HALBORN

MetaPool - ETH Staking

Smart Contract Security Audit

Prepared by: **Halborn**

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

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY 4
CONTACTS 5
1 EXECUTIVE OVERVIEW 6
1.1 INTRODUCTION 7
1.2 AUDIT SUMMARY 7
1.3 TEST APPROACH & METHODOLOGY 7
2 RISK METHODOLOGY 9
2.1 EXPLOITABILITY 10
2.2 IMPACT 11
2.3 SEVERITY COEFFICIENT 13
2.4 SCOPE 15
3 ASSESSMENT SUMMARY & FINDINGS OVERVIEW 16
4 FINDINGS & TECH DETAILS 17
4.1 (HAL-01) MINIMUM DEPOSIT RESTRICTION CAN BE BYPASSED - MEDIUM(5.6) 19
Description 19
Code Location 19
BVSS 20
Proof of Concept 21
Recommendation 22
Remediation Plan 22
4.2 (HAL-02) ERC4626 VAULT DEPOSITS AND WITHDRAWS SHOULD CONSIDER SLIPPAGE - LOW(3.4) 23
Description 23
Code Location 23

BVSS 24
Recommendation 24
References 25
Remediation Plan 25
4.3 (HAL-03) VAULTS ARE NOT EIP-4626 COMPLIANT - LOW(2.5) 26
Description 26
Code Location 27
BVSS 27
Recommendation 27
References 27
Remediation Plan 27
4.4 (HAL-04) USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS - INFORMATIONAL(0.0) 29
Description 29
BVSS 29
Recommendation 29
Remediation Plan 30
4.5 (HAL-05) USE UINT256 INSTEAD OF UINT IN FUNCTION ARGUMENTS - INFORMATIONAL(0.0) 31
Description 31
BVSS 31
Recommendation 31
Remediation Plan 31
4.6 (HAL-06) LOOP GAS USAGE OPTIMIZATION - INFORMATIONAL(0.0) 32
Description 32
Code Location 32
BVSS 32

Recommendation 33
Remediation Plan 33
4.7 (HAL-07) FLOATING PRAGMA - INFORMATIONAL(0.0) 34
Description 34
Risk Level 34
Recommendation 34
Remediation Plan 34
4.8 (HAL-08) TYPOS IN COMMENTS - INFORMATIONAL(0.0) 35
Description 35
Code Location 35
BVSS 35
Recommendation 35
Remediation Plan 35
5 AUTOMATED TESTING 36
5.1 STATIC ANALYSIS REPORT 37
Description 37
Results 38
5.2 AUTOMATED SECURITY SCAN 40
Description 40
MythX results 40
6 APPENDIX 41
Deployment contract used for MetaPool ETH testing 42

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 successfully 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:

EXECUTIVE OVERVIEW

- 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) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

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) 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

2.4 SCOPE

IN-SCOPE CODE & COMMITS:

- Repository: metapool-ethereum
  - Commit ID: f0833b091124e26e18393f53dabc15d658dcad84
  - Smart contracts **in scope**:
    - All smart contracts under /contracts folder.

REMEDIATION COMMITS:

- Repository: metapool-ethereum
  - Commit IDs:
    - 79f910ea4f79ba108d21c2c67eb9b59478c2e7c0
    - 6b4e6770d840a8b90d3bda6ef31fb5de2665d753
    - d6f739a7064ccfe965adb21ea498bcc1d5bb28ef
    - c86bac226b5cf581724b368385999cddda4e0bda
    - 09e5810f590ecb890d914b42bfe6f7d8d085643a
    - f75a74db30d6ad74b7f78af95aabecde315967aa
    - 2150d0bf5d3cd8194bf03802d64b2e7a6cb1526c

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	2	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MINIMUM DEPOSIT RESTRICTION CAN BE BYPASSED	Medium (5.6)	SOLVED - 06/09/2023
ERC4626 VAULT DEPOSITS AND WITHDRAWS SHOULD CONSIDER SLIPPAGE	Low (3.4)	SOLVED - 06/05/2023
VAULTS ARE NOT EIP-4626 COMPLIANT	Low (2.5)	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
FLOATING PRAGMA	Informational (0.0)	SOLVED - 06/09/2023
TYPOS IN COMMENTS	Informational (0.0)	SOLVED - 06/09/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) MINIMUM DEPOSIT RESTRICTION CAN BE BYPASSED - MEDIUM (5.6)

Description:

LiquidUnstakePool and Staking smart contracts allow to deposit/withdraw tokens by using ERC4626 custom vaults. In this implementation, a modifier is involved in deposits since it guarantees a minimum amount of tokens in each deposit. This modifier should be applied to each function related to deposit.

