Competitive Security Assessment

RewardStation

Jul 16th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.
The comprehensive examination and auditing scope includes:
• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
• Verify the code base is compliant with the most up-to-date industry standards and security best practices.
• Comprehensive line-by-line manual code review of the entire codebase by industry experts.
The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

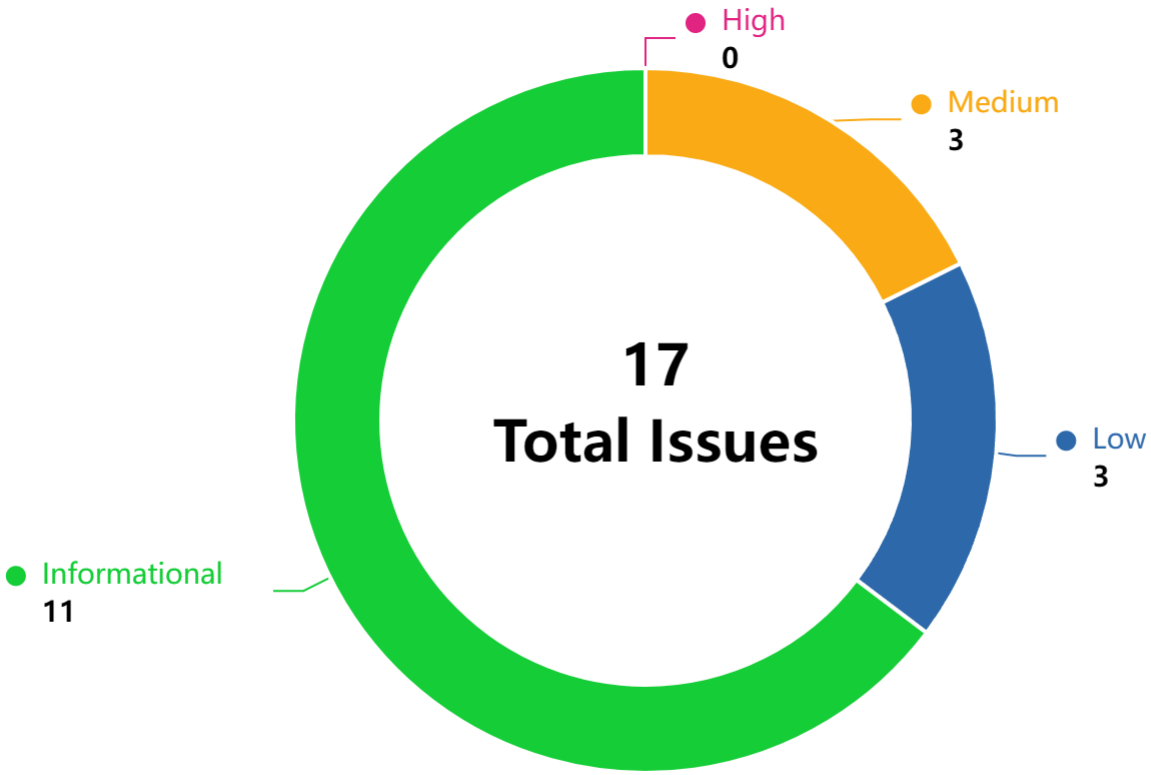
Overview

Project Name	RewardStation
Language	solidity
Codebase	• https://github.com/mantle-xyz/reward-station-contracts • audit version - ae96e12608590fb33238eae2ac58a824d99b6e14 • final version - 11fcffce55ff49f3299b074ca6bc9389b8e90743

Audit Scope

File	SHA256 Hash
src/StakingMNT.sol	f289105ea0a63c3b809c39d81c0f59d1d20cc6cae80fa74a7cc1ea65c745de38
src/AllocateRegister.sol	48982dfcccf5d8bcbacf0c8669c3a6ac9f8770930c835083d8bc8326621414e6
src/StakingMETH.sol	7e67a5c1ab1242fec2206cbeaab2ca9bfe36a54ab78bb02a03c5f514c53d2edf
src/distributes/DistributeYieldERC20.sol	2e1798020d5ef85f2a36ae2e00fbf17abf3246ada6b17bb9446e915e0001ff8d
src/distributes/DistributeMerkleERC20.sol	9e955542ea13c469e8a73f1dc3529b582eba0df2c7ed9d8c475d188a704f8c7d
src/rewards/StandardYieldFarm.sol	6f00532d2b31f7bd42602eeeb5fac70c877542f8134f8278b138d7acd9d2d669
src/LockupUpgradable.sol	9f952d76c4059e8ededb8522dcb26b8ad26e32a229276ba032f7638fb0630894
src/BareVaultUpgradable.sol	e3c62f7fb8d8f9d3065f52e1624778a4ee99df649e78d1438952fff81d4c75fd
src/Pauser.sol	237d77b5858aabc01eddea813275ecd9d7b2fc39e5fce07256f89c8a0775e71f
src/rewards/StandardTranche.sol	a2cfebaa90f7475621e6a93ca8b327da848e1068accae0618e34be3dbff3529c
src/rewards/StandardConversion.sol	f88875e4543e8ac2a93b778478d8205af0ccb6f3f3dbb4ab9b04221a28f45ca6
src/rewards/StandardRewardPoint.sol	39cc54b5bdf482a470526250e697d4f9bd727b1fe5344f7e8d6c2ad9619a4040

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
RST-1	allocate and unallocate will revert	Language Specific	Medium	Fixed	***
RST-2	User stake permanently locked in Contract and amount reduced.	Logical	Medium	Fixed	***
RST-3	Calculation in AllocateRegister.changeAllocate() is wrong due to incorrect casting to uint256	Language Specific	Medium	Fixed	***
RST-4	_updateLockUp logical error	Logical	Low	Fixed	***
RST-5	Missing Check for Claim Pause State in claim Function	Logical	Low	Fixed	***
RST-6	Cei pattern not followed	Reentrancy	Low	Fixed	***
RST-7	unused `errors` and `events` in contracts	Code Style	Informational	Fixed	***
RST-8	uids length not considered	Logical	Informational	Fixed	***

RST-9	`ContextUpgradeable.sol` is not initialized	Logical	Informational	Fixed	***
RST-10	Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom()	Code Style	Informational	Fixed	***
RST-11	Use `disableInitializers` to prevent front-running on the initialize function	Language Specific	Informational	Acknowledged	***
RST-12	Tranche is not validated against during allocation	Logical	Informational	Fixed	***
RST-13	Second Preimage Attack in Merkle Proof Verification using shortned proofs	Privilege Related	Informational	Acknowledged	***
RST-14	Redundant imports in `LockupUpgradable.sol` contract	Gas Optimization	Informational	Fixed	***
RST-15	Native token may be locked in the contract due to functions being payable	Logical	Informational	Fixed	***
RST-16	Missing Pause Check	Logical	Informational	Fixed	***
RST-17	Exchanging exact minimum amount will still revert due to an off by one error	Logical	Informational	Fixed	***

RST-1:allocate and unallocate will revert

Category	Severity	Client Response	Contributor
Language Specific	Medium	Fixed	***

Code Reference

- code/src/AllocateRegister.sol#L199-L202
- code/src/AllocateRegister.sol#L199
- code/src/AllocateRegister.sol#L224-L227
- code/src/AllocateRegister.sol#L224

```
199: bool ok = userAllocated[tranche].set(owner, _amount + amount);
200:         if (!ok) {
201:             revert AllocatedFailed();
202:         }
```

```
199: bool ok = userAllocated[tranche].set(owner, _amount + amount);
```

```
224: bool ok = userAllocated[tranche].set(owner, _amount - amount);
225:             if (!ok) {
226:                 revert UnAllocatedFailed();
227:             }
```

```
224: bool ok = userAllocated[tranche].set(owner, _amount - amount);
```

- code/src/rewards/StandardTranche.sol#L52-L55
- code/src/rewards/StandardTranche.sol#L52
- code/src/rewards/StandardTranche.sol#L69-L72
- code/src/rewards/StandardTranche.sol#L69

```
52: bool ok = userAllocations.set(user, allocated_);
53:         if (!ok) {
54:             revert AllocatedFailed();
55:         }
```

```
52: bool ok = userAllocations.set(user, allocated_);
```

```
69: bool ok = userAllocations.set(user, _amount - amount);
70:             if (!ok) {
71:                 revert UnAllocatedFailed();
72:             }
```

```
69: bool ok = userAllocations.set(user, _amount - amount);
```

Description

***: Currently allocate and unallocate functions in code/src/AllocateRegister.sol and code/src/AllocateRegister.sol will always revert due to incorrect use of EnumerableMap.

