Nexus Security Review

Pashov Audit Group

Conducted by: ast3ros, merlinboii, 0xunforgiven

November 29th - December 3rd

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Nexus-2023/NativeYield** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Nexus

Nexus allows users to deposit and earn native yields on any L2 of their choice while Nexus finds the best strategies for them to earn yield.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - a2c729ef6523d622510b17d2f87533d66cd70090

fixes review commit hash - df7a1d651891904c0dde5fdbd0982a148ad75e6e

Scope

The following smart contracts were in scope of the audit:

- `DepositETHBera`
- `DepositUSDBera`
- `MessagingBera`
- `nETH`
- `nUSD`
- `AddressLibrary`
- `BytesArrayLibrary`
- `DSRStrategy`
- `LiDoStrategy`
- `WETHStrategy`
- `DepositETH`
- `DepositUSD`
- `Messaging`
- `StrategyManager`

7. Executive Summary

Over the course of the security review, ast3ros, merlinboii, 0xunforgiven engaged with Nexus to review Nexus. In this period of time a total of **28** issues were uncovered.

Protocol Summary

Protocol Name	Nexus
Repository	https://github.com/Nexus-2023/NativeYield
Date	November 29th - December 3rd
Protocol Type	Yield Aggregator

Findings Count

Severity	**Amount**
Critical	3
High	6
Medium	6
Low	13
Total Findings	**28**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Wrong token transfer in executeWithdrawal()	Critical	Resolved
[C-02]	Incorrect reward calculation due to decimal handling	Critical	Resolved
[C-03]	Inaccurate share price due to lack of withdrawal tracking	Critical	Resolved
[H-01]	Incorrect access control logic in onlyDeposit modifier	High	Resolved
[H-02]	Lack of approval before depositing DAI into DSR contract	High	Resolved
[H-03]	Lack of withdrawal functionality in strategy contracts	High	Acknowledged
[H-04]	Stale share price usage leading to fund loss and unfair distribution	High	Acknowledged
[H-05]	Arithmetic underflow in total balance calculation	High	Resolved
[H-06]	Share price manipulation	High	Resolved
[M-01]	LayerZero Fee refunds cannot be processed	Medium	Resolved
[M-02]	Potential hash collision in withdrawal request generation	Medium	Resolved
[M-03]	Potential protocol insolvency due to lido slashing events	Medium	Acknowledged
[M-04]	LayerZero fee refunds misdirected to deposit contracts	Medium	Resolved

[M-05]	Incorrect TOKEN_ADDRESS and DSR_DEPOSIT_ADDRESS	Medium	Resolved
[M-06]	Wrong fee estimate in getlzFee() function	Medium	Resolved
[L-01]	Withdrawal replay attack	Low	Resolved
[L-02]	Insufficient address validation in convertToAddress	Low	Resolved
[L-03]	The stETH received can be less	Low	Acknowledged
[L-04]	BERA native gas token can stuck	Low	Resolved
[L-05]	Invalid address string can become valid and be used in the operation	Low	Resolved
[L-06]	Precision loss in share-to-asset conversion	Low	Resolved
[L-07]	Uninitialized proxy share price	Low	Resolved
[L-08]	Fixed gas low-level operations pose the risk of transaction failure	Low	Resolved
[L-09]	Function executeWithdrawal() doesn't follow CEI pattern	Low	Resolved
[L-10]	Function _updateSharePrice() in L2 deposit contracts never get called from L1	Low	Resolved
[L-11]	Front-running vulnerability in reward distribution	Low	Acknowledged
[L-12]	Implementation contracts allow unauthorized Initialization	Low	Acknowledged
[L-13]	Users who deposit DAI directly will lose their deposits	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Wrong token transfer in `executeWithdrawal()`

Severity

Impact: High

Likelihood: High

Description

The `executeWithdrawal` function in the `DepositUSD` contract is incorrectly implemented to send ETH instead of the USD tokens (e.g., DAI, sDAI) that users originally deposited.

The contract attempts to send ETH using `call{value: amount}` even though it's designed to handle stablecoins:

```
function executeWithdrawal(address _receiver, bytes32 _hash) external {
        if(msg.sender!=strategyExecutor) revert IncorrectStrategyExecutor
          (msg.sender);
        (bool success,bytes memory returndata) = _receiver.call{
            value:withdrawals[_receiver][_hash].amount,gas:10000
            }(""); // @audit send eth instead of DAI
        if(!success){
            if (returndata.length == 0) revert CallFailed();
            assembly {
                revert(add(32, returndata), mload(returndata))
            }
        }
        withdrawals[_receiver][_hash].withdrawalExecutionTime = block.timestamp;
        emit WithdrawalCompleted(_receiver, _hash);
    }
```

The impacts are:

- The contract doesn't hold any ETH, so the transfer will always fail
- User funds become effectively locked in the contract
- The intended tokens such as DAI or sDAI tokens remain locked in the contract

Recommendations

Reimplement the `executeWithdrawal` function to send the correct token to the user.

[C-02] Incorrect reward calculation due to decimal handling

Severity

Impact: High

Likelihood: High

Description

In the DSRStrategy contract's `getRewards` function, there is a decimal handling error when calculating asset balances and rewards. The function multiplies the token balance by the conversion rate without accounting for decimals, leading to severely inflated reward calculations.

```
function getRewards(
      addressreceiver,
      IStrategyManager.StrategyStructmemory_strategyBalance
    ) public override view returns(uint256
        uint256 balance_strategy = _strategyBalance.valueDeposited
            + _strategyBalance.rewardsEarned -_strategyBalance.valueWithdrawn;
        uint256 assetBalance = IERC20(TOKEN_ADDRESS).balanceOf(receiver)
            * sDAI(TOKEN_ADDRESS).convertToAssets(1e18);
        if (assetBalance>balance_strategy){
            return assetBalance - balance_strategy;
        }
        else{
            return 0;
        }
    }
```

The impact is reward amount is inflated by a factor of 1e18. It causes incorrect share price calculations in the deposit contracts. And new depositors receive

fewer shares than they should due to artificially high share prices. Early depositors gain an unfair advantage at the expense of later depositors.