However, there is existing a public function named mint which allows specifying an amount of shares to mint instead of an amount of tokens to deposit as deposit functions do. This function is not restricted by the aforementioned modifier and allows to mint arbitrary amount of shares without restrictions, thus breaking the invariant set by validDeposit modifier.

Code Location:

Listing 1: contracts/LiquidUnstakePool.sol

```
58 modifier validDeposit(uint _amount) {
59     require(_amount >= MIN_DEPOSIT, "Deposit at least 0.01 ETH");
60     _;
61 }
```

Listing 2: contracts/LiquidUnstakePool.sol (Line 134)

```
129 function _deposit(
130     address _caller,
131     address _receiver,
132     uint _assets,
133     uint _shares
134 ) internal virtual override nonReentrant {
135     _assets = _getAssetsDeposit(_assets);
```

```
136     _mint(_receiver, _shares);
137     ethBalance += _assets;
138     emit AddLiquidity(_caller, _receiver, _assets, _shares);
139 }
```

Listing 3: contracts/Staking.sol

```
61 modifier validDeposit(uint _amount) {
62     require(_amount >= MIN_DEPOSIT, "Deposit at least 0.01 ETH");
63     _;
64 }
```

Listing 4: contracts/Staking.sol (Line 270)

```
265 function _deposit(
266     address _caller,
267     address _receiver,
268     uint256 _assets,
269     uint256 _shares
270 ) internal override checkWhitelisting() {
271     _assets = _getAssetsDeposit(_assets);
272     (uint sharesFromPool, uint assetsToPool) = _getmpETHFromPool(
 ↳ _shares, _receiver);
273     _shares -= sharesFromPool;
274     _assets -= assetsToPool;
275 
276     if (_shares > 0) _mint(_receiver, _shares);
277 
278     stakingBalance += _assets;
279     emit Deposit(_caller, _receiver, _assets + assetsToPool,
 ↳ _shares + sharesFromPool);
280 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:L/Y:N/R:N/S:U (5.6)

Proof of Concept:

In order to prove this issue, since the mint function is available, an account just has to perform a call with an amount lower than 0.1 ether as these values are supposed to be restricted.

Listing 5: Minimum deposit restriction bypass

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

    console.log("[1] LiquidUnstakePool balance (ETH):", address(
↳ liquidunstakepool).balance);
    console.log("[1] Attacker balance (shares):",
↳ IERC20MetadataUpgradeable(liquidunstakepool).balanceOf(ALICE));
    console.log("");

    vm.startPrank(ALICE);
    {
        weth.approve(address(liquidunstakepool), 1);
        liquidunstakepool.mint(1, ALICE);
    }

    console.log("[2] LiquidUnstakePool balance (ETH):", address(
↳ liquidunstakepool).balance);
    console.log("[2] Attacker balance (shares):",
↳ IERC20MetadataUpgradeable(liquidunstakepool).balanceOf(ALICE));
}

function prepareBalances() public {
    vm.deal(ALICE, 200 ether);
    vm.deal(BOB, 200 ether);
    vm.deal(CHARLIE, 200 ether);

    weth.mint(ALICE, 200 ether);
    weth.mint(BOB, 200 ether);
    weth.mint(CHARLIE, 200 ether);
}
```

```
[PASS] testMintWithoutRestrictions() (gas: 229619)
Logs:
  [1] LiquidUnstakePool balance (ETH): 0
  [1] Attacker balance (shares): 0

  [2] LiquidUnstakePool balance (ETH): 1
  [2] Attacker balance (shares): 1

Test result: ok. 1 passed; 0 failed; finished in 15.68ms
```

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 set the validDeposit modifier in the mint function or include it in the _deposit internal function.

Remediation Plan:

SOLVED: The MetaPool team solved the issue by checking this invariant in the _deposit private function in the following commit ID:

- 79f910ea4f79ba108d21c2c67eb9b59478c2e7c0.