Here is the code for `EnumerableMap.set()`:

```
/**
 * @dev Adds a key-value pair to a map, or updates the value for an existing
 * key. O(1).
 *
 * Returns true if the key was added to the map, that is if it was not
 * already present.
 */
function set(Bytes32ToBytes32Map storage map, bytes32 key, bytes32 value) internal returns (bool) {
    map._values[key] = value;
    return map._keys.add(key);
}
```

Here the add() comes from EnumerableSet:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/05f218fb6617932e56bf5388c3b389c3028a7b73/contracts/utils/structs/EnumerableSet.sol#L65-L75

```
function _add(Set storage set, bytes32 value) private returns (bool) {
    if (!_contains(set, value)) {
        set._values.push(value);
        // The value is stored at length-1, but we add 1 to all indexes
        // and use 0 as a sentinel value
        set._positions[value] = set._values.length;
        return true;
    } else {
        return false;
    }
}
```

We can see that if the map already contains the key, `set()` will always return false because of how `EnumerableSet._add()` is programmed.
Back to the allocate and unallocate functions, take `StandardTranche.allocate()` as example:

```
    /// @notice Add a new allocation to owner
    /// @param user The address of the allocation owner
    /// @param amount The amount allocated
    function allocate(address user, uint256 amount) external nonReentrant onlyRegister {
        (,uint256 _amount) = userAllocations.tryGet(user);
        uint256 allocated_ = _amount + amount;
        if (capacity != -1 && int256(allocated_) > capacity ) {
            revert AllocateOverBound();
        }
        // @audit-issue [High] It returns false when key "user" already exists
        bool ok = userAllocations.set(user, allocated_);
        if (!ok) {
            revert AllocatedFailed();
        }
    }
```

The boolean value `ok` here will be false since in this scenario we assume the key `user` exists in EnumerableMap `userAllocations`, and we are trying to update the value corresponding to that key. The consequence is that such txs will always revert, rendering allocate and unallocate functionalities unless.
***: In `AllocateRegister.sol` and `StandardTranche.sol` ,when they allocate and unallocate reward point, they use same code logic like this:

```
bool ok = userAllocations.set(user, _amount - amount);
            if (!ok) {
                revert UnAllocatedFailed();
            }
```

If the original key has a value, this openzeppelin function will return false and trigger a revert.
We can see it in `EnumerableMap.sol` , it will return the result of the element addition of the EnumerableSet:

```
    function set(Bytes32ToBytes32Map storage map, bytes32 key, bytes32 value) internal returns (bool) {
        map._values[key] = value;
        return map._keys.add(key);
    }
```

In EnumerableSet,if there contains this element,it will return false:

```
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }
```

Using `StandardTranche.sol` as an example, `AllocateRegister.sol` also has the same problem.
In `allocate`,if it's the first allocation to an account,it will be successfully executed.But subsequent allocations will revert, because the key already exists and `set` function will return false:

```
    function allocate(address user, uint256 amount) external nonReentrant onlyRegister {
        (,uint256 _amount) = userAllocations.tryGet(user);
        uint256 allocated_ = _amount + amount;
        if (capacity != -1 && int256(allocated_) > capacity ) {
            revert AllocateOverBound();
        }
        bool ok = userAllocations.set(user, allocated_);
        if (!ok) {
            revert AllocatedFailed();
        }
    }
```

and in `unallocate`,if user has zero amount,and register unallocate zero amount,it will be successfully executed.
In other cases it will revert.
It means that the register operator cannot change any of the allocated reward points.
because reward point affect reward amount，so I think it's a high finding.

Recommendation

***: When updating a key-value pair, don't check if `set()` returns true since it will always return false. Take `StandardTranche.allocate()` as example, the logic can be implemented in two branches:

1. If `_amount` is 0, user record is not in the EnumerableMap so the current code is good.
2. If `_amount` is non-zero, just call `set()` without checking its return value:

```
userAllocations.set(user, allocated_);
```

***: remove set return check in function:

```
function allocate(address user, uint256 amount) external nonReentrant onlyRegister {
    (,uint256 _amount) = userAllocations.tryGet(user);
    uint256 allocated_ = _amount + amount;
    if (capacity != -1 && int256(allocated_) > capacity ) {
        revert AllocateOverBound();
    }
    userAllocations.set(user, allocated_);
}
```

Client Response

client response for ret2basic: Fixed. Fixed
under 02b2205
client response for linmiaomiao: Fixed. Fixed
under 02b2205

RST-2:User stake permanently locked in Contract and amount reduced.

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/src/LockupUpgradable.sol#L80

```
80: $._lockUps[owner].amounts[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
```

Description

***: When updatings to release expired locks, the old lock is discarded by overriding it with the last element in the lock array, then popping the last element. But when the old lock is being overriden, the amount and duration which is meant to be that of the last is assigned the lockStart time, same as the duration.

```
function _updateLockUp(address owner) internal {
        UserLockStorage storage $ = _getUserLockStorage();
        (,uint256 amount) = $._userLocked.tryGet(owner);
        if ($._lockUps[owner].amounts.length == 0) {
            return;
        }
        // [1-,2,3,4-] => [1-,2,3] => [3,2];
        uint256 currentLen;
        uint256 currentAmount;
        for (uint256 i = $._lockUps[owner].amounts.length; i > 0; i--) {
            // @note start + duration = end
            // less than timestamp means it has expired
            // @note - 1 cause it is starting from the end so len.max is not a valid index
            if ($._lockUps[owner].lockStarts[i-1] + $._lockUps[owner].durations[i-1] < block.timestamp) {

                currentLen = $._lockUps[owner].lockStarts.length;
                currentAmount = $._lockUps[owner].amounts[i-1];

                if (i == currentLen) {
                    // if element is the last on; skip swap
                } else {
                >>>>>>>>>>>>>>>>>>here>>>>>>>>>>>>>>>>
                    $._lockUps[owner].lockStarts[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
                    $._lockUps[owner].amounts[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
                    $._lockUps[owner].durations[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
                >>>>>>>>>>>>>>>>>>here>>>>>>>>>>>>>>>>
                }
                amount -= currentAmount;
                $._lockUps[owner].lockStarts.pop();
                $._lockUps[owner].amounts.pop();
                $._lockUps[owner].durations.pop();
            }
        }
        $._userLocked.set(owner, amount);
        emit UserLockUpdated(owner, amount);
    }
```

It means now that the new duration of the lock is basically LockStart + LockStart, keep in mind that lock start is a time stamp, counting from 1 jan 1970, if the start time is **1720009840, it means that the stake will be locked for about 54 years. Also the amount is ovveriden, which means the new amount would be 1720009840, this is regardless of if the user had a lesser or higher amount, with this either the user losses funds or the protocol does.**

Recommendation

***: Fill the right values into the mapping.

Client Response

client response for 0xffchain: Fixed. Fixed under 9c25f75

RST-3:Calculation in AllocateRegister.changeAllocate() is wrong due to incorrect casting to uint256

Category	Severity	Client Response	Contributor
Language Specific	Medium	Fixed	***

Code Reference

- code/src/AllocateRegister.sol#L157-L168
- code/src/AllocateRegister.sol#L165
- code/src/AllocateRegister.sol#L185-L194

```
157: function changeAllocate(AllocateMsg[] memory msgs) external onlyRole(REGISTER_OPERATOR_ROLE) {
158:         if (pauser.isAllocationPaused()) {
159:             revert Paused();
160:         }
161:         for (uint256 i; i < msgs.length; i++) {
162:             if (msgs[i].amount >= 0) {
163:                 _allocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
164:             } else {
165:                 _unallocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
166:             }
167:         }
168:     }
```

```
165: _unallocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
```

```
185: function unallocate(
186:         address tranche,
187:         address owner,
188:         uint256 amount
189:     ) public onlyRole(REGISTER_OPERATOR_ROLE) returns (address, uint256, uint256, uint256) {
190:         if (pauser.isAllocationPaused()) {
191:             revert Paused();
192:         }
193:         return _unallocate(tranche, owner, amount);
194:     }
```

Description

***: The following line of code attempts to convert a negative number `msgs[i].amount` to its absolute value by casting it to uint256 directly. This will give unexpected result since casting to uint256 is not equivalent to computing absolute value. Instead, it attempts to interpret 2's complement representation of a negative number as positive number, therefore it will return a gibberish large number.
For a toy PoC, copy the following contract into Remix IDE:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.8.20;

contract int256ToUint256 {

    function convert() public pure returns(uint256){
        int256 x = -3;
        return uint256(x);
    }
}
```

Developer might expect it to return 3, but it returns a huge number instead:

```
115792089237316195423570985008687907853269984665640564039457584007913129639933
```

***: The `changeAllocate` function attempts to cast a negative int256 amount to uint256 when calling `_unallocate`. This results in a large positive value due to underflow, causing unexpected behavior.

```
    function changeAllocate(AllocateMsg[] memory msgs) external onlyRole(REGISTER_OPERATOR_ROLE) {
        if (pauser.isAllocationPaused()) {
            revert Paused();
        }
        for (uint256 i; i < msgs.length; i++) {
            if (msgs[i].amount >= 0) {
                _allocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
            } else {
                _unallocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
            }
        }
    }


function unallocate(
        address tranche,
        address owner,
        uint256 amount
    ) public onlyRole(REGISTER_OPERATOR_ROLE) returns (address, uint256, uint256, uint256) {
        if (pauser.isAllocationPaused()) {
            revert Paused();
        }
        return _unallocate(tranche, owner, amount);
    }
```

poc

```
function testChangeAllocatenegative() public {
        vm.prank(registerOperator);
        allocateRegisterMNT.allocate(address(tranche), owner, amount);
        require(allocateRegisterMNT.totalAllocated(address(tranche)) == amount, "totalAllocated not correct
after allocate");
        require(allocateRegisterMNT.userTotalAllocated(owner) == amount, "userTotalAllocated not correct aft
er allocate");


        IAllocateRegister.AllocateMsg[] memory msgs = new IAllocateRegister.AllocateMsg[](1);
        msgs[0] = IAllocateRegister.AllocateMsg({tranche: address(tranche), owner: owner, amount: -int256(am
ount)});
        vm.prank(registerOperator);
        allocateRegisterMNT.changeAllocate(msgs);
    }
```

***: changeAllocate function in AllocateRegister.sol have this code:

