// Private Smart Contract Security Assessment

06.24.2024 - 06.27.2024

Inception LRT

Tagus Labs

HALBORN

Inception LRT - Tagus Labs

Prepared by: HALBORN Last Updated 06/28/2024

Date of Engagement by: June 24th, 2024 - June 27th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
8	0	0	0	1	7

1. Introduction

The **Tagus Labs** team engaged Halborn to conduct a security assessment on their smart contracts beginning on **06-24-2024** and ending on **06-27-2024**. The security assessment was scoped to the smart contracts provided in the **https://github.com/inceptionlrt/smart-contracts** GitHub repository. Commit hashes and further details can be found in the Scope section of this report. The **Inceptionlrt/Smart Contracts** codebase in scope mainly consists of **native and isolated liquid restaking for the Eigenlayer ecosystem.**

TABLE OF CONTENTS

2. Assessment Summary

Halborn was provided 4 days for the engagement and assigned 1 full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the Tagus Labs team. The main identified issue was:

- **Missing fee amounts verification.**

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 this 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 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.
- Local testing with custom scripts (Foundry).
- Fork testing against main networks (Foundry).
- Static analysis of security for scoped contract, and imported functions (Slither).

Out-Of-Scope

- External libraries and financial-related attacks.
- New features/implementations after/within the **remediation commit IDs**.
- Changes that occur outside of the scope of PRs.
- The following contracts, while they affect the in-scope code, were not specifically reviewed as part of the scope and are assumed to not contain any issues:
    - EigenLayerHandler.sol
    - InceptionAssetsHandler.sol
    - RatioFeed.sol
    - InceptionToken.sol
    - DelegationManager.sol

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
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

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
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
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: smart-contracts

(b) Assessed Commit ID: 8a9c4a6

(c) Items in scope:

- Flash withdrawal functionality from contracts/Inception/lib/InceptionLibrary.sol
- contracts/Inception/vaults/InceptionVault.sol

Out-of-Scope: contracts/Inception/eigenlayer-handler/EigenLayerHandler.sol, contracts/Inception/lib/Convert.sol, contracts/Inception/assets-handler/InceptionAssetsHandler.sol, contracts/Inception/restaker/InceptionRestaker.sol, contracts/Inception/tokens/InceptionToken.sol, contracts/Inception/vaults/vault_e1/InVault_E1.sol, contracts/Inception/vaults/vault_e2/InVault_E2.sol, contracts/TimeLock/InceptionTimeLock.sol, contracts/InceptionRateProvider, contracts/tests

REMEDIATION COMMIT ID:

- 0229f480229f48

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	1

INFORMATIONAL
7

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING FEE AMOUNTS VERIFICATION	LOW	SOLVED - 06/26/2024
NON-REENTRANT MODIFIER ORDERING	INFORMATIONAL	ACKNOWLEDGED
OPEN TO-DO	INFORMATIONAL	SOLVED - 06/26/2024
INCOMPATIBILITY RISK FOR EVM VERSIONS IN DIFFERENT CHAINS	INFORMATIONAL	ACKNOWLEDGED

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
FLOATING PRAGMA	INFORMATIONAL	ACKNOWLEDGED
UNUSED IMPORT	INFORMATIONAL	SOLVED - 06/26/2024
DISPENSABLE GAS FESS FROM PUBLIC FUNCTIONS NOT CALLED WITHIN CONTRACT	INFORMATIONAL	SOLVED - 06/26/2024
DISPENSABLE GAS FEES FROM UNOPTIMIZED LOOPS	INFORMATIONAL	ACKNOWLEDGED

7. FINDINGS & TECH DETAILS

7.1 MISSING FEE AMOUNTS VERIFICATION

// LOW

Description

In the `flashWithdraw()` function from the `InceptionVault` contract, the `fee` and `protocolWithdrawalFee` variables are not verified to be greater than zero. Although with the current implementation, it is highly unlikely to have a fee value of zero (for example, in cases where the `targetCapacity` value is set to a very low value), this oversight could allow a user the execution of flash withdrawals without paying the expected fee.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (2.5)

Recommendation

It is recommended to add a check to ensure that the `fee` and `protocolWithdrawalFee` variables are greater than zero before proceeding with the flash withdrawal.