Recommendations

Modify the `getRewards` function to properly handle decimals:

```
- uint256 assetBalance = IERC20(TOKEN_ADDRESS).balanceOf(receiver)*sDAI
- (TOKEN_ADDRESS).convertToAssets(1e18);
+ uint256 assetBalance = IERC20(TOKEN_ADDRESS).balanceOf(receiver)*sDAI
+ (TOKEN_ADDRESS).convertToAssets(1e18) / 1e18;
```

[C-03] Inaccurate share price due to lack of withdrawal tracking

Severity

Impact: High

Likelihood: High

Description

The `DepositETH` and `DepositUSD` contracts fail to update the withdrawal counters (`ethWithdrawn`/`stableWithdrawn`) and shares (`nETHMinted`, `nUSDMinted`) when processing withdrawals.

This leads to incorrect accounting because the share price will be calculated by including the portion of withdrawn funds in the total balance.

The share price is calculated based on the total balance:

```
function _updateSharePrice(uint256 _rewardsEarned) internal {
    rewardsClaimed+=_rewardsEarned;
    uint256 totalEthBalance = (ethDeposited-ethWithdrawn+rewardsClaimed);
    if(nETHMinted==0){
        sharePrice = BASE_POINT;
    } else {
        sharePrice = (nETHMinted)*BASE_POINT/
          (ethDeposited-ethWithdrawn+rewardsClaimed);
    }
    emit SharePriceUpdated(sharePrice);
}
```

This can lead to incorrect share price calculations, causing new depositors to receive fewer rewards, as the shares include withdrawn funds from the account of a previous withdrawer.

Let's consider this scenario:

1. Initial State:
   - `sharePrice` = `1e18`
   - All balances = `0` ETH
2. Alice Deposits `1` ETH:
   - Gets `1e18` shares
   - `ethDeposited` = `1e18`
   - `nETHMinted` = `1e18`
3. System Earns `0.05` ETH Reward:
   - `rewardsClaimed` = `5e16` (0.05 ETH)
   - `totalEthBalance` = `1.05` ETH
   - `sharePrice` updates = `1e18 * 1e18 / 1.05 ETH = ~0.952e18`
4. Alice Withdraws Everything:
   - Withdraws `~1.05` ETH (original + rewards)
   - **But shares and ethWithdrawn not updated**
   - `sharePrice` calculation still includes withdrawn amount
5. Carol Deposits `1` ETH:
   - Gets `0.952e18` shares: `nETHShares` = `1e18 * 0.952e18 / 1e18` = `0.952e18`
   - `nETHMinted` += `0.952e18` = `1.952e18`
   - `ethDeposited` += `1e18` = `2e18`
   - `CAROL_SHARES` = `0.952e18`
6. System Earns Another `0.05` ETH Reward:

- `rewardEarned` = `5e16` (0.05 ETH)
- `rewardClaimed` += `5e16` = `1e17`
- `totalEthBalance` = `ethDeposited - ethWithdrawn + rewardClaimed`
- `totalEthBalance` = `2e18 - 0 + 1e17` = `2.1e18`
- New `sharePrice` = `1.952e18 * 1e18 / 2.1e18` = `0.929e18`

7. Carol Claims Withdrawal:

- `withdrawAmount` = `0.952e18 * 1e18 / 0.929e18` = `1.024e18`
- Carol receives `1.024` ETH for her `1` ETH deposit,
- even though she should have received all rewards because she held all available shares.

Recommendations

Implement proper tracking of withdrawals and shares when funds are withdrawn.

```
-function _withdrawFunds(address _receiver, uint256 _value) internal {
+function _withdrawFunds
+ (address _receiver, uint256 _value, uint256 _shares) internal {
    bytes32 withdrawalHash = keccak256(abi.encodePacked
      (_receiver,_value,block.timestamp));
    WithdrawalStruct memory newWithdrawal;
    newWithdrawal.amount = _value;  //@audit withdraw amount
+    newWithdrawal.shares = _shares;  //@audit withdraw shares
    newWithdrawal.withdrawalCreationTime = block.timestamp;
    withdrawals[_receiver][withdrawalHash] = newWithdrawal;
    emit WithdrawalRequestCreated(_receiver,withdrawalHash,newWithdrawal);
}
```

```
function executeWithdrawal(address _receiver, bytes32 _hash) external {
    uint256 amount = withdrawals[_receiver][_hash].amount;
    uitn256 sharesToBurn = withdrawals[_receiver][_hash].shares;

    // Update state before transfer
    ethWithdrawn += amount;
    nETHMinted -= sharesToBurn;
---
}
```

Additionally, the `_value` when converting shares to assets can suffer from precision loss due to integer division, which may lead to leftover dust in nETHMinted if not handled carefully.

8.2. High Findings

[H-01] Incorrect access control logic in `onlyDeposit` modifier

Severity

Impact: Medium

Likelihood: High

Description

The `onlyDeposit` modifier in both Messaging and MessagingBera contracts contains a logical error in its access control implementation that leads to complete denial of service.

```
modifier onlyDeposit(){
        if(msg.sender!=depositETHBera
            || msg.sender!=depositUSDBera) revert NotDepositContract
              (msg.sender);
        _;
    }
```

```
modifier onlyDeposit(){
        if(msg.sender!=depositETH
            || msg.sender!=depositUSD) revert NotDepositContract(msg.sender);
        _;
    }
```

The logical issue is that the condition `msg.sender != A || msg.sender != B` will always evaluate to true for any msg.sender value, because any address must be different from at least one of the two deposit addresses. This means:

- If msg.sender equals `depositETH`, it must not equal `depositUSD`
- If msg.sender equals `depositUSD`, it must not equal `depositETH`
- If msg.sender doesn't equal either, both conditions are true

Therefore, the revert will be triggered for all calls, including legitimate ones from the authorized deposit contracts, making the `sendMessage` function completely unusable.