```
    function changeAllocate(AllocateMsg[] memory msgs) external onlyRole(REGISTER_OPERATOR_ROLE) {
        if (pauser.isAllocationPaused()) {
            revert Paused();
        }
        for (uint256 i; i < msgs.length; i++) {
            if (msgs[i].amount >= 0) {
                _allocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
            } else {
                _unallocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
            }
        }
    }
```

when uncllocating reward point,it need `msgs[i].amount < 0`,and will change an negative int256 amount to uint256:

```
_unallocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
```

This number conversion is wrong， It will convert a negative number to the complement, not the positive number with the negative sign removed.For example,-1 will convert to type(uint.256).max - 1,not 1.
If the amount allocated is not large enough, then unallocate will fail with insufficient amount.And when the allocated amount is large enough, then the register operator will unallocate an incorrect amount.
Reward point will affect users reward,so I think it's a high finding.

Recommendation

***: An elegant way to handle such case is implementing a function to compute absolute value:

```
function abs(int256 x) private pure returns (uint256) {
    return x >= 0 ? uint256(x) : uint256(-x);
}
```

And change the code to:

```
_unallocate(msgs[i].tranche, msgs[i].owner, abs(msgs[i].amount));
```

***:

```
function changeAllocate(AllocateMsg[] memory msgs) external onlyRole(REGISTER_OPERATOR_ROLE) {
    if (pauser.isAllocationPaused()) {
        revert Paused();
    }
    for (uint256 i; i < msgs.length; i++) {
        if (msgs[i].amount >= 0) {
            _allocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
        } else {
-            _unallocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
+            _unallocate(msgs[i].tranche, msgs[i].owner, uint256(-msgs[i].amount));
        }
    }
}
```

***: fix code like this:

```
    function changeAllocate(AllocateMsg[] memory msgs) external onlyRole(REGISTER_OPERATOR_ROLE) {
        if (pauser.isAllocationPaused()) {
            revert Paused();
        }
        for (uint256 i; i < msgs.length; i++) {
            if (msgs[i].amount >= 0) {
                _allocate(msgs[i].tranche, msgs[i].owner, uint256(msgs[i].amount));
            } else {
                _unallocate(msgs[i].tranche, msgs[i].owner, uint256(-msgs[i].amount));
            }
        }
    }
```

Client Response

client response for ret2basic: Fixed. fixed under commit 4ddc56c
client response for 8olidity: Fixed. fixed under commit 4ddc56c
client response for linmiaomiao: Fixed. fixed under commit 4ddc56c

RST-4:_updateLockUp logical error

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/src/LockupUpgradable.sol#L79-L81
- code/src/LockupUpgradable.sol#L80

```
79: $._lockUps[owner].lockStarts[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
80:                     $._lockUps[owner].amounts[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
81:                     $._lockUps[owner].durations[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
```

```
80: $._lockUps[owner].amounts[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
```

Description

***: **Description**:
The `_updateLockUp` function in the LockupUpgradable contract contains multiple incorrect assignment statements. Specifically, the lines:

```
$._lockUps[owner].amounts[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
$._lockUps[owner].durations[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
```

are incorrect because they assign the lockStarts value to the amounts and durations arrays, respectively. This can lead to data corruption. The correct assignments should be:

```
$._lockUps[owner].amounts[i-1] = $._lockUps[owner].amounts[currentLen-1];
$._lockUps[owner].durations[i-1] = $._lockUps[owner].durations[currentLen-1];
```

These corrections ensure that the amounts and durations arrays are correctly updated with the appropriate values.
Impact:
The incorrect assignment can have severe consequences, especially in the context of a staking mechanism where accurate accounting of user lockups is crucial. The potential impacts include:

- Incorrect values in the amounts array can lead to corrupted data, making it impossible to accurately track user lockups.
- Users may see incorrect lockup amounts, leading to potential financial losses or gains that are not intended by the protocol.
- Malicious actors could exploit the corrupted data to unlock more assets than they should be able to, leading to potential financial exploits.

***: The `lockStarts` value cannot be directly assigned to the `amounts` and `durations` arrays.

```
} else {
                    $._lockUps[owner].lockStarts[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
                    $._lockUps[owner].amounts[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
                    $._lockUps[owner].durations[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
                }
```

Recommendation

***: To mitigate this issue, Update the _updateLockUp function to ensure correct assignments:

```
    function _updateLockUp(address owner) internal {
        UserLockStorage storage $ = _getUserLockStorage();
        (,uint256 amount) = $._userLocked.tryGet(owner);
        if ($._lockUps[owner].amounts.length == 0) {
            return;
        }
        // [1-,2,3,4-] => [1-,2,3] => [3,2];
        uint256 currentLen;
        uint256 currentAmount;
        for (uint256 i = $._lockUps[owner].amounts.length; i > 0; i--) {
            if ($._lockUps[owner].lockStarts[i-1] + $._lockUps[owner].durations[i-1] < block.timestamp) {
                currentLen = $._lockUps[owner].lockStarts.length;
                currentAmount = $._lockUps[owner].amounts[i-1];
                if (i == currentLen) {
                    // if element is the last on; skip swap
                } else {
                    $._lockUps[owner].lockStarts[i-1] = $._lockUps[owner].lockStarts[currentLen-1];
                    $._lockUps[owner].amounts[i-1] = $._lockUps[owner].amounts[currentLen-1];
                    $._lockUps[owner].durations[i-1] = $._lockUps[owner].durations[currentLen-1];
                }
                amount -= currentAmount;
                $._lockUps[owner].lockStarts.pop();
                $._lockUps[owner].amounts.pop();
                $._lockUps[owner].durations.pop();
            }
        }
        $._userLocked.set(owner, amount);
        emit UserLockUpdated(owner, amount);
    }
```

***: To correct this issue, ensure that the correct values are being reassigned during the update process. The corrected line should be:

```
$._lockUps[owner].amounts[i-1] = $._lockUps[owner].amounts[currentLen-1];
$._lockUps[owner].durations[i-1] = $._lockUps[owner].durations[currentLen-1];
```

Client Response

client response for 0xShax2nk: Fixed. under 9c25f75
client response for 8olidity: Fixed. under 9c25f75

RST-5:Missing Check for Claim Pause State in claim Function

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/src/distributes/DistributeMerkleERC20.sol#L142

```
142: function claim(uint256 id, uint256 index, address account, uint256 amount, bytes32[] calldata merkleProo
f)
```

Description

***: Description**:
The `DistributeMerkleERC20Upgradeable` contract is designed to distribute ERC20 tokens using Merkle proofs. It inherits from a pauser contract, which includes an `isClaimPaused` state variable to control whether claims can be processed. However, the `claim` function does not check this variable before proceeding with the claim logic. The absence of this check means that claims can be processed even when the system is paused, which could lead to unauthorized or unintended token distributions.
Impact:
The missing check for the isClaimPaused state variable can have several negative impacts:

- **Unauthorized Claims**: Claims can be processed even when the system is paused, leading to unauthorized or unintended token distributions.
- **Security Risks**: Pausing the system is often a security measure to prevent further actions during an investigation or fix. Allowing claims during this period undermines this security measure.
- **Operational Issues**: The ability to pause claims is crucial for maintenance and updates. Without this check, the system cannot be reliably paused, complicating operational management.

Recommendation

***: To mitigate the risks associated with this issue, it is recommended to add a check for the isClaimPaused state variable in the claim function. If the claim is paused, the function should revert with a ClaimPaused error.

```
 function claim(uint256 id, uint256 index, address account, uint256 amount, bytes32[] calldata merkleProof)
    public
    virtual
    override
    nonReentrant
    onlyAllowedToken(id)
    {

+        // Check if claiming is paused
+        if ($.pauser.isClaimPaused()) revert ClaimPaused();

        if (isClaimed(id, index)) revert AlreadyClaimed();

        // Verify the merkle proof.
        DistributeMerkleERC20Storage storage $ = _getDistributeMerkleERC20Storage();
        if ($.endTimes[id] != 0 && block.timestamp > $.endTimes[id]) revert ClaimWindowFinished();
        bytes32 node = keccak256(abi.encodePacked(id, index, account, amount));
        if (!MerkleProof.verify(merkleProof, $.merkleRoots[id], node)) revert InvalidProof();

        // Mark it claimed and send the token.
        _setClaimed(id, index);
        IERC20($.allowedTokens[id]).safeTransferFrom($.sponsors[id], account, amount);

        emit Claimed($.allowedTokens[id], index, account, amount);
    }
```

Steps to Implement the Recommendation:

1. Add the ClaimPaused Error:

- error ClaimPaused();

2. Update the claim Function:

- Add the check for isClaimPaused before any other logic in the claim function.
- Revert with the ClaimPaused error if claiming is paused.

Client Response

client response for 0xShax2nk: Fixed. under commit 6484532

RST-6:Cei pattern not followed

Category	Severity	Client Response	Contributor
Reentrancy	Low	Fixed	***

Code Reference

- code/src/distributes/DistributeYieldERC20.sol#L140

```
140: IERC20($.allowedTokens[id]).safeTransferFrom($.sponsors[id], account, amount);
```

- code/src/StakingMNT.sol#L247-L260

```
247: function _withdraw(
248:         address caller,
249:         address receiver,
250:         uint256 assets
251:     ) internal override {
252:         BareVaultStorage storage $ = super._getBareVaultStorage();
253:
254:         Address.sendValue(payable(receiver), assets);
255:
256:         $._deposit.set(caller, $._deposit.get(caller) - assets);
257:         $._totalDeposit -= assets;
258:
259:         emit Withdraw(caller, receiver, assets);
260:     }
```

Description

***: The `_withdraw` function demonstrates a potential vulnerability to reentrancy attacks due to non-adherence to the Checks-Effects-Interactions (CEI) pattern. The CEI pattern is a fundamental principle in Ethereum smart contract development, which mandates that state changes (effects) should occur before making external calls (interactions). This pattern mitigates the risk of reentrancy attacks, where an external contract can repeatedly call back into the vulnerable contract before the initial function call is completed.
In the given function, the external call to transfer assets (`Address.sendValue(payable(receiver), assets)`) occurs before updating the internal state variables (`$._deposit.set(caller, $._deposit.get(caller) - assets);` and `$._totalDeposit -= assets;`). This sequence of operations can lead to a scenario where an attacker exploits the read only reentrancy vulnerability by reading the state variables before update.
Note: The Read-Only Renentrancy can occur even when the `nonReentrant` modifier is used in the contract.
***: ## Vulnerability Detail
`getReward` function in `DistributeYieldERC20` contract even have `nonReentrant` modifier it does not follow the best security practice related to Check-Effects-Interaction(CEI) pattern where all check should be at top and effects carried out before and interaction to avoid risk of any type of potential attack, refer to this article.