Remediation Plan

SOLVED: The **Tagus Labs team** has addressed the finding in commit `0229f48e38eacf23baa53c33190c968c8d2e324f` by following the mentioned recommendation.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/0229f48e38eacf23baa53c33190c968c8d2e324f

References

InceptionVault.sol#L287-L314

7.2 NON-REENTRANT MODIFIER ORDERING

// INFORMATIONAL

Description

In Solidity, if a function has multiple modifiers they are executed in the order specified. If checks or logic of modifiers depend on other modifiers this has to be considered in their ordering.
The `flashWithdraw()`, `redeem()` and `withdraw()` functions from the `InceptionVault` contract have multiple modifiers with one of them being nonreentrant which prevents reentrancy on the functions. This should ideally be the first one to prevent even the execution of other modifiers in case of reentrancies.
While there is currently no obvious vulnerability with nonreentrant being the last modifier in the list, placing it first ensures that all other modifiers are executed only if the call is non-reentrant. This is a safer practice and can prevent potential issues in future updates or unforeseen scenarios.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (0.5)

Recommendation

Switch modifier order to consistently place the `nonReentrant` modifier as the first one to run so that all other modifiers are executed only if the call is non-reentrant.

Remediation Plan

ACKNOWLEDGED: The **Tagus Labs team** made a business decision to acknowledge this finding and not alter the contracts.

References

InceptionVault.sol#L221
InceptionVault.sol#L245

InceptionVault.sol#L290

7.3 OPEN TO-DO

// INFORMATIONAL

Description

The `__InceptionVault_init()` function from the `InceptionVault` contract contains a `TODO` comment that may indicates that the function has not been not completely implemented. This incomplete implementation can lead to unexpected behavior and/or potential vulnerabilities within the contract.

```
minAmount = 100;

/// @notice TODO

protocolFee = 50 * 1e8;
```

Score

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

Recommendation

Ensure that all TODO comments are resolved and the function is fully implemented.

Remediation Plan

SOLVED: The **Tagus Labs team** has addressed the finding in commit `0229f48e38eacf23baa53c33190c968c8d2e324f` by following the mentioned recommendation.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/0229f48e38eacf23baa53c33190c968c8d2e324f

References

InceptionVault.sol#L69

7.4 INCOMPATIBILITY RISK FOR EVM VERSIONS IN DIFFERENT CHAINS

// INFORMATIONAL

Description

From Solidity versions `0.8.20` through `0.8.24`, the default target EVM version is set to Shanghai, which results in the generation of bytecode that includes `PUSH0` opcodes. Starting with version `0.8.25`, the default EVM version shifts to Cancun, introducing new opcodes for transient storage, `TSTORE` and `TLOAD`.
In this aspect, it is crucial to select the appropriate EVM version when it's intended to deploy the contracts on networks other than the Ethereum mainnet, which may not support these opcodes. Failure to do so could lead to unsuccessful contract deployments or transaction execution issues.

Score

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

Recommendation

Make sure to specify the target EVM version when using Solidity versions from `0.8.20` and above if deploying to chains that may not support newly introduced opcodes. Additionally, it is crucial to stay informed about the opcode support of different chains to ensure smooth deployment and compatibility.

Remediation Plan

ACKNOWLEDGED: The **Tagus Labs team** made a business decision to acknowledge this finding and not alter the contracts.

7.5 FLOATING PRAGMA

// INFORMATIONAL

Description

The files in scope currently use floating pragma version `^0.8.22`, which means that the code can be compiled by any compiler version that is greater than or equal to `0.8.22`, and less than `0.9.0`. However, it is recommended that contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

Score

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

Recommendation

Lock the pragma version to the same version used during development and testing.

Remediation Plan

ACKNOWLEDGED: The **Tagus Labs team** made a business decision to acknowledge this finding and not alter the contracts.

References

7.6 UNUSED IMPORT

// INFORMATIONAL

Description

The `InceptionVault` contract imports the `IRebalanceStrategy` interface but does not use it in the contract. This can be considered as unnecessary code and should be removed to keep the code clean and easy to read.

Score

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

Recommendation

Remove the unused `IRebalanceStrategy` import statement.