Recommendations

Replace the OR (||) operator with AND (&&) to implement the intended access control:

```
- if(msg.sender!=depositETHBera
    || msg.sender!=depositUSDBera) revert NotDepositContract(msg.sender);
+ if(msg.sender!=depositETHBera
    && msg.sender!=depositUSDBera) revert NotDepositContract(msg.sender);
```

[H-02] Lack of approval before depositing DAI into DSR contract

Severity

Impact: Medium

Likelihood: High

Description

The DSRStrategy contract's `deposit` function attempts to deposit DAI tokens into the DSR (Dai Savings Rate) contract to receive sDAI, but fails to implement the required ERC20 approval step. Since ERC20 tokens require approval before a contract can transfer them on behalf of the owner, it leads to deposit transactions to revert.

```
function deposit(uint256 _value) public returns(uint256){
        sDAI(DSR_DEPOSIT_ADDRESS).deposit(_value,address(this));
        return 100;
    }
```

Recommendations

Add the necessary approval step before attempting the deposit:

```
function deposit(uint256 _value) public returns(uint256){
+       IERC20(DAI_ADDRESS).approve(DSR_DEPOSIT_ADDRESS, _value);
        sDAI(DSR_DEPOSIT_ADDRESS).deposit(_value,address(this));
        return 100;
    }
```

[H-03] Lack of withdrawal functionality in strategy contracts

Severity

Impact: High

Likelihood: Medium

Description

The `LidoStrategy` and `DSRStrategy` contracts are responsible for managing `ETH` and `DAI` deposits into `Lido` and `sDAI`, respectively, to earn yields.

These strategies are integrated with the `DepositETH` and `DepositUSD` contracts, which handle user deposits and facilitate interactions with `Lido` and `sDAI`.

```
function executeStrategies
  (StrategyExecution[] memory _executionData) external override {
    if(msg.sender!=strategyExecutor) revert IncorrectStrategyExecutor
      (msg.sender);
    unchecked{
        for(uint i=0;i<_executionData.length;i++){
            if(!IStrategyManager(strategyManager)
                .strategyExists
                  (_executionData[i].strategy)) revert IncorrectStrategy(_executionDat
@>          (bool success, bytes memory returndata) =
                _executionData[i].strategy.delegatecall
                  (_executionData[i].executionData);
            if(!success){
                if (returndata.length == 0) revert CallFailed();
                assembly {
                    revert(add(32, returndata), mload(returndata))
                }
            }
---
}
```

However, the `LidoStrategy` and `DSRStrategy` contracts lack withdrawal functionality. As a result, the `DepositETH` and `DepositUSD` contracts cannot retrieve deposited tokens along with their accumulated rewards.

This limitation prevents the protocol from redeeming assets when needed, leaving it unable to maintain reserves required to fulfill users' share redemptions, ultimately resulting in users losing access to their funds.

Recommendations

Implement withdrawal functionality to the `LidoStrategy` and `DSRStrategy` contracts, enabling the protocol to retrieve deposited tokens along with accumulated rewards.

Moreover, for the `Lido` strategy, the withdrawal from Lido (primary market) requires the request and claim process to integrate with their withdrawal queue so the protocol will need to handle the withdrawal request and claim process or choose to integrate withdrawal with the secondary market.

[H-04] Stale share price usage leading to fund loss and unfair distribution

Severity

Impact: High

Likelihood: Medium

Description

The share conversion is a critical operation in the `DepositETH` and `DepositUSD` contracts, where deposits and withdrawals are based on the current share price. The share price can fluctuate over time due to the yield received from `Lido` and `sDAI`, respectively.

However, the `DepositETH` and `DepositUSD` contracts do not properly update the share price when users perform deposits, withdrawals, or quote LayerZero fees. This failure to update the share price can result in the use of stale share prices, leading to a loss of funds and unfair share distribution for users.

```
function _depositETH(
  stringmemory_receiver,
  uint256_value,
  uint32_destID,
  uint256_lzFee
) internal {
    if(msg.value==_value+_lzFee){
@>      uint256 nETHShares= (sharePrice*_value)/BASE_POINT;
        bytes memory data = abi.encode(ASSET_ID,1, _receiver,nETHShares);
        ethDeposited+=_value;
        nETHMinted+=nETHShares;
        IMessaging(messageApp).sendMessage{value:_lzFee}(data, _destID,_lzFee);
    }else{
        revert IncorrectValue();
    }
}
```

```
function messageReceivedL2(
  uint256_id,
  address_receiver,
  uint256_value
) external override onlyMessageApp {
    if(_id==3){
@>      uint256 _valueUSDCWithdraw = _value*BASE_POINT/sharePrice;
        _withdrawFunds(_receiver, _valueUSDCWithdraw);
    }
    else {
        revert IncorrectTypeID(_id,msg.sender);
    }
}
```

For example, during a withdrawal, if rewards accrue (thus changing the share price) between the time of the withdrawal request and its execution, the user will calculate the withdrawal amount using an outdated share price, which does not account for the new rewards. This can result in users withdrawing fewer funds than they are entitled to.

Recommendations

Consider updating the functions that involve share calculations, such as deposits, withdrawals, and LayerZero fee quotes, to ensure that the share price is recalculated and updated during each crucial action.

[H-05] Arithmetic underflow in total balance calculation

Severity

Impact: High

Likelihood: Medium

Description

The `_updateSharePrice()` in `DepositETH` calculates the total ETH balance using an arithmetic expression that can underflow due to operation precedence.

```
function _updateSharePrice(uint256 _rewardsEarned) internal {
    rewardsClaimed+=_rewardsEarned;
@>  uint256 totalEthBalance = (ethDeposited-ethWithdrawn+rewardsClaimed);
    if(nETHMinted==0){
        sharePrice = BASE_POINT;
    }
    else{
@>      sharePrice = (nETHMinted)*BASE_POINT/
  (ethDeposited-ethWithdrawn+rewardsClaimed);
    }
    emit SharePriceUpdated(sharePrice);
}
```

The expression

```
ethDeposited-ethWithdrawn+rewardsClaimed
```

performs operations from left to right, which can cause an underflow when `ethWithdrawn` is greater than `ethDeposited`, even if `rewardsClaimed` would make the total balance positive.

This is particularly problematic because `ethWithdrawn` naturally grows larger than `ethDeposited` in normal protocol operation, as the withdrawal tracking amount includes both principal and earned rewards.

`DepositUSD._updateSharePrice()` shares the same issue.