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 6: contracts/LiquidUnstakePool.sol

```
108 function deposit(
109     uint _assets,
110     address _receiver
111 ) public override validDeposit(_assets) returns (uint)
```

Listing 7: contracts/LiquidUnstakePool.sol

```
119 function depositETH(
120     address _receiver
121 ) external payable validDeposit(msg.value) returns (uint)
```

Listing 8: contracts/LiquidUnstakePool.sol

```
168 function redeem(
169     uint _shares,
170     address _receiver,
171     address _owner
172 ) public virtual override nonReentrant returns (uint ETHToSend)
```

Listing 9: contracts/Staking.sol

```
239 function deposit(uint256 _assets, address _receiver)
240         public
241         override
242         validDeposit(_assets)
243         returns (uint256)
```

Listing 10: contracts/Staking.sol

```
252 function depositETH(address _receiver)
253 public
254 payable
255 validDeposit(msg.value)
256 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 IDs:

- 6b4e6770d840a8b90d3bda6ef31fb5de2665d753.

4.3 (HAL-03) 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 (LiquidUnstakePool).
- 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 (LiquidUnstakePool):
  - MUST emit the Deposit event.
- Redeem function (LiquidUnstakePool):
  - MUST emit the Withdraw event.

Code Location:

Listing 11: contracts/LiquidUnstakePool.sol

```
158 function withdraw(
159     uint256,
160     address,
161     address
162 ) public pure override returns (uint) {
163     revert("Use redeem");
164 }
```

Listing 12: contracts/LiquidUnstakePool.sol

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

Listing 13: contracts/LiquidUnstakePool.sol

```
184 emit RemoveLiquidity(msg.sender, _shares, ETHToSend, mpETHToSend);
```

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:

SOLVED: The MetaPool team solved the issue in metapool-ethereum by sticking to EIP-4626 definitions in the following commit ID:

- d6f739a7064ccfe965adb21ea498bcc1d5bb28ef.

4.4 (HAL-04) 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 14

```
error CustomError();

// ...

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

Remediation Plan:

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

4.5 (HAL-05) 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 in the following commit ID:

- c86bac226b5cf581724b368385999cddda4e0bda.

4.6 (HAL-06) 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 15: contracts/Staking.sol

```
121 for (uint i = 0; i < addresses.length; i++)
```

Listing 16: contracts/Staking.sol

```
128 for (uint i = 0; i < addresses.length; i++)
```

Listing 17: contracts/Staking.sol

```
219 for (uint i = 0; i < nodesLength; 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 in the following commit ID:

- 09e5810f590ecb890d914b42bfe6f7d8d085643a.

4.7 (HAL-07) FLOATING PRAGMA - INFORMATIONAL (0.0)

Description:

Smart contracts in metapool-ethereum use the floating pragma ^0.8. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, either an outdated compiler version that might introduce bugs that affect the contract system negatively or a pragma version too new which has not been extensively tested.

Risk Level:

Likelihood - 0
Impact - 0

Recommendation:

Consider locking the pragma version with known bugs for the compiler version by removing the caret (^) symbol. When possible, do not use floating pragma in the final live deployment. Specifying a fixed compiler version ensures that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Remediation Plan:

SOLVED: The MetaPool team solved the issue in the following commit ID:

- f75a74db30d6ad74b7f78af95aabecde315967aa.

4.8 (HAL-08) 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 18: contracts/Staking.sol (Line 160)

```
160 /// @notice Update Withdrawal contract address
161 /// @dev Updater function
162 /// @notice Updates nodes total balance
163 /// @param _newNodesBalance Total current ETH balance from
 ↳ validators
164 function updateNodesBalance(uint _newNodesBalance) external
 ↳ onlyRole(UPDATER_ROLE) {
```

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 removing the Update Withdrawal comment.

Remediation Plan:

SOLVED: The MetaPool team solved the issue in the following commit ID:

- 2150d0bf5d3cd8194bf03802d64b2e7a6cb1526c.

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:

```
ERC4626Upgradeable._deposit(address,address,uint256,uint256) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol#238-255) uses arbitrary from in transferFrom: SafeERC20Upgradeable.safeTransferFrom(_asset,caller,address(this),assets) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol#251)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom

Staking.pushToBeacon(Staking.Node[],uint256,uint256) (contracts/Staking.sol#201-235) sends eth to arbitrary user
        Dangerous calls:
        - depositContract.deposit{value: 32000000000000000000}(_nodes[i].pubkey,_nodes[i].withdrawCredentials,_nodes[i].signature,_nodes[i].depositDataRoot) (contracts/Staking.sol#220-225)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations

Reentrancy in Staking._deposit(address,address,uint256,uint256) (contracts/Staking.sol#265-280):
        External calls:
        - _assets = _getAssetsDeposit(_assets) (contracts/Staking.sol#271)
                - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/utils/SafeERC20Upgradeable.sol#110)
                - IERC20Upgradeable(asset()).safeTransferFrom(msg.sender,address(this),_assets) (contracts/Staking.sol#288-292)
                - IWETH(asset()).withdraw(_assets) (contracts/Staking.sol#293)
                - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts-upgradeable/utils/AddressUpgradeable.sol#135)
        - (sharesFromPool,assetsToPool) = _getmpETHFromPool(_shares,_receiver) (contracts/Staking.sol#272)
                - require(bool,string)(LiquidUnstakePool(liquidUnstakePool).swapETHFormpETH{value: assetsToPool}(_receiver) == sharesFromPool,Pool _shares transfer error) (contracts/Staking.sol#314-319)
        External calls sending eth:
        - _assets = _getAssetsDeposit(_assets) (contracts/Staking.sol#271)
                - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts-upgradeable/utils/AddressUpgradeable.sol#135)
        - (sharesFromPool,assetsToPool) = _getmpETHFromPool(_shares,_receiver) (contracts/Staking.sol#272)
                - require(bool,string)(LiquidUnstakePool(liquidUnstakePool).swapETHFormpETH{value: assetsToPool}(_receiver) == sharesFromPool,Pool _shares transfer error) (contracts/Staking.sol#314-319)
        State variables written after the call(s):
        - _mint(_receiver,_shares) (contracts/Staking.sol#276)
                - _balances[account] += amount (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#272)
        ERC20Upgradeable._balances (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#37) can be used in cross function reentrancies:
        - ERC20Upgradeable._burn(address,uint256) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#290-306)
        - ERC20Upgradeable._mint(address,uint256) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#264-277)
        - ERC20Upgradeable._transfer(address,address,uint256) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#231-253)
        - ERC20Upgradeable.balanceOf(address) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#106-108)
        - _mint(_receiver,_shares) (contracts/Staking.sol#276)
                - _totalSupply += amount (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#269)
        ERC20Upgradeable._totalSupply (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#41) can be used in cross function reentrancies:
        - ERC20Upgradeable._burn(address,uint256) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#290-306)
        - ERC20Upgradeable._mint(address,uint256) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#264-277)
        - ERC20Upgradeable.totalSupply() (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol#99-101)
        - stakingBalance += _assets (contracts/Staking.sol#278)
        Staking.stakingBalance (contracts/Staking.sol#38) can be used in cross function reentrancies:
        - Staking._deposit(address,address,uint256,uint256) (contracts/Staking.sol#265-280)
        - Staking.pushToBeacon(Staking.Node[],uint256,uint256) (contracts/Staking.sol#201-235)
        - Staking.stakingBalance (contracts/Staking.sol#38)
        - Staking.totalAssets() (contracts/Staking.sol#105-112)
Reentrancy in Staking.pushToBeacon(Staking.Node[],uint256,uint256) (contracts/Staking.sol#201-235):
        External calls:
        - LiquidUnstakePool(liquidUnstakePool).getEthForValidator(_requestPoolAmount) (contracts/Staking.sol#213)
        - Withdrawal(withdrawal).getEthForValidator(_requestWithdrawalAmount) (contracts/Staking.sol#215)
        - depositContract.deposit{value: 32000000000000000000}(_nodes[i].pubkey,_nodes[i].withdrawCredentials,_nodes[i].signature,_nodes[i].depositDataRoot) (contracts/Staking.sol#220-225)
        External calls sending eth:
        - depositContract.deposit{value: 32000000000000000000}(_nodes[i].pubkey,_nodes[i].withdrawCredentials,_nodes[i].signature,_nodes[i].depositDataRoot) (contracts/Staking.sol#220-225)
        State variables written after the call(s):
        - stakingBalance -= requiredBalanceFromStaking (contracts/Staking.sol#232)
        Staking.stakingBalance (contracts/Staking.sol#38) can be used in cross function reentrancies:
        - Staking._deposit(address,address,uint256,uint256) (contracts/Staking.sol#265-280)
        - Staking.pushToBeacon(Staking.Node[],uint256,uint256) (contracts/Staking.sol#201-235)
        - Staking.stakingBalance (contracts/Staking.sol#38)
        - Staking.totalAssets() (contracts/Staking.sol#105-112)
        - totalNodesActivated = _totalNodesActivated (contracts/Staking.sol#234)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities

LiquidUnstakePool.redeem(uint256,address,address) (contracts/LiquidUnstakePool.sol#168-185) performs a multiplication on the result of a division:
        - poolPercentage = (_shares * 1000000000000000000) / totalSupply() (contracts/LiquidUnstakePool.sol#176)
        - ETHToSend = (poolPercentage * ethBalance) / 1000000000000000000 (contracts/LiquidUnstakePool.sol#177)
LiquidUnstakePool.redeem(uint256,address,address) (contracts/LiquidUnstakePool.sol#168-185) performs a multiplication on the result of a division:
        - poolPercentage = (_shares * 1000000000000000000) / totalSupply() (contracts/LiquidUnstakePool.sol#176)
        - mpETHToSend = (poolPercentage * Staking(STAKING).balanceOf(address(this))) / 1000000000000000000 (contracts/LiquidUnstakePool.sol#178-179)
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)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply

ERC4626Upgradeable._convertToAssets(uint256,MathUpgradeable.Rounding) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol#217-221) uses a dangerous strict equality:
        - (supply == 0) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol#219-220)
ERC4626Upgradeable._convertToShares(uint256,MathUpgradeable.Rounding) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol#194-200) uses a dangerous strict equality:
        - (assets == 0 || supply == 0) (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol#196-199)
LiquidUnstakePool._getAssetsDeposit(uint256) (contracts/LiquidUnstakePool.sol#143-155) uses a dangerous strict equality:
        - _assets == 0 (contracts/LiquidUnstakePool.sol#144)
ERC4626Upgradeable.isVaultCollateralized() (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol#286-288) uses a dangerous strict equality:
        - totalAssets() > 0 || totalSupply() == 0 (node_modules/@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeable.sol#287)
Staking._getAssetsDeposit(uint256) (contracts/Staking.sol#284-296) uses a dangerous strict equality:
        - _assets == 0 (contracts/Staking.sol#285)
Staking._getmpETHFromPool(uint256,address) (contracts/Staking.sol#302-322) uses a dangerous strict equality:
        - require(bool,string)(LiquidUnstakePool(liquidUnstakePool).swapETHFormpETH{value: assetsToPool}(_receiver) == sharesFromPool,Pool _shares transfer error) (contracts/Staking.sol#314-319)
Staking.initialize(IDeposit,IERC20MetadataUpgradeable,address,address,address) (contracts/Staking.sol#72-97) uses a dangerous strict equality:
        - require(bool,string)(address(this).balance == 0,Error initialize with no zero balance) (contracts/Staking.sol#86-89)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities

Contract locking ether found:
        Contract MockDepositor (test/foundry/utils/mocks/MockDepositor.sol#6-20) has payable functions:
         - IDeposit.deposit(bytes,bytes,bytes,bytes32) (contracts/IDeposit.sol#5-10)
         - MockDepositor.deposit(bytes,bytes,bytes,bytes32) (test/foundry/utils/mocks/MockDepositor.sol#8-15)
         - MockDepositor.receive() (test/foundry/utils/mocks/MockDepositor.sol#17)
        But does not have a function to withdraw the ether
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether
```

- Send ether to an external account issue does not pose any risk since depositContract is supposed to be a contract which stores the ether about to be staked.
- 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 MetaPool ETH testing:

Listing 19: DeploymentHelper.sol

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

import "forge-std/Test.sol";

import { MockDepositor } from "./mocks/MockDepositor.sol";
import { MockWETH } from "./mocks/MockWETH.sol";

import { Staking } from "contracts/Staking.sol";
import { LiquidUnstakePool } from "contracts/LiquidUnstakePool.sol
↳ ";
import { Withdrawal } from "contracts/Withdrawal.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 TREASURY = makeAddr("TREASURY");
    address public UPDATER = makeAddr("UPDATER");
    address public ACTIVATOR = makeAddr("ACTIVATOR");

    MockDepositor public depositor;
    MockWETH public weth;

    Staking public staking;
    LiquidUnstakePool public liquidunstakepool;
    Withdrawal public withdrawal;

    constructor() {
        vm.warp(1683645434);    // Realistic timestamp

        depositor = new MockDepositor();
        weth = new MockWETH("Wrapped ETH", "WETH");

```

```
40          staking = new Staking();
41          liquidunstakepool = new LiquidUnstakePool();
42          withdrawal = new Withdrawal();
43
44          staking.initialize(
45              depositor,
46              IERC20MetadataUpgradeable(address(weth)),
47              TREASURY,
48              UPDATER,
49              ACTIVATOR
50          );
51
52          liquidunstakepool.initialize(
53              payable(staking),
54              IERC20MetadataUpgradeable(address(weth)),
55              TREASURY
56          );
57
58          withdrawal.initialize(
59              payable(staking)
60          );
61
62          staking.updateLiquidPool(payable(liquidunstakepool));
63
64          address[] memory whitelist = new address[](2);
65          whitelist[0] = address(liquidunstakepool);
66          whitelist[1] = address(withdrawal);
67
68          staking.addToWhitelist(whitelist);
69      }
70  }
```

THANK YOU FOR CHOOSING

// HALBORN