Remediation Plan

SOLVED: The **Tagus Labs** team has addressed the finding in commit `0229f48e38eacf23baa53c33190c968c8d2e324f` by following the mentioned recommendation.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/0229f48e38eacf23baa53c33190c968c8d2e324f

References

InceptionVault.sol#L9

7.7 DISPENSABLE GAS FESS FROM PUBLIC FUNCTIONS NOT CALLED WITHIN CONTRACT

// INFORMATIONAL

Description

Several functions throughout the files in scope currently define several functions with the `public` visibility modifier, even though the functions are not called from within the contract, possibly resulting in higher gas costs than necessary.
In the case of Solidity `public` functions, arguments of the functions are copied to `memory`. While on the other hand, functions with `external` visibility can directly read arguments from `calldata`. Since `calldata` is cheaper to access than `memory`, external functions result in a lower execution cost than public functions.
The affected functions are:

- `deposit()`
- `depositWithReferral()`
- `redeem()`
- `getTotalDeposited()`
- `getDelegatedTo()`
- `getPendingWithdrawalOf()`
- `maxDeposit()`
- `maxRedeem()`

Score

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

Recommendation

Modify the aforementioned functions with the `external` visibility modifier.

Remediation Plan

SOLVED: The **Tagus Labs team** has addressed the finding in commit `0229f48e38eacf23baa53c33190c968c8d2e324f` by following the mentioned recommendation.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/0229f48e38eacf23baa53c33190c968c8d2e324f

References

InceptionVault.sol#L108
InceptionVault.sol#L117
InceptionVault.sol#L245
InceptionVault.sol#L418
InceptionVault.sol#L435
InceptionVault.sol#L441
InceptionVault.sol#L461
InceptionVault.sol#L468
InceptionVault.sol#L479

7.8 DISPENSABLE GAS FEES FROM UNOPTIMIZED LOOPS

// INFORMATIONAL

Description

Throughout the code in scope, there are several instances of unoptimized `for` loop declarations that may incur in higher gas costs than necessary. The affected functions are:

- `redeem()`
- `isAbleToRedeem()`
- `getTotalDelegated()`
- `_verifyDelegated()`

Score

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

Recommendation

Optimize the `for` loop declarations to reduce gas costs. Best practices include: the non-redundant initialization of the iterator with a default value (`i = 0` instead of simply `i`), the use of the pre-increment operator (`++i`). Additionally, when reading from storage variables, it is recommended to reduce gas costs significantly by caching the array to read locally and iterate over it to avoid reading from storage on every iteration:
Moreover, if there are several loops in the same function, the `i` variable can be re-used, to be able to set the value from non-zero to zero and reduce gas costs without additional variable declaration. For example:

```
uint256[] memory arrayInMemory = arrayInStorage;

uint256 i;
for (; i < arrayInMemory.length ;) {
  // code logic
  unchecked { ++i; }
}

delete i;

uint256[] memory arrayInMemory2 = arrayInStorage2;

for (; i < arrayInMemory2.length ;) {
  // code logic
  unchecked { ++i; }
}
```

Remediation Plan

ACKNOWLEDGED: The **Tagus Labs team** made a business decision to acknowledge this finding and not alter the contracts.

References

InceptionVault.sol#L256-L269
InceptionVault.sol#L396-L402
InceptionVault.sol#L428-L431
InceptionVault.sol#L446-L451