Recommendations

Reorder arithmetic operations to prevent underflow:

```
function _updateSharePrice(uint256 _rewardsEarned) internal {
    rewardsClaimed += _rewardsEarned;
-    uint256 totalEthBalance = (ethDeposited-ethWithdrawn+rewardsClaimed);
+    uint256 totalEthBalance = ethDeposited + rewardsClaimed - ethWithdrawn;

    if(nETHMinted == 0) {
        sharePrice = BASE_POINT;
    } else {
-       sharePrice = (nETHMinted)*BASE_POINT/
- (ethDeposited-ethWithdrawn+rewardsClaimed);
+       sharePrice = (nETHMinted * BASE_POINT) / totalEthBalance;
    }
    emit SharePriceUpdated(sharePrice);
}
```

[H-06] Share price manipulation

Severity

Impact: High

Likelihood: Medium

Description

The protocol is vulnerable to a share price manipulation attack that can result in permanent loss of user deposits. An attacker can exploit the share price calculation mechanism in both the ETH and USD vaults by manipulating the initial state through a small "donation" of rewards.

The attack can be executed through the following steps:

1. When the vault is newly deployed (`nETHMinted` = 0), an attacker deposits a minimal amount (e.g. 100 wei) of stETH to the `DepositETH` contract
2. The attacker triggers `updateRewards`, which calls the vulnerable `getRewards` function in `LidoStrategy`:

```
function getRewards(
      addressreceiver,
      IStrategyManager.StrategyStructmemory_strategyBalance
    ) public override view returns(uint256

                uint256 balance_strategy = _strategyBalance.valueDeposited +_strategy
        if (IERC20(TOKEN_ADDRESS).balanceOf(receiver)>balance_strategy){
            return IERC20(TOKEN_ADDRESS).balanceOf
            //(receiver) - balance_strategy; // @audit donate so rewards > 0
        }
        else{
            return 0;
        }
    }
```

3. This causes `rewardsClaimed` > 0, leading to:
- `totalEthBalance` > 0 while `nETHMinted` = 0
- `sharePrice` = (0 * BASE_POINT) / (totalEthBalance) = 0

```
function _updateSharePrice(uint256 _rewardsEarned) internal {
        rewardsClaimed+=_rewardsEarned;
        uint256 totalEthBalance = (ethDeposited-ethWithdrawn+rewardsClaimed);
        if (totalEthBalance==0){
            sharePrice = BASE_POINT;
        }
        else{
            sharePrice = (nETHMinted)*BASE_POINT/
            //(ethDeposited-ethWithdrawn+rewardsClaimed); // @audit nETHMinted = 0 lea
        }
        emit SharePriceUpdated(sharePrice);
    }
```

Next, when a legitimate user deposit for example 100 ETH, because the `sharePrice` is 0, the `nETHShares` that user would receive is also 0. In other words, all subsequent user deposits receive 0 share.

```
function _depositETH(
      stringmemory_receiver,
      uint256_value,
      uint32_destID,
      uint256_lzFee
    ) internal {
        if(msg.value==_value+_lzFee){
            uint256 nETHShares=
            //(sharePrice*_value)/BASE_POINT; // @audit number of shares minted = 0
            bytes memory data = abi.encode(ASSET_ID,1, _receiver,nETHShares);
            ...
    }

    function _depositERC(
      address_tokenAddress,
      stringmemory_receiver,
      uint256_value,
      uint32_destID,
      uint256_lzFee
    ) internal onlyWhitelisted(_tokenAddress
        ...
        uint256 nETHShares=
        //(sharePrice*assetValue)/BASE_POINT; // @audit number of shares minted = 0
        bytes memory data = abi.encode(ASSET_ID,1, _receiver,nETHShares);
        ...
    }
```

The same vulnerability exists in the USD vault when using DSRStrategy.

It leads to users lose 100% of their deposited funds.

Recommendations

Admin mints initial amount of shares to avoid the donation attack.

8.3. Medium Findings

[M-01] LayerZero Fee refunds cannot be processed

Severity

Impact: Medium

Likelihood: Medium

Description

The `DepositUSD`, `DepositETHBera`, and `DepositUSDBera` contracts are intended to serve as the refund address for LayerZero, the messaging protocol responsible for paying fees in the native gas token. However, these contracts are unable to receive these refunds, causing a revert when excess fees are sent.

Although the `DepositUSD`, `DepositETHBera`, and `DepositUSDBera` contracts act as implementation contracts, the refunds are transferred to the Proxy contract. The standard ERC1697Proxy implements the fallback function to forward all calls, making a delegate call to the implementation contract.

Therefore, it is expected that the implementation contract should provide functionality to receive native gas transfers.

Recommendation

Consider adding `receive()` functions to the `DepositUSD`, `DepositETHBera`, and `DepositUSDBera` contracts if these contracts are intended to receive users' refund excess fees.

[M-02] Potential hash collision in withdrawal request generation

Severity

Impact: High

Likelihood: Low

Description

The `_withdrawFunds` function in both `DepositETH` and `DepositUSD` contracts generates a withdrawal hash that could potentially collide if multiple withdrawal requests are made by the same user in the same block. The hash is generated using only the receiver address, withdrawal amount, and block timestamp:

```
function _withdrawFunds(address _receiver, uint256 _value) internal {
        bytes32 withdrawalHash = keccak256(abi.encodePacked
        //(_receiver,_value,block.timestamp)); // @audit same hash if two withdrawals
        ...
        emit WithdrawalRequestCreated(_receiver,withdrawalHash,newWithdrawal);
    }
```

If a user submits multiple withdrawal requests in the same block:

- The same hash would be generated for different withdrawal requests
- The second withdrawal would overwrite the first one in the `withdrawals` mapping
- One of the withdrawals would be lost since `executeWithdrawal` sets `withdrawalExecutionTime` using this hash
- This could result in users losing funds if the system processes only one of multiple intended withdrawals

Recommendations

Using a globally incrementing withdrawal ID:

```
uint256 withdrawalId = ++withdrawalCounter;
    bytes32 withdrawalHash = keccak256(abi.encodePacked(
        _receiver,
        _value,
        block.timestamp,
        withdrawalId
    ));
```

[M-03] Potential protocol insolvency due to lido slashing events

Severity

Impact: High

Likelihood: Low

Description

The `DepositETH` fails to properly handle potential reductions in the Lido stETH balance due to slashing events. This results in an inflated share price, which can lead to users being able to withdraw more than the actual funds in the protocol, risking insolvency.

To demonstrate the issue, the `DepositETH.updateRewards()` currently only records positive rewards, but it doesn't handle the scenario where the `stETH` balance may decrease due to slashing or other negative events.

This means that the rewards are never reduced, and the share price is calculated based on an inflated reward value.