```
    function getReward(uint256 id, address account) public virtual nonReentrant returns (uint256) {
        DistributeYieldERC20Storage storage $ = _getDistributeYieldERC20Storage();
        if ($.endTimes[id] != 0 && block.timestamp > $.endTimes[id]) revert ClaimWindowFinished();

        uint256 amount = IAllocateYield($.yields[id]).getReward(account);
        if (amount > 0) {
            IERC20($.allowedTokens[id]).safeTransferFrom($.sponsors[id], account, amount);
            $.totalClaimed[id] += amount;
            (, uint256 _userClaimed) = $.userClaimed[id].tryGet(account);
            $.userClaimed[id].set(account, _userClaimed + amount);
            emit Claimed($.yields[id], account, amount);
        }
        return amount;
    }
```

Recommendation

***: To prevent read only reentrancy attacks and ensure the security of the contract, it is crucial to adhere to the CEI pattern by updating the internal state variables before making any external calls. The revised `_withdraw` function should update the `_deposit` and `_totalDeposit` variables prior to transferring the assets. Here is the recommended modification:

```
function _withdraw(
    address caller,
    address receiver,
    uint256 assets
) internal override {
    BareVaultStorage storage $ = super._getBareVaultStorage();

    // Update state variables before making the external call
    $._deposit.set(caller, $._deposit.get(caller) - assets);
    $._totalDeposit -= assets;

    // Transfer the assets to the receiver
    Address.sendValue(payable(receiver), assets);

    // Emit the Withdraw event
    emit Withdraw(caller, receiver, assets);
}
```

***: The recommendation is made to implement code by follow the CEI pattern for best security practice.

```
    function getReward(uint256 id, address account) public virtual nonReentrant returns (uint256) {
        DistributeYieldERC20Storage storage $ = _getDistributeYieldERC20Storage();
        if ($.endTimes[id] != 0 && block.timestamp > $.endTimes[id]) revert ClaimWindowFinished();

        uint256 amount = IAllocateYield($.yields[id]).getReward(account);
        if (amount > 0) {
-           IERC20($.allowedTokens[id]).safeTransferFrom($.sponsors[id], account, amount);
            $.totalClaimed[id] += amount;
            (, uint256 _userClaimed) = $.userClaimed[id].tryGet(account);
            $.userClaimed[id].set(account, _userClaimed + amount);
            emit Claimed($.yields[id], account, amount);
+           IERC20($.allowedTokens[id]).safeTransferFrom($.sponsors[id], account, amount);

        }
        return amount;
    }
```

Client Response

client response for 0xzoobi: Fixed. under c81dc53c
client response for Saaj: Fixed. under c81dc53c

RST-7:unused `errors` and `events` in contracts

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	***

Code Reference

- code/src/StakingMETH.sol#L22

```
22: error UnexpectedAmount();
```

Description

***: Following errors and events in contracts have not been used in contract functionalities so it should be removed to make the code more readeable. Its recommended to not keep unused errors/functions/variables/events in contract which unnecessary increases byte size.
In `StakingMETH.sol`:

```
    error UnexpectedAmount();
    error AutoUnAllocateFailed();

    event Allocated(address indexed tranche, address indexed owner, uint256 amount);
    event UnAllocated(address indexed owner, uint256 amount);
```

In `StakingMNT.sol`:

```
    error AutoUnAllocateFailed();

    event Allocated(address indexed tranche, address indexed owner, uint256 amount);
    event UnAllocated(address indexed owner, uint256 amount);
```

Recommendation

***: Remove unused errors and events as listed above in relevant contracts.

Client Response

client response for 0xRizwan: Fixed. fixed under 8d064ec

RST-8:uids length not considered

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/src/distributes/DistributeYieldERC20.sol#L177

```
177: function _setElements(uint256[] memory uids, address[] memory yields, address[] memory tokens, address[] memory sponsors, uint256[] memory endTimes) internal {
```

Description

***:

`uids` length is not checked in the `_setElements` function while all other inputs arguments are checked.
It is better to check length of `uids` to avoid bypassing or overriding of inputs if length differe from rest of the params.

```
    function _setElements(
        uint256[] memory uids,
        address[] memory yields,
        address[] memory tokens,
        address[] memory sponsors,
        uint256[] memory endTimes
    ) internal {
        DistributeYieldERC20Storage storage $ = _getDistributeYieldERC20Storage();

        if (yields.length != tokens.length || tokens.length != sponsors.length || sponsors.length != endTimes.length) {
            revert LengthDiffer();
        }
        for (uint256 i; i < tokens.length; i++) {
            if ($.allowedTokens[uids[i]] != address(0)) {
                revert RepeatSetElement();
            }
            $.yields[uids[i]] = yields[i];
            $.sponsors[uids[i]] = sponsors[i];
            $.endTimes[uids[i]] = endTimes[i];
            $.allowedTokens[uids[i]] = tokens[i];
        }
        emit ProtocolConfigChanged(
            bytes4(keccak256("_setElements(uint256[],address[],address[],address[],uint256[])")),
            "_setElements(uint256[],address[],address[],address[],uint256[])",
            abi.encode(uids, yields, tokens, sponsors, endTimes)
        );
    }
```

Recommendation

***: The recommendation is made to also add check for length of `uids`.

```
    function _setElements(
        uint256[] memory uids,
        address[] memory yields,
        address[] memory tokens,
        address[] memory sponsors,
        uint256[] memory endTimes
    ) internal {
        DistributeYieldERC20Storage storage $ = _getDistributeYieldERC20Storage();

-         if (yields.length != tokens.length || tokens.length != sponsors.length || sponsors.length != endTim
es.length) {

+         if (uids.length != yields.length || yields.length != tokens.length || tokens.length != sponsors.len
gth || sponsors.length != endTimes.length) {

            revert LengthDiffer();
        }
        for (uint256 i; i < tokens.length; i++) {
            if ($.allowedTokens[uids[i]] != address(0)) {
                revert RepeatSetElement();
            }
            $.yields[uids[i]] = yields[i];
            $.sponsors[uids[i]] = sponsors[i];
            $.endTimes[uids[i]] = endTimes[i];
            $.allowedTokens[uids[i]] = tokens[i];
        }
        emit ProtocolConfigChanged(
            bytes4(keccak256("_setElements(uint256[],address[],address[],address[],uint256[])")),
            "_setElements(uint256[],address[],address[],address[],uint256[])",
            abi.encode(uids, yields, tokens, sponsors, endTimes)
        );
    }
```

Client Response

client response for Saaj: Fixed. Fixed under b6b2557

RST-9: `ContextUpgradeable.sol` is not initialized

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/src/BareVaultUpgradable.sol#L14

```
14: abstract contract BareVaultUpgradable is ContextUpgradeable, AccessControlEnumerableUpgradeable {
```

Description

***: `BareVaultUpgradable.sol` is the base contract which is inherited by both `StakingMNT.sol` and `StakingMETH.sol` contracts. `BareVaultUpgradable.sol` also inherits the openzeppelin's `ContextUpgradeable.sol`

```
abstract contract BareVaultUpgradable is ContextUpgradeable, AccessControlEnumerableUpgradeable {
```

The issue is that `ContextUpgradeable.sol` is not initialized in `initialize()` function. As per openzeppelin, for upgradeable contracts `__Context_init()` should be initialized in inheriting contract.

Recommendation

***: Consider below changes in `BareVaultUpgradable.sol`:

```
    function __BareVault_init(address _asset) internal onlyInitializing {
        __AccessControlEnumerable_init();
+           __Context_init();
        __BareVault_init_unchained(_asset);
    }
```

Client Response

client response for 0xRizwan: Fixed.
fixed under 8b271ba, actually no extra logic under __Context_init();
Info / Recommandation

RST-10:Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom()

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	***

Code Reference

- code/src/rewards/StandardConversion.sol#L23
- code/src/rewards/StandardConversion.sol#L54
- code/src/rewards/StandardConversion.sol#L54-L63
- code/src/rewards/StandardConversion.sol#L60
- code/src/rewards/StandardConversion.sol#L68

```
23: constructor(IERC20 _sourceToken, IERC20 _targetToken, uint256 _exchangeRate) Ownable() {
```

```
54: bool sourceTokenTransferSuccess = sourceToken.transferFrom(msg.sender, address(this), _sourceTokenAmount);
```