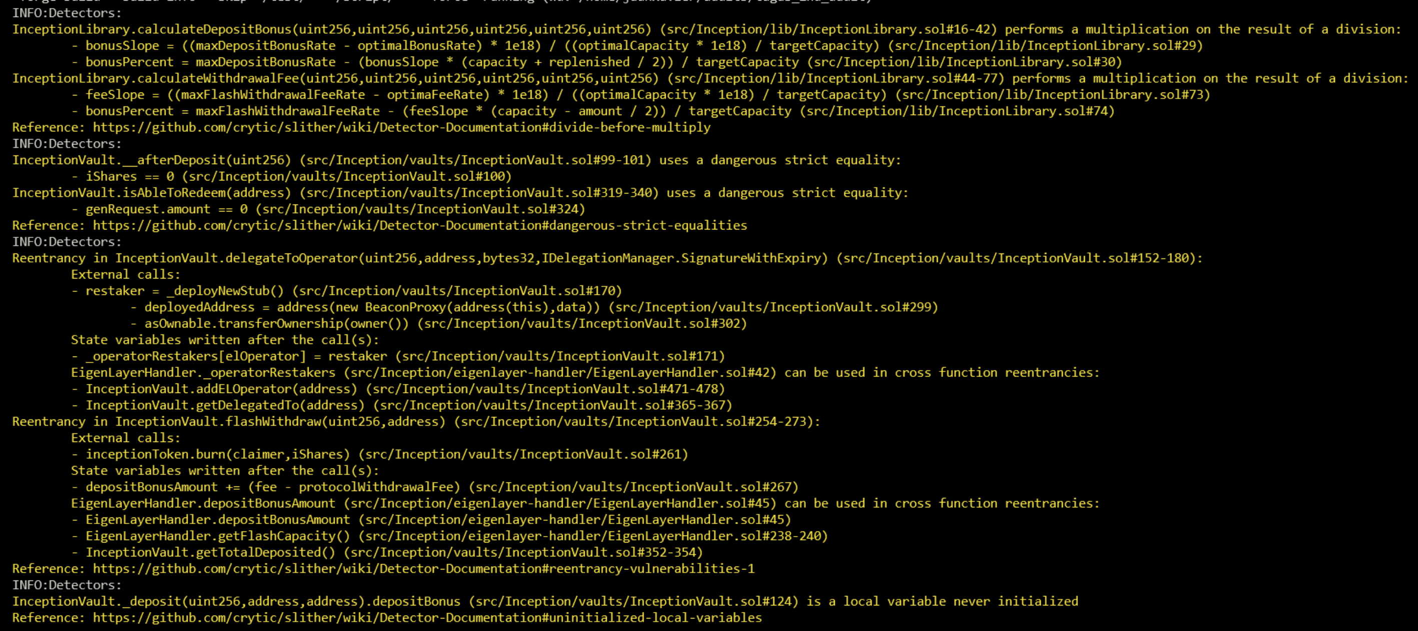
STATIC ANALYSIS REPORT

Description

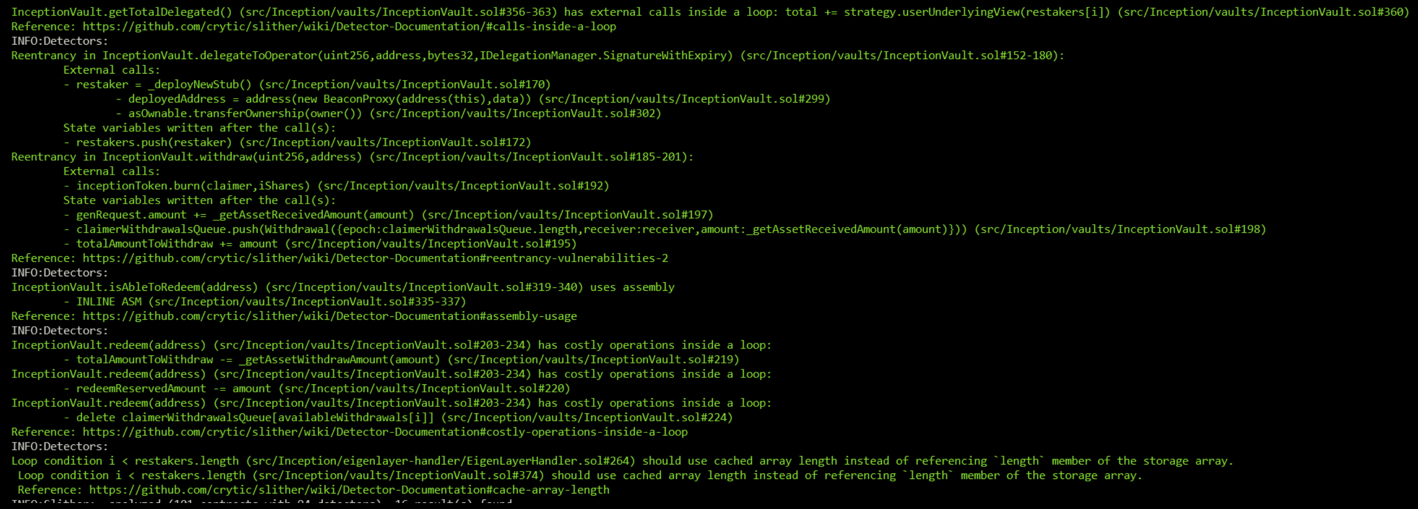
Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the 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.
The security team assessed all findings identified by the Slither software, however, findings with related to external dependencies are not included in the below results for the sake of report readability.