The `LidoStrategy.getRewards()` only considers positive rewards, which ignores situations where the stETH balance decreases due to slashing or other negative events.

```
// LidoStrategy.sol
function getRewards(
  addressreceiver,
  IStrategyManager.StrategyStructmemory_strategyBalance
) public override view returns(uint256

         uint256 balance_strategy = _strategyBalance.valueDeposited +_strategyBalance.
@>     if (IERC20(TOKEN_ADDRESS).balanceOf(receiver)>balance_strategy){
        return IERC20(TOKEN_ADDRESS).balanceOf(receiver) - balance_strategy;
    }
    else{
        return 0;
    }
}
```

The share price is calculated based on the `rewardsClaimed`, which is not reduced in cases of negative rewards (e.g., slashing). This leads to an inflated share price, resulting in users being able to withdraw more funds than actually available.

```
function _updateSharePrice(uint256 _rewardsEarned) internal {
@>    rewardsClaimed+=_rewardsEarned;
    uint256 totalEthBalance = (ethDeposited-ethWithdrawn+rewardsClaimed);
    if (totalEthBalance==0){
        sharePrice = BASE_POINT;
    }
    else{
@>      sharePrice = (nETHMinted)*BASE_POINT/
  (ethDeposited-ethWithdrawn+rewardsClaimed);
    }
    emit SharePriceUpdated(sharePrice);
}
```

```
function updateRewards(address[] memory _strategy) external{
    uint256 totalRewardsEarned;
    unchecked {
        for(uint i=0;i<_strategy.length;i++){
            if(!IStrategyManager(strategyManager).strategyExists
              (_strategy[i])) revert IncorrectStrategy(_strategy[i]);

@>          uint256 rewardsEarned = IStrategy(_strategy[i]).getRewards(address
  (this), tokenBalance);
            if(rewardsEarned>0){
                totalRewardsEarned+=rewardsEarned;
                IStrategyManager.StrategyStruct memory change;
                change.rewardsEarned=rewardsEarned;
                IStrategyManager(strategyManager).updateStrategy
                  (_strategy[i], change);
            }
        }
    }
    _updateSharePrice(totalRewardsEarned);
    emit RewardsCollected(totalRewardsEarned);
}
```

To illustrate the issue, consider the following scenario: 0. Alice deposits 1 stETH into the `DepositETH` contract, minting 1 ETH shares.

1. The protocol holds 1 stETH worth 1 ETH at the time of deposit.
2. At `t1`, rewards of 0.1 ETH accrue, resulting in the total balance becoming 1.1 ETH.
3. `DepositETH.updateRewards()` records +0.1 ETH as `rewardEarned` for the `LidoStrategy` and updates `rewardsClaimed` for `DepositETH`.
4. A Lido slashing event occurs, reducing the stETH balance to 1.05 ETH. (-0.05 ETH)
5. `DepositETH.updateRewards()` does not register any update because the current balance: 1.05 ETH is less than the previous balance: 1.1 ETH.

```
// LidoStrategy.sol
function getRewards(
  addressreceiver,
  IStrategyManager.StrategyStructmemory_strategyBalance
) public override view returns(uint256

        uint256 balance_strategy = _strategyBalance.valueDeposited +_strategyBalance.
   if (IERC20(TOKEN_ADDRESS).balanceOf
   //(receiver)>balance_strategy){ //@audit check 1.05 ETH > 1.10 ETH
        return IERC20(TOKEN_ADDRESS).balanceOf(receiver) - balance_strategy;
    }
    else{
@>      return 0; //@audit false: execute the else case
    }
}
```

6. The protocol continues to account for the original 1.1 ETH in the share price for any new deposits and incoming withdrawals.
7. Alice redeems her 1 ETH shares, Alice's withdrawal request is recorded 1.1 ETH amount to be withdrawn, while the actual stETH balance is 1.05 ETH.

Recommendations

Consider updating the share price calculation to account for Lido slashing events by reflecting any reduction in the stETH balance.

[M-04] LayerZero fee refunds misdirected to deposit contracts

Severity

Impact: Medium

Likelihood: Medium

Description

The LayerZero fee, which is intended to cover the cross-chain messaging cost for deposits and withdrawals, is not refunded to the sender who initiates the transaction through the Ethereum mainnet or Berachain.

Instead, this fee is directed to the deposit contract itself (`DepositETH`, `DepositUSD`, `DepositETHBera`, and `DepositUSDBera`), despite the fact that the sender is the one incurring the cost of the cross-chain messaging.

This can result in an unfair burden on the sender, who bears the full cost of initiating the cross-chain transaction without receiving any compensation for the excess fee.

```
@> function sendMessage(
  bytesmemory_data,
  uint32_destId,
  uint256_lzFee
) external override payable onlyDeposit{
    if(!destIdAvailable(_destId)) revert NotWhitelisted(_destId);
    MessagingReceipt memory receipt = _lzSend(
        _destId,
        _data,
        optionsDestId[_destId],
        MessagingFee(_lzFee, 0),
@>        payable(msg.sender)
    );
    emit MessageSent(_destId,_data,receipt);
}
```

Using the `Messaging` contract as an example, the `sendMessage()` can only be called by `DepositETH` or `DepositUSD`. Consequently, the `msg.sender` passed as the refund address is either the `DepositETH` or `DepositUSD` contract, rather than the user who originally initiated the deposit.

To note that the `MessageBera.sendMessage()` that is used in `DepositETHBera` and `DepositUSDBera` contracts shares the same issues.

Recommendations

Consider passing the user address that initiated the action as the refund address to ensure the fee is properly refunded to the user.

```
- function sendMessage
- (bytes memory _data, uint32 _destId, uint256 _lzFee) external override payable onlyD
+ function sendMessage
+ (bytes memory _data, uint32 _destId, uint256 _lzFee, address refundAddress) external
    if(!destIdAvailable(_destId)) revert NotWhitelisted(_destId);
    MessagingReceipt memory receipt = _lzSend(
        _destId,
        _data,
        optionsDestId[_destId],
        MessagingFee(_lzFee, 0),
-        payable(msg.sender)
+       payable(refundAddress)
    );
    emit MessageSent(_destId,_data,receipt);
}
```

[M-05] Incorrect `TOKEN_ADDRESS` and `DSR_DEPOSIT_ADDRESS`

Severity

Impact: Low

Likelihood: High

Description

The `DSRStrategy` declares incorrect contract addresses for `TOKEN_ADDRESS` and `DSR_DEPOSIT_ADDRESS`, which are intended to represent the `sDAI` contract for depositing `DAI` tokens and earning yield.

```
//mainnet
address
   public constant TOKEN_ADDRESS=0x3F1c547b21f65e10480dE3ad8E19fAAC46C95034;

address
   public constant DSR_DEPOSIT_ADDRESS=0x3F1c547b21f65e10480dE3ad8E19fAAC46C95034;
```

The `DSRStrategy` is primarily used as the implementation strategy for the DepositUSD contract on the Ethereum mainnet. However, the current addresses point to an account with no code.

