHub.sol#215)
DDMetaMorpho.deposit(uint256) (src/core/direct-deposit/pools/DDMetaMorpho.sol#43-46) ignores return value by zai.transfer
From(msg.sender,address(this),wad) (src/core/direct-deposit/pools/DDMetaMorpho.sol#44)
PegStabilityModule.mint(address,uint256) (src/core/psm/PegStabilityModule.sol#87-98) ignores return value by collateral.t
ransferFrom(msg.sender,address(this),amount) (src/core/psm/PegStabilityModule.sol#93)
PegStabilityModule.redeem(address,uint256) (src/core/psm/PegStabilityModule.sol#101-110) ignores return value by zai.tran
sferFrom(msg.sender,address(this),shares) (src/core/psm/PegStabilityModule.sol#104)
PegStabilityModule.redeem(address,uint256) (src/core/psm/PegStabilityModule.sol#101-110) ignores return value by collater
al.transfer(dest,amount) (src/core/psm/PegStabilityModule.sol#106)
PegStabilityModule.sweepFees() (src/core/psm/PegStabilityModule.sol#112-114) ignores return value by collateral.transfer(
feeDestination,feesCollected()) (src/core/psm/PegStabilityModule.sol#113)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer

Reentrancy in DDHub._exec(IDDPool,IDDHub.PoolInfo) (src/core/direct-deposit/DDHub.sol#219-243):
        External calls:
        - _sweepFees(pool) (src/core/direct-deposit/DDHub.sol#221)
                - pool.withdraw(fees) (src/core/direct-deposit/DDHub.sol#208)
                - zai.transfer(feeCollector,fees) (src/core/direct-deposit/DDHub.sol#215)
        - pool.withdraw(toWithdraw) (src/core/direct-deposit/DDHub.sol#227)
        - zai.burn(address(this),toWithdraw) (src/core/direct-deposit/DDHub.sol#229)
        - zai.mint(address(this),toSupply) (src/core/direct-deposit/DDHub.sol#235)
        - pool.deposit(toSupply) (src/core/direct-deposit/DDHub.sol#236)
        State variables written after the call(s):
        - _updatePoolInfo(pool,info) (src/core/direct-deposit/DDHub.sol#242)
                - _poolInfos[_pool] = _info (src/core/direct-deposit/DDHub.sol#246)
        DDHub._poolInfos (src/core/direct-deposit/DDHub.sol#51) can be used in cross function reentrancies:
        - DDHub._sweepFees(IDDPool) (src/core/direct-deposit/DDHub.sol#197-217)
        - DDHub._updatePoolInfo(IDDPool,IDDHub.PoolInfo) (src/core/direct-deposit/DDHub.sol#245-248)
        - DDHub.evaluatePoolAction(IDDPool) (src/core/direct-deposit/DDHub.sol#136-180)
        - DDHub.isPool(address) (src/core/direct-deposit/DDHub.sol#76-78)
        - DDHub.poolInfos(IDDPool) (src/core/direct-deposit/DDHub.sol#71-73)
        - DDHub.reduceDebtCeiling(IDDPool,uint256) (src/core/direct-deposit/DDHub.sol#101-105)
        - DDHub.setDebtCeiling(IDDPool,uint256) (src/core/direct-deposit/DDHub.sol#108-112)
        - DDHub.shutdownPool(IDDPool) (src/core/direct-deposit/DDHub.sol#125-129)
Reentrancy in DDHub.exec(IDDPool) (src/core/direct-deposit/DDHub.sol#81-91):
        External calls:
        - pool.preDebtChange() (src/core/direct-deposit/DDHub.sol#85)
        - _exec(pool,info) (src/core/direct-deposit/DDHub.sol#88)
                - pool.withdraw(toWithdraw) (src/core/direct-deposit/DDHub.sol#227)
                - pool.withdraw(fees) (src/core/direct-deposit/DDHub.sol#208)
                - zai.burn(address(this),toWithdraw) (src/core/direct-deposit/DDHub.sol#229)
                - zai.transfer(feeCollector,fees) (src/core/direct-deposit/DDHub.sol#215)
                - zai.mint(address(this),toSupply) (src/core/direct-deposit/DDHub.sol#235)
                - pool.deposit(toSupply) (src/core/direct-deposit/DDHub.sol#236)
        State variables written after the call(s):
        - _exec(pool,info) (src/core/direct-deposit/DDHub.sol#88)
                - _poolInfos[_pool] = _info (src/core/direct-deposit/DDHub.sol#246)
        DDHub._poolInfos (src/core/direct-deposit/DDHub.sol#51) can be used in cross function reentrancies:
        - DDHub._sweepFees(IDDPool) (src/core/direct-deposit/DDHub.sol#197-217)
        - DDHub._updatePoolInfo(IDDPool,IDDHub.PoolInfo) (src/core/direct-deposit/DDHub.sol#245-248)