Output

The findings obtained as a result of the Slither scan were reviewed, and most of them were not included in the report because they were determined as false positives.



```
INFO:Detectors:
InceptionLibrary.calculateDepositBonus(uint256,uint256,uint256,uint256,uint256,uint256) (src/Inception/lib/InceptionLibrary.sol#16-42) performs a multiplication on the result of a division:
        - bonusSlope = ((maxDepositBonusRate - optimalBonusRate) * 1e18) / ((optimalCapacity * 1e18) / targetCapacity) (src/Inception/lib/InceptionLibrary.sol#29)
        - bonusPercent = maxDepositBonusRate - (bonusSlope * (capacity + replenished / 2)) / targetCapacity (src/Inception/lib/InceptionLibrary.sol#30)
InceptionLibrary.calculateWithdrawalFee(uint256,uint256,uint256,uint256,uint256,uint256) (src/Inception/lib/InceptionLibrary.sol#44-77) performs a multiplication on the result of a division:
        - feeSlope = ((maxFlashWithdrawalFeeRate - optimaFeeRate) * 1e18) / ((optimalCapacity * 1e18) / targetCapacity) (src/Inception/lib/InceptionLibrary.sol#73)
        - bonusPercent = maxFlashWithdrawalFeeRate - (feeSlope * (capacity - amount / 2)) / targetCapacity (src/Inception/lib/InceptionLibrary.sol#74)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
InceptionVault.__afterDeposit(uint256) (src/Inception/vaults/InceptionVault.sol#99-101) uses a dangerous strict equality:
        - iShares == 0 (src/Inception/vaults/InceptionVault.sol#100)
InceptionVault.isAbleToRedeem(address) (src/Inception/vaults/InceptionVault.sol#319-340) uses a dangerous strict equality:
        - genRequest.amount == 0 (src/Inception/vaults/InceptionVault.sol#324)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Reentrancy in InceptionVault.delegateToOperator(uint256,address,bytes32,IDelegationManager.SignatureWithExpiry) (src/Inception/vaults/InceptionVault.sol#152-180):
        External calls:
        - restaker = _deployNewStub() (src/Inception/vaults/InceptionVault.sol#170)
                - deployedAddress = address(new BeaconProxy(address(this),data)) (src/Inception/vaults/InceptionVault.sol#299)
                - asOwnable.transferOwnership(owner()) (src/Inception/vaults/InceptionVault.sol#302)
        State variables written after the call(s):
        - _operatorRestakers[elOperator] = restaker (src/Inception/vaults/InceptionVault.sol#171)
        EigenLayerHandler._operatorRestakers (src/Inception/eigenlayer-handler/EigenLayerHandler.sol#42) can be used in cross function reentrancies:
        - InceptionVault.addELOperator(address) (src/Inception/vaults/InceptionVault.sol#471-478)
        - InceptionVault.getDelegatedTo(address) (src/Inception/vaults/InceptionVault.sol#365-367)
Reentrancy in InceptionVault.flashWithdraw(uint256,address) (src/Inception/vaults/InceptionVault.sol#254-273):
        External calls:
        - inceptionToken.burn(claimer,iShares) (src/Inception/vaults/InceptionVault.sol#261)
        State variables written after the call(s):
        - depositBonusAmount += (fee - protocolWithdrawalFee) (src/Inception/vaults/InceptionVault.sol#267)
        EigenLayerHandler.depositBonusAmount (src/Inception/eigenlayer-handler/EigenLayerHandler.sol#45) can be used in cross function reentrancies:
        - EigenLayerHandler.depositBonusAmount (src/Inception/eigenlayer-handler/EigenLayerHandler.sol#45)
        - EigenLayerHandler.getFlashCapacity() (src/Inception/eigenlayer-handler/EigenLayerHandler.sol#238-240)
        - InceptionVault.getTotalDeposited() (src/Inception/vaults/InceptionVault.sol#352-354)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
InceptionVault._deposit(uint256,address,address).depositBonus (src/Inception/vaults/InceptionVault.sol#124) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```