This will prevent all operations and interactions with the intended address.

Recommendations

Update the `TOKEN_ADDRESS` and `DSR_DEPOSIT_ADDRESS` to the correct address: `sDAI: 0x83F20F44975D03b1b09e64809B757c47f942BEeA` (ETH Mainnet).

[M-06] Wrong fee estimate in `getlzFee()` function

Severity

Impact: Low

Likelihood: High

Description

Function `getlzFee()` is supposed to calculate the Layerzero bridge message fee inside deposit contracts. The issue is that the code uses the wrong `dataSend` value to estimate the bridge fee:

```
bytes memory dataSend = abi.encode(1, _receiver,nETHShares);
```

The correct data sent through Layerzero has 4 variable and the estimated fee will be lower which can cause the bridge transaction to fail.

```
bytes memory data = abi.encode(ASSET_ID,1, _receiver,nETHShares);
```

This happens in all the deposit contracts.

Recommendations

When estimating the fee use `ASSET_ID` too.

8.4. Low Findings

[L-01] Withdrawal replay attack

The `executeWithdrawal` function does not check if the withdrawal has already been executed. While it updates `withdrawalExecutionTime`, it never validates this timestamp. Therefore, the same withdrawal hash can be reused multiple times to drain funds.

Withdrawal replay attack can happen and it allows malicious or compromised strategy executors to drain the protocol's funds by reusing the same withdrawal hash multiple times.

It's recommended to add a check to ensure that the withdrawal has not been executed before:

```
function executeWithdrawal(address _receiver, bytes32 _hash) external {
+    if
+ (withdrawals[_receiver][_hash].withdrawalExecutionTime > 0) revert WithdrawalAlready
    ...
    }
```

[L-02] Insufficient address validation in convertToAddress

The `convertToAddress` has the following issues:

- Missing validation for "0x" prefix
- No check for standard Ethereum address length (40 hex characters + "0x")
- Could accept shorter inputs and pad with zeros due to assembly operation

```
function convertToAddress(string memory _address)internal pure returns
      (address){
        bytes memory ss = bytes(_address);
        require(ss.length%2 == 0); // length must be even
        bytes memory r = new bytes(ss.length/2);
        for (uint i=0; i<ss.length/2; ++i) {
            r[i] = bytes1(fromHexChar(uint8(ss[2*i])) * 16 +
                        fromHexChar(uint8(ss[2*i+1])));
        }
        address tempAddress;
        assembly {
            tempAddress := div(mload(add(add
              (r, 0x20), 1)), 0x1000000000000000000000000)
        }
        return tempAddress;
    }
```

It's recommended to add a validation for the address length and format.

[L-03] The stETH received can be less

1. When users deposit ETH through the LidoStrategy contract, the contract assumes the returned stETH will equal the deposited ETH amount. However, it may not be true.

```
function deposit(uint256 _value) public returns(uint256){
        (
          boolsuccess,
          bytesmemoryreturndata
        ) = LIDO_DEPOSIT_ADDRESS.call{value:_value}(abi.encodeWithSignature("submit(ad
        ...
        return 100;
    }
```

First, in Lido's `_submit` function, the ETH amount is converted to shares which rounds down:

```
function _submit(address _referral) internal returns (uint256) {
        require(msg.value != 0, "ZERO_DEPOSIT");
        ...

        uint256 sharesAmount = getSharesByPooledEth(msg.value);

        _mintShares(msg.sender, sharesAmount);

        ...
    }
```

When checking stETH balance, another round of conversion occurs which can also round down.

```
function balanceOf(address _account) external view returns (uint256) {
        return getPooledEthByShares(_sharesOf(_account));
    }
```

2. The same rounding issue happens when stETH is transferred to the DepositETH contract, the transfer `_value` can be less than the receive amount.

```
function _depositERC(
    address_tokenAddress,
    stringmemory_receiver,
    uint256_value,
    uint32_destID,
    uint256_lzFee
  ) internal onlyWhitelisted(_tokenAddress
        ...
        IERC20(_tokenAddress).transferFrom(msg.sender, address(this), _value);
        ...
    }
```

It's recommended to:

- bookkeep in shares instead of stETH
- using wstETH instead of stETH
- calculate the actual stETH received by the difference between the balance before and after transfer. For reference: https://docs.lido.fi/guides/lido-tokens-integration-guide/#1-2-wei-corner-case

[L-04] `BERA` native gas token can stuck

The `withdraw()` in both `DepositETHBera` and `DepositUSDBera` contracts do not validate if `msg.value` exceeds the `_lzFee`, potentially leading to `BERA` native gas tokens being stuck in the contract.

```
function withdraw(
  uint32_destId,
  stringmemory_receiver,
  uint256_amount,
  uint256_lzFee
) external payable{
    nETH(nETHToken).burnnETH(msg.sender, _amount);
    bytes memory sendMessageData = abi.encode(ASSET_ID,3,_receiver,_amount);
    IMessaging(messageApp).sendMessage{value:_lzFee}
      (sendMessageData,_destId,_lzFee);
}
```

Implement a check within the `withdraw()` function to ensure that `msg.value` is equal to `_lzFee`.

[L-05] Invalid address string can become valid and be used in the operation

The `deposit()` in the `DepositETH` and `DepositUSD` contracts, as well as the `withdraw()` in the `DepositETHBera` and `DepositUSDBera` contracts, receive a dynamic string as the receiver input address. This input is converted into an address using the `AddressLibrary` library on the destination chain.

However, the receiver address string can be passed as an invalid hex string and still be validated due to the conversion logic in `AddressLibrary.convertToAddress()`.