```
54: bool sourceTokenTransferSuccess = sourceToken.transferFrom(msg.sender, address(this), _sourceTokenAmount);
55:         if (!sourceTokenTransferSuccess) {
56:             revert TransferFailed();
57:         }
58:
59:         // Transfer equivalent ENA to the user
60:         bool targetTokenTransferSuccess = targetToken.transfer(msg.sender, targetTokenAmount);
61:         if (!targetTokenTransferSuccess) {
62:             revert TransferFailed();
63:         }
```

```
60: bool targetTokenTransferSuccess = targetToken.transfer(msg.sender, targetTokenAmount);
```

```
68: bool withdrawSuccess = targetToken.transfer(owner(), targetToken.balanceOf(address(this)));
```

Description

***: In the `Conversion` contract, token transfers are performed using `transfer` and `transferFrom` methods. These methods do not handle all potential return values correctly, which can lead to security issues. Specifically, if a token contract does not return a boolean value as expected, the transfer may fail silently, leading to potential inconsistencies in the contract's state and user balances.
The following code snippet shows the problematic sections in the `exchangeTargetToken` and `withdraw` functions:

```
bool sourceTokenTransferSuccess = sourceToken.transferFrom(msg.sender, address(this), _sourceTokenAmount);
if (!sourceTokenTransferSuccess) {
    revert TransferFailed();
}

// Transfer equivalent ENA to the user
bool targetTokenTransferSuccess = targetToken.transfer(msg.sender, targetTokenAmount);
if (!targetTokenTransferSuccess) {
    revert TransferFailed();
}

function withdraw() external onlyOwner {
    bool withdrawSuccess = targetToken.transfer(owner(), targetToken.balanceOf(address(this)));
    if (!withdrawSuccess) {
        revert TransferFailed();
    }
}
```

The use of `bool` alone does not fully mitigate the issue because the ERC20 standard does not require token contracts to return a boolean value for `transfer` and `transferFrom` functions. Some token contracts might not return a value at all or may revert on failure without returning false. Therefore, the contract might still incorrectly assume a transfer was successful when it wasn't.

Impact

If the `transfer` or `transferFrom` method fails without returning the expected boolean value, the contract may not correctly handle the failure, potentially leading to loss of funds for users or the contract itself. This can cause users to lose their tokens or receive incorrect amounts, damaging the integrity and reliability of the contract.
***:** In the `Conversion` contract, token transfers are performed using `transfer` and `transferFrom` methods. These methods do not handle all potential return values correctly, which can lead to security issues. Specifically, if a token contract does not return a boolean value as expected, the transfer may fail silently, leading to potential inconsistencies in the contract's state and user balances.
The following code snippet shows the problematic sections in the `exchangeTargetToken` and `withdraw` functions:

```
bool sourceTokenTransferSuccess = sourceToken.transferFrom(msg.sender, address(this), _sourceTokenAmount);
if (!sourceTokenTransferSuccess) {
    revert TransferFailed();
}

// Transfer equivalent ENA to the user
bool targetTokenTransferSuccess = targetToken.transfer(msg.sender, targetTokenAmount);
if (!targetTokenTransferSuccess) {
    revert TransferFailed();
}

function withdraw() external onlyOwner {
    bool withdrawSuccess = targetToken.transfer(owner(), targetToken.balanceOf(address(this)));
    if (!withdrawSuccess) {
        revert TransferFailed();
    }
}
```

The use of `bool` alone does not fully mitigate the issue because the ERC20 standard does not require token contracts to return a boolean value for `transfer` and `transferFrom` functions. Some token contracts might not return a value at all or may revert on failure without returning false. Therefore, the contract might still incorrectly assume a transfer was successful when it wasn't.

Impact

If the `transfer` or `transferFrom` method fails without returning the expected boolean value, the contract may not correctly handle the failure, potentially leading to loss of funds for users or the contract itself. This can cause users to lose their tokens or receive incorrect amounts, damaging the integrity and reliability of the contract.
***:

Code Snippet

Vulnerability Detail

The ERC20.transfer() and ERC20.transferFrom() functions return a boolean value indicating success. This parameter needs to be checked for success.
Some tokens do not revert if the transfer failed but return false instead.

```
        // Transfer source token to the contract
        bool sourceTokenTransferSuccess = sourceToken.transferFrom(msg.sender, address(this), _sourceTokenAm
ount);
        if (!sourceTokenTransferSuccess) {
            revert TransferFailed();
        }

        // Transfer equivalent ENA to the user
        bool targetTokenTransferSuccess = targetToken.transfer(msg.sender, targetTokenAmount);
        if (!targetTokenTransferSuccess) {
            revert TransferFailed();
        }
    }

    // Allows the owner of the contract to withdraw a certain amount of ENA tokens from the contract.
    function withdraw() external onlyOwner {
        bool withdrawSuccess = targetToken.transfer(owner(), targetToken.balanceOf(address(this)));
        if (!withdrawSuccess) {
            revert TransferFailed();
        }
    }
```

Some tokens (like USDT) don't correctly implement the EIP20 standard and their `transfer/ transferFrom` function return void instead of a success boolean. Calling these functions with the correct EIP20 function signatures will always revert.

Impact

Tokens that don't actually perform the transfer and return false are still counted as a correct transfer and tokens that don't correctly implement the latest EIP20 spec, like USDT, will be unusable in the protocol as they revert the transaction because of the missing return value.
***:** The issue is related to usage of `transfer` and `transferFrom`, and how those functions are implemented by IERC20 interface. The `targetToken` as well as `sourceToken` are of data type `IERC20`. Please see a snippet from OZ's `IERC20.sol` below:

```
function transferFrom(address from, address to, uint256 value) external returns (bool);
...
function transfer(address to, uint256 value) external returns (bool);
```

Those functions are required by the interface to return a bool value. On the other hand, some tokens, such as USDT, do not return anything on `transfer`. Since those tokens are assigned `IERC20` type, the function calls such as

```
bool withdrawSuccess = targetToken.transfer(owner(), targetToken.balanceOf(address(this)));
```

will revert due to not returning expected value.

As an additional, less critical impact - if `sourceToken` is non compliant with `IERC20`, then the users will not be able to convert it.
Consider standalone unit test as a PoC:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import "../src/IERC20.sol";

contract USDTTest is Test {

IERC20 usdt = IERC20(0xdAC17F958D2ee523a2206206994597C13D831ec7);
IERC20 public sourceToken;
address public bob = makeAddr("bob");
address public alice = makeAddr("alice");
IERC20 public mntToken;
uint256 mainnetFork; // forge test --fork-url $INFURA_MAINNET --fork-block-number 20189632

    function setUp() public {
        sourceToken = IERC20(0xdAC17F958D2ee523a2206206994597C13D831ec7); //usdt
        mntToken = IERC20(0x3c3a81e81dc49A522A592e7622A7E711c06bf354);
        mainnetFork = vm.createFork("https://mainnet.infura.io/v3/[YOUR INFURA KEY]");
        //deal(address(usdt), alice, 1_000_000_000_000);

    }

    function test_transfer() public {
        console.log(sourceToken.balanceOf(0x23878914EFE38d27C4D67Ab83ed1b93A74D4086a)); //random whale
        console.log(mntToken.balanceOf(address(0x78605Df79524164911C144801f41e9811B7DB73D))); //random whale
        vm.prank(0x78605Df79524164911C144801f41e9811B7DB73D);
        bool success = mntToken.transfer(alice, 10_000);
        console.log("First transfer: ", success);
        vm.prank(0x23878914EFE38d27C4D67Ab83ed1b93A74D4086a);
        success = sourceToken.transfer(alice, 10_000);//revert
    }

}
```

***: The `exchangeTargetToken` and `withdraw` functions of **Conversion** contract, performs an ERC20 `transfer` / `transferFrom` but does not check the return value, nor does it work with all legacy tokens

Bug Description

Some tokens (like **USDT**) don't correctly implement the **EIP20** standard and their `transfer` / `transferFrom` function return void instead of a success boolean. Calling these functions with the correct **EIP20** function signatures will always revert.
The `ERC20.transfer()` and `ERC20.transferFrom()` functions return a boolean value indicating success. This parameter needs to be checked for success.

Some tokens do not revert if the transfer failed but return false instead.
Tokens that don't actually perform the `transfer` and return false are still counted as a correct transfer and tokens that don't correctly implement the latest **EIP20** spec, like **USDT**, will be unusable in the protocol as they revert the transaction because of the missing return value.

Impact

The `exchangeTargetToken` and `withdraw` functions do not work with all ERC20 Tokens, even if the `sourceToken` / `targetToken` would return `false` indicating that the `transfer` was not performed it would be given as successful giving the possibility of various security problems such as token theft.

References

- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.0/contracts/token/ERC20/utils/SafeERC20.sol

***:** The transfer and transferFrom functions from the IERC20 interface are used to transfer tokens within the Conversion contract. These functions return a boolean value indicating success or failure. However, some ERC20 token implementations might revert on failure instead of returning false. The current implementation does not handle this possibility, which could lead to unhandled exceptions and failure of token transfers.