```
InceptionVault.getTotalDelegated() (src/Inception/vaults/InceptionVault.sol#356-363) has external calls inside a loop: total += strategy.userUnderlyingView(restakers[i]) (src/Inception/vaults/InceptionVault.sol#360)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop
INFO:Detectors:
Reentrancy in InceptionVault.delegateToOperator(uint256,address,bytes32,IDelegationManager.SignatureWithExpiry) (src/Inception/vaults/InceptionVault.sol#152-180):
        External calls:
        - restaker = _deployNewStub() (src/Inception/vaults/InceptionVault.sol#170)
                - deployedAddress = address(new BeaconProxy(address(this),data)) (src/Inception/vaults/InceptionVault.sol#299)
                - asOwnable.transferOwnership(owner()) (src/Inception/vaults/InceptionVault.sol#302)
        State variables written after the call(s):
        - restakers.push(restaker) (src/Inception/vaults/InceptionVault.sol#172)
Reentrancy in InceptionVault.withdraw(uint256,address) (src/Inception/vaults/InceptionVault.sol#185-201):
        External calls:
        - inceptionToken.burn(claimer,iShares) (src/Inception/vaults/InceptionVault.sol#192)
        State variables written after the call(s):
        - genRequest.amount += _getAssetReceivedAmount(amount) (src/Inception/vaults/InceptionVault.sol#197)
        - claimerWithdrawalsQueue.push(Withdrawal({epoch:claimerWithdrawalsQueue.length,receiver:receiver,amount:_getAssetReceivedAmount(amount)})) (src/Inception/vaults/InceptionVault.sol#198)
        - totalAmountToWithdraw += amount (src/Inception/vaults/InceptionVault.sol#195)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
InceptionVault.isAbleToRedeem(address) (src/Inception/vaults/InceptionVault.sol#319-340) uses assembly
        - INLINE ASM (src/Inception/vaults/InceptionVault.sol#335-337)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
InceptionVault.redeem(address) (src/Inception/vaults/InceptionVault.sol#203-234) has costly operations inside a loop:
        - totalAmountToWithdraw -= _getAssetWithdrawAmount(amount) (src/Inception/vaults/InceptionVault.sol#219)
InceptionVault.redeem(address) (src/Inception/vaults/InceptionVault.sol#203-234) has costly operations inside a loop:
        - redeemReservedAmount -= amount (src/Inception/vaults/InceptionVault.sol#220)
InceptionVault.redeem(address) (src/Inception/vaults/InceptionVault.sol#203-234) has costly operations inside a loop:
        - delete claimerWithdrawalsQueue[availableWithdrawals[i]] (src/Inception/vaults/InceptionVault.sol#224)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#costly-operations-inside-a-loop
INFO:Detectors:
Loop condition i < restakers.length (src/Inception/eigenlayer-handler/EigenLayerHandler.sol#264) should use cached array length instead of referencing `length` member of the storage array.
 Loop condition i < restakers.length (src/Inception/vaults/InceptionVault.sol#374) should use cached array length instead of referencing `length` member of the storage array.
 Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#cache-array-length
```

UNIT TESTS AND FUZZING

The original repository used the **Hardhat** environment to develop and test the smart contracts. Additionally, the project in scope was cloned to a **Foundry** environment, to allow for additional testing and fuzz testing that covered ~1,000,000 runs per test. These additional tests were run successfully.

```
Ran 8 tests for test/InceptionVault.t.sol:InceptionTest
[PASS] test_ZeroFeeFlashWithdraw() (gas: 371278)
[PASS] test_flashWithdraw() (gas: 448972)
[PASS] test_flashWithdrawAndDeposit() (gas: 499776)
[PASS] test_flashWithdrawFuzz(uint256,uint256,uint256) (runs: 1000011, μ: 461234, ~: 463880)
[PASS] test_setDepositBonusParams() (gas: 35256)
[PASS] test_setFlashWithdrawFeeParams() (gas: 35215)
[PASS] test_setProtocolFee() (gas: 25827)
[PASS] test_setTargetFlashCapacity() (gas: 42664)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 1007.59s (1007.59s CPU time)

Ran 4 tests for test/InceptionLibrary.t.sol:InceptionLibraryTest
[PASS] test_calculateDepositBonus() (gas: 31991)
[PASS] test_calculateDepositBonusFuzz(uint256,uint256,uint128) (runs: 1000011, μ: 21575, ~: 15124)
[PASS] test_calculateWithdrawalFee() (gas: 17156)
[PASS] test_calculateWithdrawalFeeFuzz(uint256,uint256,uint256,uint256) (runs: 1000011, μ: 23622, ~: 22986)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 1007.60s (471.81s CPU time)

Ran 2 test suites in 1007.65s (2015.19s CPU time): 12 tests passed, 0 failed, 0 skipped (12 total tests)
```

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.