The issue arises because `convertToAddress()` defaults all invalid characters to 0. As a result, an address string like:

```
0x#A.98Eda*1Ce92ff4A4CC@7A4fFFb5A43EBC7-DC
```

It will valid and becomes

```
0x0A098Eda01Ce92ff4A4CC07A4fFFb5A43EBC70DC
```

While this issue may have a low impact because the resulting address is technically valid, the receiver parameter is used in event emissions and may be tracked off-chain. As a result, an invalid address could be emitted immediately alongside the input address.

[L-06] Precision loss in share-to-asset conversion

The `DSRStrategy` contract performs share-to-asset conversions manually by using the token price returned from `convertToAssets()` of `1e18` as a reference value for share-to-asset conversion:

```
function tokenPrice() external override view returns(uint256){
        return sDAI(TOKEN_ADDRESS).convertToAssets(1e18);
    }
```

However, the `sDAI` contract already provides a `convertToAssets()` designed for this purpose:

```
// SavingsDai.sol
function convertToAssets(uint256 shares) public view returns (uint256) {
    uint256 rho = pot.rho();
    uint256 chi = (block.timestamp > rho) ? _rpow(pot.dsr
      (), block.timestamp - rho) * pot.chi() / RAY : pot.chi();
    return shares * chi / RAY;
}
```

This approach can reduce the precision of share-to-asset conversions because `convertToAssets()` involves integer division, which effectively rounds down the result.

In the `DepositUSD._depositERC()`, the calculation of assetValue and the subsequent adjustment of strategy values may suffer from reduced precision:

```
function _depositERC(
  address_tokenAddress,
  stringmemory_receiver,
  uint256_value,
  uint32_destID,
  uint256_lzFee
) internal onlyWhitelisted(_tokenAddress
    if(msg.value!=_lzFee) revert IncorrectValue();
    IERC20(_tokenAddress).transferFrom(msg.sender, address(this), _value);
    (uint256 tokenValue, address _strategy) = IStrategyManager
      (strategyManager).fetchStrategy(_tokenAddress);
@>    uint256 assetValue = (_value*tokenValue)/BASE_POINT;
---
}
```

In the `DSRStrategy.getRewards()`, the calculation of `assetBalance` also relies on `convertToAssets()`, which may lead to discrepancies due to rounding:

```
function getRewards(
  addressreceiver,
  IStrategyManager.StrategyStructmemory_strategyBalance
) public override view returns(uint256

        uint256 balance_strategy = _strategyBalance.valueDeposited +_strategyBalance.
@>  uint256 assetBalance = IERC20(TOKEN_ADDRESS).balanceOf(receiver)*sDAI
  (TOKEN_ADDRESS).convertToAssets(1e18);
---
}
```

Consider consistently using the provided `convertToAssets()` for all share-to-asset conversions.

[L-07] Uninitialized proxy share price

The `sharePrice` variable in the `DepositETHBera` and `DepositUSDBera` contracts is improperly initialized within the implementation contracts instead of the proxy storage. As a result, the sharePrice for proxies begins at zero.

Moreover, since there is no use case of the `sharePrice` from these contracts in this scope, we recommend considering properly initializing the `sharePrice` in the proxy storage.

```
contract DepositETHBera {
---
    uint256 public sharePrice = 1e18;  //@audit This only initializes
    // implementation storage
---
}
```

Consider initializing the share price in the proxy contract through a constant/immutable declaration or during the initialization steps.

[L-08] Fixed gas low-level operations pose the risk of transaction failure

The use of `.call()` with a manually specified gas limit in functions: `DepositETH.executeWithdrawal()`, `DepositUSD.executeWithdrawal()`, and `StrategyManager.addStrategy()` may cause failures during execution. This design poses risks of incompatibility with future network changes, such as gas cost adjustments or new opcode introductions.

```
function executeWithdrawal(address _receiver, bytes32 _hash) external {
    if(msg.sender!=strategyExecutor) revert IncorrectStrategyExecutor
      (msg.sender);
@>    (
  boolsuccess,
  bytesmemoryreturndata
) = _receiver.call{value:withdrawals[_receiver][_hash].amount,gas:10000}(""
---
}
```

```
function addStrategy(address _strategy) external onlyOwner{
    StrategyStruct memory strategyBalance;
    address _tokenAddress = IStrategy(_strategy).TOKEN_ADDRESS();
    if(strategies[_tokenAddress]!=address(0)){
        strategyBalance = strategyDeposits[strategies[_tokenAddress]];
    }
    strategies[_tokenAddress] = _strategy;
    strategyDeposits[_strategy] = strategyBalance;
@>  (bool success,bytes memory returndata)=depositL1.call{gas:50000}
  (abi.encodeWithSignature("whitelistToken(address)", _tokenAddress));
---
}
```

Consider using dynamic gas forwarding instead of fixed gas limits in external calls. Additionally, consider implementing the best practices pattern, checks-effects-interactions, when interacting with untrusted external parties.

[L-09] Function `executeWithdrawal()` doesn't follow CEI pattern

Function `executeWithdrawal()` will finalize the withdrawal requests and send the funds to the receiver. The issue is that the function doesn't follow the CEI pattern and it sets the withdraw request state after it calls the receiver so it would be possible to perform a reentrancy attack and finalize a withdrawal multiple times if the strategy executor address was comprised. Because of this issue even if correct checks have been added to the function to prevent double finalization of the withdrawals by performing reentrancy it would be possible to finalize a withdrawal request multiple times.

[L-10] Function `_updateSharePrice()` in L2 deposit contracts never get called from L1

There's a `_updateSharePrice()` function in L2 that updates the share price and it can be triggered by bridged messages from L1. The issue is that there's no code in L1 contracts to trigger this functionality and the share price in L2 deposit contracts will always show the wrong value. Currently, There's no usage for the share price in L2 but if it's not going to have any utility in L2 then it's better to remove this functionality otherwise the L1 should send the new price to trigger this function execution.