Recommendation

***:** Replace the `transfer` and `transferFrom` methods with `safeTransfer` and `safeTransferFrom` from OpenZeppelin's `SafeERC20` library. These methods handle potential issues with token contracts that do not return a boolean value and ensure that the transfer is successful.
***:** Replace the `transfer` and `transferFrom` methods with `safeTransfer` and `safeTransferFrom` from OpenZeppelin's `SafeERC20` library. These methods handle potential issues with token contracts that do not return a boolean value and ensure that the transfer is successful.
***:** ## Recommendation
Recommend using OpenZeppelin's SafeERC20 with the safeTransfer and safeTransferFrom functions that handle the return value check as well as non-standard-compliant tokens.

```
        // Transfer source token to the contract
-       bool sourceTokenTransferSuccess = sourceToken.transferFrom(msg.sender, address(this), _sourceTokenAm
ount);
+       bool sourceTokenTransferSuccess = sourceToken.safeTransferFrom(msg.sender, address(this), _sourceTok
enAmount);
        if (!sourceTokenTransferSuccess) {
            revert TransferFailed();
        }

        // Transfer equivalent ENA to the user
-       bool targetTokenTransferSuccess = targetToken.transfer(msg.sender, targetTokenAmount);
+       bool targetTokenTransferSuccess = targetToken.safeTransfer(msg.sender, targetTokenAmount);
        if (!targetTokenTransferSuccess) {
            revert TransferFailed();
        }
    }

    // Allows the owner of the contract to withdraw a certain amount of ENA tokens from the contract.
    function withdraw() external onlyOwner {
-       bool withdrawSuccess = targetToken.transfer(owner(), targetToken.balanceOf(address(this)));
+        bool withdrawSuccess = targetToken.safeTransfer(owner(), targetToken.balanceOf(address(this)));
        if (!withdrawSuccess) {
            revert TransferFailed();
        }
    }
```

***: Use `safeTransferFrom` in this contract or a low level call and assign its return value to `bool success` instead.
***: Recommend using OpenZeppelin's **SafeERC20** versions with the `safeTransfer` and `safeTransferFrom` functions that handle the return value check as well as non-standard-compliant tokens.

```
@@ -3,6 +3,7 @@ pragma solidity 0.8.20;

 import {Ownable} from "openzeppelin/access/Ownable.sol";
 import {IERC20} from "openzeppelin/token/ERC20/IERC20.sol";
+import {SafeERC20} from "openzeppelin/token/ERC20/utils/SafeERC20.sol";
 import {Pausable} from "openzeppelin/security/Pausable.sol";
 import {ReentrancyGuard} from "openzeppelin/security/ReentrancyGuard.sol";

@@ -51,23 +52,14 @@ contract Conversion is Ownable, Pausable, ReentrancyGuard {
         }

         // Transfer source token to the contract
-        bool sourceTokenTransferSuccess = sourceToken.transferFrom(msg.sender, address(this), _sourceTokenA
mount);
-        if (!sourceTokenTransferSuccess) {
-            revert TransferFailed();
-        }
+        SafeERC20.safeTransferFrom(sourceToken, msg.sender, address(this), _sourceTokenAmount);

         // Transfer equivalent ENA to the user
-        bool targetTokenTransferSuccess = targetToken.transfer(msg.sender, targetTokenAmount);
-        if (!targetTokenTransferSuccess) {
-            revert TransferFailed();
-        }
+        SafeERC20.safeTransfer(targetToken, msg.sender, targetTokenAmount);
     }

     // Allows the owner of the contract to withdraw a certain amount of ENA tokens from the contract.
     function withdraw() external onlyOwner {
-        bool withdrawSuccess = targetToken.transfer(owner(), targetToken.balanceOf(address(this)));
-        if (!withdrawSuccess) {
-            revert TransferFailed();
-        }
+        SafeERC20.safeTransfer(targetToken, owner(), targetToken.balanceOf(address(this)));
     }
```

***:** Use safeTransfer and safeTransferFrom from the SafeERC20 library to ensure that token transfers are handled safely. These functions automatically handle potential errors and reverts, providing a more robust and secure implementation.

Client Response

client response for Daniel526: Fixed. under 8ed9423c, Info level. changed severity to Informational
client response for Daniel526: Fixed. under 8ed9423c, Info level. changed severity to Informational
client response for Saaj: Fixed. under 8ed9423c, Info level. changed severity to Informational
Secure3: . changed severity to Informational
client response for 0xluk3: Fixed. under 8ed9423c, Info level. changed severity to Informational
client response for Rotcivegaf: Fixed. under 8ed9423c, Info level. changed severity to Informational

client response for ginlee: Fixed. under 8ed9423c, Info level.

RST-11:Use `disableInitializers` to prevent front-running on the initialize function

Category	Severity	Client Response	Contributor
Language Specific	Informational	Acknowledged	***

Code Reference

- code/script/upgrade.s.sol#L85

```
85: bytes memory callData = abi.encodeCall(ITransparentUpgradeableProxy.upgradeTo, (implAddress));
```

- code/src/distributes/DistributeMerkleERC20.sol#L106
- code/src/distributes/DistributeMerkleERC20.sol#L106-L119

```
106: function initialize(Init memory init) virtual external initializer {
```

```
106: function initialize(Init memory init) virtual external initializer {
107:         __AccessControlEnumerable_init();
108:         __DistributeMerkleERC20_init(init.pauser, init.uids, init.tokens, init.merkleRoots, init.sponsor
s, init.endTimes);
109:
110:         // set admin roles
111:         _setRoleAdmin(DISTRIBUTOR_MANAGER_ROLE, DEFAULT_ADMIN_ROLE);
112:
113:         // grant roles
114:         if (init.admin == address(0) || init.manager == address(0)) {
115:             revert RoleNotSet();
116:         }
117:         _grantRole(DEFAULT_ADMIN_ROLE, init.admin);
118:         _grantRole(DISTRIBUTOR_MANAGER_ROLE, init.manager);
119:     }
```

- code/src/distributes/DistributeYieldERC20.sol#L94

```
94: function initialize(Init memory init) virtual external initializer {
```

Description

***:

Vulnerability Detail

Uninitialized implementation in contract can be taken over by an attacker with initialize function.
All contracts in scope are are Upgradeable but does not have constructors which makes call to the `_disableInitializers` to have protection from implementation initialized to any version.

Impact

According to OZ'S guideline for protection of initialize function with `_disableInitializers()` method, implementation contracts should not remain uninitalized. Uninitalization can lead to attack where a malicious attacker can take over control of contract.
Ensure prevention of initialization by an attacker which will have a direct impact on the contract as the implementation contract's constructor should have `_disableInitializers()` method .
The `initialize` function of `DistributeMerkleERC20` and `DistributeYieldERC20` contract does not have protection against front running by implementing `_disableInitializers()` method in constructor.
***: `DistributeMerkleERC20.sol` and `DistributeYieldERC20.sol` are both upgradeable contracts. Both of these contracts have used `initialize()` function to initialize the contract.
The issue is that both the contracts have not disabled initializers.
OpenZeppelin states:
Avoid leaving a contract uninitialized.
An uninitialized contract can be taken over by an attacker. This applies to both a proxy and its implementation contract, which may impact the proxy.
To prevent the implementation contract from being used, you should invoke the `_disableInitializers` function in the constructor to automatically lock it when it is deployed.
`DistributeMerkleERC20.sol` and `DistributeYieldERC20.sol` contracts does not invoke the `_disableInitializers` function in their constructors so affected by this issue.
It should be noted that, other upgradeable contracts like `StakingMNT.sol` and `StakingMETH.sol` have called `_disableInitializers` function in their constructors.
***: This finding applies to all upgradeable contracts in this protocol.
script/upgrade.s.sol is the script for upgrading contracts for all contracts in the protocol. In this script, `upgrade()` function calls `TransparentUpgradeableProxy.upgradeTo()`:

```
bytes memory callData = abi.encodeCall(ITransparentUpgradeableProxy.upgradeTo, (implAddress));
```

which comes from OpenZeppelin TransparentUpgradeableProxy.sol. In that library file, we can see the call chain:

```
    function upgradeTo(address newImplementation) external virtual ifAdmin {
        _upgradeTo(newImplementation);
    }

    ...

    function _upgradeTo(address newImplementation) internal virtual {
        _setImplementation(newImplementation);
        emit Upgraded(newImplementation);
    }

    /**
     * @dev Stores a new address in the EIP1967 implementation slot.
     */
    function _setImplementation(address newImplementation) private {
        require(Address.isContract(newImplementation), "ERC1967Proxy: new implementation is not a contrac
t");

        bytes32 slot = _IMPLEMENTATION_SLOT;

        // solhint-disable-next-line no-inline-assembly
        assembly {
            sstore(slot, newImplementation)
        }
    }
```

The problem here is that `initialize()` in each function is not called within a single tx (no atomicity), therefore attacker can frontrun admin's calls to `initialize()` and attacker will become the owner. The result is disastrous since attacker can grant himself `DEFAULT_ADMIN_ROLE` and take over the system completely.

Recommendation

***: Add constructor to `DistributeMerkleERC20` and `DistributeYieldERC20` contracts that calls `_disableInitializers()` method .
***: Add the following code to `DistributeMerkleERC20.sol` and `DistributeYieldERC20.sol` contracts:

```
+ constructor() {
+       _disableInitializers();
+ }
```

***: Use `upgradeToAndCall()` instead of `upgradeTo()`. This function can call `initialize()` within a single tx to prevent the frontrun attack.