        - DDHub.evaluatePoolAction(IDDPool) (src/core/direct-deposit/DDHub.sol#136-180)
        - DDHub.isPool(address) (src/core/direct-deposit/DDHub.sol#76-78)
        - DDHub.poolInfos(IDDPool) (src/core/direct-deposit/DDHub.sol#71-73)
        - DDHub.reduceDebtCeiling(IDDPool,uint256) (src/core/direct-deposit/DDHub.sol#101-105)
        - DDHub.setDebtCeiling(IDDPool,uint256) (src/core/direct-deposit/DDHub.sol#108-112)
        - DDHub.shutdownPool(IDDPool) (src/core/direct-deposit/DDHub.sol#125-129)
Reentrancy in PegStabilityModule.mint(address,uint256) (src/core/psm/PegStabilityModule.sol#87-98):
        External calls:
        - collateral.transferFrom(msg.sender,address(this),amount) (src/core/psm/PegStabilityModule.sol#93)
        - zai.mint(dest,shares) (src/core/psm/PegStabilityModule.sol#94)
        State variables written after the call(s):
        - debt += shares (src/core/psm/PegStabilityModule.sol#96)
        PegStabilityModule.debt (src/core/psm/PegStabilityModule.sol#43) can be used in cross function reentrancies:
        - PegStabilityModule.debt (src/core/psm/PegStabilityModule.sol#43)
        - PegStabilityModule.feesCollected() (src/core/psm/PegStabilityModule.sol#152-154)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

```
DDHub.registerPool(IDDPool,IDDPlan,uint256) (src/core/direct-deposit/DDHub.sol#94-98) ignores return value by zai.approve
(address(pool),type()(uint256).max) (src/core/direct-deposit/DDHub.sol#97)
DDMetaMorpho.initialize(address,address,address) (src/core/direct-deposit/pools/DDMetaMorpho.sol#28-39) ignores return va
lue by zai.approve(_vault,type()(uint256).max) (src/core/direct-deposit/pools/DDMetaMorpho.sol#38)
DDMetaMorpho.deposit(uint256) (src/core/direct-deposit/pools/DDMetaMorpho.sol#43-46) ignores return value by vault.deposi
t(wad,address(this)) (src/core/direct-deposit/pools/DDMetaMorpho.sol#45)
DDMetaMorpho.withdraw(uint256) (src/core/direct-deposit/pools/DDMetaMorpho.sol#50-52) ignores return value by vault.withd
raw(wad,msg.sender,address(this)) (src/core/direct-deposit/pools/DDMetaMorpho.sol#51)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return

DDHub.initialize(address,uint256,address,address)._feeCollector (src/core/direct-deposit/DDHub.sol#55) lacks a zero-check
 on :
                - feeCollector = _feeCollector (src/core/direct-deposit/DDHub.sol#61)
DDHub.setFeeCollector(address).who (src/core/direct-deposit/DDHub.sol#115) lacks a zero-check on :
                - feeCollector = who (src/core/direct-deposit/DDHub.sol#116)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

Reentrancy in PegStabilityModule.redeem(address,uint256) (src/core/psm/PegStabilityModule.sol#101-110):
        External calls:
        - zai.transferFrom(msg.sender,address(this),shares) (src/core/psm/PegStabilityModule.sol#104)
        - zai.burn(address(this),shares) (src/core/psm/PegStabilityModule.sol#105)
        - collateral.transfer(dest,amount) (src/core/psm/PegStabilityModule.sol#106)
        State variables written after the call(s):
        - debt -= shares (src/core/psm/PegStabilityModule.sol#108)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2

Reentrancy in DDHub._sweepFees(IDDPool) (src/core/direct-deposit/DDHub.sol#197-217):
        External calls:
        - pool.withdraw(fees) (src/core/direct-deposit/DDHub.sol#208)
        - zai.transfer(feeCollector,fees) (src/core/direct-deposit/DDHub.sol#215)
        Event emitted after the call(s):
        - DDEventsLib.Fees(pool,fees) (src/core/direct-deposit/DDHub.sol#216)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
. analyzed (101 contracts with 68 detectors), 17 result(s) found
```

The reentrancy instances flagged by Slither were checked individually, and have been categorised as false positives. The ones that may be vulnerable are all protected with the nonReentrant modifier.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.