[L-11] Front-running vulnerability in reward distribution

The `updateRewards` function in both `DepositETH` and `DepositUSD` contracts distributes rewards by updating the global share price, making it vulnerable to front-running attacks. This can be exploited through the following sequence:

- An attacker monitors the mempool for `updateRewards` transactions
- Front-run the transaction by calling `deposit`
- Allow `updateRewards` to execute
- Back-run with the withdrawal transaction

```
function updateRewards(address[] memory _strategy) external{
        ...
        _updateSharePrice(totalRewardsEarned);
        emit RewardsCollected(totalRewardsEarned);
    }
```

This allows attackers to capture rewards without meaningfully participating in the protocol's intended operation.

The impact is legitimate long-term holders receive diluted rewards.

It's recommended to:

- Accrue rewards before depositing OR
- Consider implementing a time-based reward distribution mechanism to reward users based on the time they hold their shares OR
- Add a minimum lock period between deposit and withdrawal.

[L-12] Implementation contracts allow unauthorized Initialization

The `DepositETH`, `DepositUSD`, `DepositETHBera`, `DepositUSDBera`, and `StrategyManager` contracts do not disable initialization. This allows the ownership of the implementation being taken. Specifically, for `DepositETH`, this vulnerability can be exploited to invoke `selfdestruct` via the `DepositETH.executeStrategies()`.

```
function executeStrategies
  (StrategyExecution[] memory _executionData) external override {
    if(msg.sender!=strategyExecutor) revert IncorrectStrategyExecutor
      (msg.sender);
    unchecked{
        for(uint i=0;i<_executionData.length;i++){
            if(!IStrategyManager(strategyManager).strategyExists
              (_executionData[i].strategy)) revert IncorrectStrategy(_executionData[i]
@>          (
  boolsuccess,
  bytesmemoryreturndata
) = _executionData[i].strategy.delegatecall(_executionData[i].executionData
---
    emit StrategyExecutionCompleted(block.timestamp);
}
```

Although the Cancun upgrade prevents `selfdestruct` from deleting the contract code, it still allows sending all the native balances in the account to the specified target address.

As `DepositETH` includes functionality to accept direct native deposits, an attacker could exploit this flaw by initializing the implementation contract, invoking `selfdestruct`, and draining all funds locked in the implementation contract.

Apply the `_disableInitializers()` from Openzeppelin in the constructor to disable the initialize and reinitialize for the implementation contract

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

[L-13] Users who deposit `DAI` directly will lose their deposits

When users deposit DAI directly into the `DepositUSD` contract, their funds are routed to the `sDAI` strategy via `executeStrategies()`. However, the design does not issue shares or properly account for these deposits as rewards, leading to a loss of funds.

```
function executeStrategies
  (StrategyExecution[] memory _executionData) external override {
    if(msg.sender!=strategyExecutor) revert IncorrectStrategyExecutor
      (msg.sender);
    unchecked{
        for(uint i=0;i<_executionData.length;i++){
            if(!IStrategyManager(strategyManager).strategyExists
              (_executionData[i].strategy)) revert IncorrectStrategy(_executionData[i]
1>          (
  boolsuccess,
  bytesmemoryreturndata
) = _executionData[i].strategy.delegatecall(_executionData[i].executionData
            if(!success){
                if (returndata.length == 0) revert CallFailed();
                assembly {
                    revert(add(32, returndata), mload(returndata))
                }
            }
            uint256 _type=abi.decode(returndata,(uint256));
            if (_type==100){
                IStrategyManager.StrategyStruct memory change;
                change.valueDeposited = _executionData[i].value;
2>              IStrategyManager(strategyManager).updateStrategy
  (_executionData[i].strategy,change);
            }else if(_type==101){
---
}
```

The `DepositUSD.executeStrategies()` is designed to handle strategy executions, including deposits into sDAI (`1>`) and updating the strategyDeposits struct (`2>`). However, it lacks the logic to issue shares to users for their deposits, and the deposited funds are incorrectly handled in the rewards calculation.

```
function deposit(uint256 _value) public returns(uint256){
    sDAI(DSR_DEPOSIT_ADDRESS).deposit(_value,address(this));
    return 100;
}
```

```
function updateStrategy(
  address_strategy,
  StrategyStructmemory_changeBalane
) external override onlyDeposit{
    strategyDeposits[_strategy].valueDeposited += _changeBalane.valueDeposited;
    strategyDeposits[_strategy].valueWithdrawn += _changeBalane.valueWithdrawn;
    strategyDeposits[_strategy].rewardsEarned += _changeBalane.rewardsEarned;
    emit StrategyBalanceUpdated(_changeBalane);
}
```

In addition, the deposited `DAI` is not recorded as part of the rewards used to calculate the share price because of the `DSRStrategy.getRewards()` function shown below. The newly deposited balance is included in the `_strategyBalance.valueDeposited`, along with the `sDAI` shares balance at the time of the deposit.

As a result, when calculating the rewards for the `DSRStrategy`, the protocol does not account for this newly deposited DAI, leading to an inaccurate reward distribution within the protocol.

```
function getRewards(
  addressreceiver,
  IStrategyManager.StrategyStructmemory_strategyBalance
) public override view returns(uint256
@>
    uint256 balance_strategy = _strategyBalance.valueDeposited +_strategyBalance.rewar
@>  uint256 assetBalance = IERC20(TOKEN_ADDRESS).balanceOf(receiver)*sDAI
  (TOKEN_ADDRESS).convertToAssets(1e18);
    if (assetBalance>balance_strategy){
        return assetBalance - balance_strategy;
    }
    else{
        return 0;
    }
}
```

For example, assuming there is an initial deposit of 0 DAI (`stableDeposited`) and 0 rewards (`rewardsClaimed`), if a user deposits `1000 DAI`, the `_strategyBalance.valueDeposited` will be `1000 DAI`, and `X sDAI` shares will be issued equivalent to the `1000 DAI`.

After some time, rewards occur (`Y DAI`), and the share price will be recalculated. The reward that will be considered in the share price calculation will be:

```
// total balance used to calculate share price
totalUSDBalance = stableDeposited-stableWithdrawn+rewardsClaimed
= 0 - 0 + Y

// the reward that will be considered
rewardClaimed += (1000 DAI + Y DAI) - 1000 DAI: balance_strategy
= Y
```

As a result, the direct deposit of `DAI` will not be properly accounted for in the reward calculations, and users will be unable to claim their funds back because no shares are issued.

Implement a function to deposit `DAI` that correctly handles user deposits, ensures proper accounting of `DAI`, and mints shares for depositors accordingly.