Client Response

client response for Saaj: Acknowledged. Info / Recommend level
client response for 0xRizwan: Acknowledged. Info / Recommend level
Secure3: . changed severity to Informational

client response for ret2basic: Acknowledged. Info / Recommend level

RST-12:Tranche is not validated against during allocation

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/src/AllocateRegister.sol#L185
- code/src/AllocateRegister.sol#L196

```
185: function unallocate(
```

```
196: function _allocate(address tranche, address owner, uint256 amount) internal returns (address, uint256, ui
nt256, uint256) {
```

Description

***: During allocation and unallocation, the tranche is included in both functions,

```
function unallocate(
        address tranche,
        address owner,
        uint256 amount
    ) public onlyRole(REGISTER_OPERATOR_ROLE) returns (address, uint256, uint256, uint256) {
        if (pauser.isAllocationPaused()) {
            revert Paused();
        }
        return _unallocate(tranche, owner, amount);
    }

     function allocate(
        address tranche,
        address owner,
        uint256 amount
    ) external onlyRole(REGISTER_OPERATOR_ROLE) returns (address, uint256, uint256, uint256) {
        if (pauser.isAllocationPaused()) {
            revert Paused();
        }
        return _allocate(tranche, owner, amount);
    }
```

The contract also maintains a registry of tranches that it supports in

```
    EnumerableSet.AddressSet internal trancheRegister;
```

But when allocating and unallocating, it does not validate that the tranche is supported and in the registry. Meaning the state of the listed supported tranche in the registry and the tranches in allocations could be inconsistent at any given time.

Recommendation

***: Validate that the tranche parameter included in allocation and unallocation is available in trencheRegistry.

Client Response

client response for 0xffchain: Fixed. under 2c64b81

RST-13:Second Preimage Attack in Merkle Proof Verification using shortned proofs

Category	Severity	Client Response	Contributor
Privilege Related	Informational	Acknowledged	***

Code Reference

- code/src/distributes/DistributeMerkleERC20.sol#L154

```
154: bytes32 node = keccak256(abi.encodePacked(id, index, account, amount));
```

Description

***: The `DistributeMerkleERC20Upgradeable` contract uses a Merkle tree to verify claims for token distributions. The leaf nodes in the Merkle tree are constructed using a single hash: `keccak256(abi.encodePacked(id, index, account, amount))` This approach can be exploited by an attacker using a second preimage attack, where the attacker provides a shortened version of a valid proof and recreates the Merkle root, leading to unauthorized claims.

Impact:

The impact of this vulnerability is significant. An attacker could potentially claim tokens that they are not entitled to by providing a manipulated proof. This could lead to unauthorized token transfers, resulting in financial losses for the contract and its users.

Attack Scenario:

1. Valid Proof:

- Suppose a valid proof for leaf ℓ_2 in the Merkle tree is [h(ℓ_1), h(b), h(f)].
- The verification process involves hashing the concatenated values step-by-step until the root is reached.

2. Attack:

- The attacker provides a = h(h(ℓ_1) + h(ℓ_2)) as a leaf and [h(b), h(f)] as the proof.
- The contract interprets a (which is h(h(ℓ_1) + h(ℓ_2))) as a leaf.
- The proof [h(b), h(f)] is valid, but a is not an original leaf.
- The contract accepts the manipulated proof, allowing the attacker to claim tokens fraudulently.

Openzeppelin Merkle tree library also comes with warning `WARNING: You should avoid using leaf values that are 64 bytes long prior to hashing, or use a hash function other than keccak256 for hashing leaves.` Openzeppelin suggests using double hashing for leafs.

Recommendation

***: To prevent this issue, double hashing of leaf node can be implemented in code as:

- Ensure that leaf nodes are double hashed before constructing the Merkle tree.
- This prevents intermediate nodes from being misinterpreted as leaf nodes.

Example implementation:

```
bytes32 node = keccak256(bytes.concat(keccak256(abi.encodePacked(id, index, account, amount))));
```

he contract is potentially vulnerable to a second preimage attack due to single hashing of leaf nodes. Implementing double hashing of leaf nodes or restricting leaf node lengths will mitigate this vulnerability, ensuring the integrity and security of the Merkle proof mechanism.
you can refer the following article for more info on second preimage attack for Merkle Trees in Solidity.

Client Response

client response for 0xShax2nk: Acknowledged. Info level. changed severity to Informational
Secure3: . changed severity to Informational

RST-14:Redundant imports in `LockupUpgradable.sol` contract

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Fixed	***

Code Reference

- code/src/LockupUpgradable.sol#L6-L8

```
6: import {ContextUpgradeable} from "openzeppelin-upgradeable/utils/ContextUpgradeable.sol";
7:
8: abstract contract LockupUpgradable is ContextUpgradeable {
```

Description

***: `StakingMETH.sol` and `StakingMNT.sol` contracts have inherited `LockupUpgradable.sol` and `BareVaultUpgradable.sol` as base contracts.
Both `LockupUpgradable.sol` and `BareVaultUpgradable.sol` inherits openzeppelin's `ContextUpgradeable.sol` contract.

```
abstract contract LockupUpgradable is ContextUpgradeable {
```

```
abstract contract BareVaultUpgradable is ContextUpgradeable, AccessControlEnumerableUpgradeable {
```

Since, `ContextUpgradeable.sol` is already inherited by BareVaultUpgradable.sol then `ContextUpgradeable.sol` should be removed from `LockupUpgradable.sol` contract.
This would make the code more readable and would reduce the byte size.
Its recommended to remove unused or redundant code.

Recommendation

***: Consider below changes in `LockupUpgradable.sol`

```
import {EnumerableMap} from "openzeppelin/utils/structs/EnumerableMap.sol";
import {EnumerableSet} from "openzeppelin/utils/structs/EnumerableSet.sol";
- import {ContextUpgradeable} from "openzeppelin-upgradeable/utils/ContextUpgradeable.sol";

- abstract contract LockupUpgradable is ContextUpgradeable {
+ abstract contract LockupUpgradable {
```

Client Response

client response for 0xRizwan: Fixed. under cf72f2b

RST-15:Native token may be locked in the contract due to functions being payable

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/src/BareVaultUpgradable.sol#L67

```
67: receive() external payable virtual {}
```

- code/src/StakingMETH.sol#L103
- code/src/StakingMETH.sol#L116

```
103: function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
```

```
116: function depositWithLockup(uint256 assets, uint256 duration) public payable nonReentrant returns (uint25
6) {
```

Description

***: `StakingMETH.sol` is a staking contract which is only supposed to accept Mantle ETH i.e METH token via deposit and withdraw functions.
As per shared documentation on `Staking METH (StakingMETH.sol)`:

> *"Handles staking of METH tokens with cooldown periods to prevent exploitative behavior"*

It means only METH tokens are allowed in `StakingMETH.sol` contract.
It should be further noted that, `StakingMETH.sol` explicitely restricts Native MNT transfer via `receive()` and `fallback()` functions.

```
    receive() external payable override { revert UnacceptedNativeTokenTransfer(); }
    fallback() external payable { revert("Not Allowed"); }
```

The issue is that, `StakingMETH` contract's deposit functions allows to transfer Native MNT tokens. `deposit()` function is implemented as:

```
    function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
```

and `depositWithLockup()` function is implemented as:

```
    function depositWithLockup(uint256 assets, uint256 duration) public payable nonReentrant returns (uint25
6) {
```

Both of these functions are `payable`, it means they accept transfer of Native MNT so when the user tranfer Native MNT via both deposit function, it will be permanently stucked in contract also there is no withdraw function to transfer these stucked Native tokens from `StakingMETH.sol` contract.
Consider a scenario:

1. Alice wants to deposit METH via `StakingMETH.deposit()`. Both function MNT and METH deposit function arguments and signature is same so Alice mistakenly transfers her MNT via `StakingMETH.deposit()` and deposit gets successful as the deposit() function is payable.
2. Alice realize, she wanted to deposit METH but she deposited MNT so she tries to withdraw her mistakenly deposited MNT via `StakingMETH.withdraw()` function but the function reverts as Alice deposited tokens are not METH.
3. Now, Alice deposited Native MNT are permanently stucked in StakingMETH contract.
4. This is due to presence of payable on StakingMETH deposit functions.

It should be noted that, METH is an ERC20 token and for its transfer across contracts/wallets, a `payable` keyword is not required.
***:** In the `StakingMETH` contract, the `deposit` and `depositWithLockup` functions are designed to handle deposits of ERC20 tokens. However, both functions are marked with the payable keyword, which is unnecessary and potentially misleading. The payable keyword is used to indicate that a function can receive native tokens, but since these functions do not deal with native tokens but with the ERC20 assets, they should not be marked as payable.
***:** The functions `deposit` and `depositWithLockup` have been marked as `payable`. While marking functions as `payable` can sometimes save a small amount of gas, it can lead to unintended consequences.
In this context. Specifically, if users send native gas tokens (such as Mantle) to the contract along with the specified assets, these tokens will become permanently locked in the contract.
***:** Any vault inherits from BareVaultUpgradable.sol will have a `receive()` function. If user sends native tokens to such vault, the tokens will be stuck in the vault since the contract did not record this info in its accounting system and there is no admin-controlled function to withdraw native tokens.
The intended way to use vault, for example StakingMNT.sol, is to send native tokens along with calls to `deposit()` or `depositWithLockup()`. The `receive()` function isn't needed in this design.
***:** The contract `StakingMETH.sol` is not meant to receive any native tokens. Firstly, it has a safeguard against this in `receive()` function and secondly, there is no any logic that could help to withdraw any stuck funds later. On the other hand, any native token received, may be only due to an user error. However, since the design of the protocol prohibits receiving of native token, the `payable` keyword should be removed from those functions.

Recommendation

***:** Consider removing `payable` from `StakingMETH.sol` deposit functions.
Consider below changes:

```
-     function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
+     function deposit(uint256 assets) public override nonReentrant returns (uint256) {
```

and

```
-     function depositWithLockup(uint256 assets, uint256 duration) public payable nonReentrant returns (uint
256) {
+     function depositWithLockup(uint256 assets, uint256 duration) public nonReentrant returns (uint256) {
```

***: Remove the payable keyword from the `deposit` and `depositWithLockup` functions to accurately reflect their intended use and prevent any unintended transfers.
***: Remove the `payable` modifier from both `deposit` and `depositWithLockup` functions. This change will prevent users from inadvertently sending native gas tokens to the contract, thus avoiding the risk of token lockup.
***: Delete the `receive()` function in BareVaultUpgradable.sol.
***: Remove the `payable` keyword from those functions.

Client Response

client response for 0xRizwan: Fixed. under 4f4cb64c
client response for 0xShax2nk: Fixed. under 4f4cb64c
client response for 0xzoobi: Fixed. under 4f4cb64c
client response for ret2basic: Fixed. under 4f4cb64c
client response for 0xluk3: Fixed. under 4f4cb64c

RST-16:Missing Pause Check

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/src/StakingMETH.sol#L103-L114
- code/src/StakingMETH.sol#L103
- code/src/StakingMETH.sol#L186

```
103: function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
104:         if (assets < minStake) {
105:             revert DepositAmountTooSmall(_msgSender(), assets, minStake);
106:         }
107:         if (maxStakeSupply != 0 && assets + totalDeposit() > maxStakeSupply) {
108:             revert DepositOverBond();
109:         }
110:         _userStakeCooldown[_msgSender()] = block.timestamp;
111:         emit DepositWithDuration(_msgSender(), block.timestamp, assets, 0);
112:
113:         return super.deposit(assets);
114:     }
```

```
103: function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
```

```
186: }
```

Description

***:

Vulnerability Detail

`deposit` in `StakingMETH` contract does not have check for either staking is paused or not.

```
function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
    if (assets < minStake) {
        revert DepositAmountTooSmall(_msgSender(), assets, minStake);
    }
    if (maxStakeSupply != 0 && assets + totalDeposit() > maxStakeSupply) {
        revert DepositOverBond();
    }
    _userStakeCooldown[_msgSender()] = block.timestamp;
    return super.deposit(assets);
}
```

This can lead to a user depositing amount into contract when it is paused that may have negative impact on funds deposited as protocols are paused either when a critical vulnerability is discovered or the protocol is hacked.
***: `StakingMETH.deposit()` lacks the pauser check:

```
    if (pauser.isStakingPaused()) {
        revert Paused();
    }
```

Therefore it can't be paused.
When pauser is in paused state, user can still call `StakingMETH.deposit()` to stake but can't withdraw since `StakingMETH.withdraw()` has the pauser check.
***: **Description**:
The `StakingMETH` contract implements a mechanism to pause staking activities using the pauser contract. While the `withdraw` and `depositWithLockup` function correctly checks if staking is paused and reverts if it is, the deposit function lacks this critical check. This inconsistency can lead to new deposits being accepted even when the staking process is paused, potentially undermining the purpose of the pause functionality.
Impact:

- Allowing deposits while staking is paused can expose the protocol to risks that the pause mechanism aims to mitigate, such as potential exploits or vulnerabilities.
- The lack of uniformity in the pause checks across functions can lead to confusion and unexpected behavior for users and developers.
- Accepting new deposits during a pause can affect the protocol's stability and integrity, especially if the pause was initiated to address an ongoing issue or exploit.

***: The `deposit` function lacks a check for the staking pause status, which may result in staking operations being allowed during the pause period.

```
    function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
        if (assets < minStake) {
            revert DepositAmountTooSmall(_msgSender(), assets, minStake);
        }
        if (maxStakeSupply != 0 && assets + totalDeposit() > maxStakeSupply) {
            revert DepositOverBond();
        }
        _userStakeCooldown[_msgSender()] = block.timestamp;
        emit DepositWithDuration(_msgSender(), block.timestamp, assets, 0);

        return super.deposit(assets);
    }
```

***: `StakingMETH.deposit()` is used to deposit the Mantle ETH i.e METH in contract and its implemented as:

```
    function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
        if (assets < minStake) {
            revert DepositAmountTooSmall(_msgSender(), assets, minStake);
        }
        if (maxStakeSupply != 0 && assets + totalDeposit() > maxStakeSupply) {
            revert DepositOverBond();
        }
        _userStakeCooldown[_msgSender()] = block.timestamp;
        emit DepositWithDuration(_msgSender(), block.timestamp, assets, 0);

        return super.deposit(assets);
    }
```

The issue is that, `deposit()` does not check whether staking is paused or not. This would break the core functionality as the staking pauser mechansim is intended to ensure its compliance in `StakingMETH.sol` contract. As per client shared documentation,

"4. Staking METH (StakingMETH.sol):"

"◦ Ensures compliance with pauser state."

It particularly states that, StakingMETH.sol complies with pauser state, however deposit() function does not comply it, Therefore, the core intended design for pauser mechanism is broken here.
It should be noted that, `StakingMETH.depositWithLockup() explicitely checks for staking pause state in its function.

```
    function depositWithLockup(uint256 assets, uint256 duration) public payable nonReentrant returns (uint25
6) {
        if (pauser.isStakingPaused()) {
            revert Paused();
        }
   . . . some code . . .
}
```

Similarly, deposit() function must also check it.
Consider a scenario:

1. The protocol team has decided to pause the staking so they pause staking by calling relevant function in Pauser.sol contract. Now, the protocol assumes that all staking i.e deposits as well as well withdraws are paused as these funtions checks pauser.isStakingPaused() function.
2. Alice wants to deposit some METH so she calls `StakingMETH.deposit()` and deposits the METH and transaction gets successful.
3. Alice transaction should have reverted with error `paused` but it didn't, This is because of this issue which is `StakingMETH.deposit()` does not check staking pause state.
4. This is how, the core pause staking design is broken in deposit() function.

Recommendation

***:** The recommendation is made to add check for paused staking to prevent user from depositing into contract when it is paused like in `depositWithLockup` function.

```
    function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
+       if (pauser.isStakingPaused()) {
+           revert Paused();
+       }
        if (assets < minStake) {
            revert DepositAmountTooSmall(_msgSender(), assets, minStake);
        }
        if (maxStakeSupply != 0 && assets + totalDeposit() > maxStakeSupply) {
            revert DepositOverBond();
        }
        _userStakeCooldown[_msgSender()] = block.timestamp;
        return super.deposit(assets);
    }
```

***:** Add check:

```
    function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
      + if (pauser.isStakingPaused()) {
      +     revert Paused();
      + }
        if (assets < minStake) {
            revert DepositAmountTooSmall(_msgSender(), assets, minStake);
        }
        if (maxStakeSupply != 0 && assets + totalDeposit() > maxStakeSupply) {
            revert DepositOverBond();
        }
        _userStakeCooldown[_msgSender()] = block.timestamp;
        return super.deposit(assets);
    }
```

***:** To ensure the protocol's security and consistency, the deposit function should include a check to verify if staking is paused. If staking is paused, the function should revert with the Paused error. This change will align the behavior of the deposit function with the withdraw function and uphold the intended functionality of the pause mechanism.
Add the pauser.isStakingPaused() check in the deposit function as follows:

```
function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
    if (pauser.isStakingPaused()) {
        revert Paused();
    }
    if (assets < minStake) {
        revert DepositAmountTooSmall(_msgSender(), assets, minStake);
    }
    if (maxStakeSupply != 0 && assets + totalDeposit() > maxStakeSupply) {
        revert DepositOverBond();
    }
    _userStakeCooldown[_msgSender()] = block.timestamp;
    emit DepositWithDuration(_msgSender(), block.timestamp, assets, 0);

    return super.deposit(assets);
}
```

***: add

```
if (pauser.isStakingPaused()) {
    revert Paused();
}
```

***: Check the staking pause state in deposit() funcion.
Consider below changes:

```
    function deposit(uint256 assets) public payable override nonReentrant returns (uint256) {
+       if (pauser.isStakingPaused()) {
+           revert Paused();
+       }
        if (assets < minStake) {
            revert DepositAmountTooSmall(_msgSender(), assets, minStake);
        }
        if (maxStakeSupply != 0 && assets + totalDeposit() > maxStakeSupply) {
            revert DepositOverBond();
        }
        _userStakeCooldown[_msgSender()] = block.timestamp;
        emit DepositWithDuration(_msgSender(), block.timestamp, assets, 0);

        return super.deposit(assets);
    }
```

Client Response

client response for Saaj: Fixed. under 6484532
client response for ret2basic: Fixed. under 6484532
client response for 0xShax2nk: Fixed. under 6484532
client response for 8olidity: Fixed. under 6484532
client response for 0xRizwan: Fixed. under 6484532

RST-17:Exchanging exact minimum amount will still revert due to an off by one error

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/src/rewards/StandardConversion.sol#L40

```
40: if (_sourceTokenAmount <= minExchangeAmount) {
```

Description

***: The amount check of an exchanged token is done in a following way:

```
if (_sourceTokenAmount <= minExchangeAmount) {
            revert OutOfBound();
        }
```

Which means that the revert will happen if the input is `lower or equal` to the minimum amount. This degrades user experience, since user will not be allowed to exchange even though they input proper minimum amount.

Recommendation

***: Change the code to:

```
if (_sourceTokenAmount < minExchangeAmount) {
            revert OutOfBound();
        }
```

Client Response

client response for 0xluk3: Fixed. under commit 8905911

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.