Native Yield

Smart Contract Security Audit

Prepared by ShellBoxes

October 14th, 2024 - October 24th, 2024

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Nexus Network
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/Nexus-2023/NativeYield	c639ed6081b5989b96d98afc3fc883e7be722cca

Re-Audit

Repository	Commit Hash
https://github.com/Nexus-2023/NativeYield	39de477370e988f8ccac278ffd65d8f0f2870c79

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Nexus Network engaged ShellBoxes to conduct a security assessment on the Native Yield beginning on October 14th, 2024 and ending October 24th, 2024. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Nexus Network

Issuer	Nexus Network
Website	`https://www.nexusnetwork.live`
Type	Solidity Smart Contract
Documentation	Nexus Network Docs
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact				
	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low

Likelihood

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Native Yield implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 2 critical-severity, 1 high-severity, 4 medium-severity, 2 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Transfer to Zero Address in withdrawCollectedFee Function	CRITICAL	Fixed
SHB.2. Unimplemented Withdrawal Functionality in DepositL1 Contract	CRITICAL	Fixed
SHB.3. Potential Loss of Deposited Token Records in addStrategy Function	HIGH	Fixed
SHB.4. Lack of Result Verification After _lzSend Call in sendMessage Function	MEDIUM	Fixed
SHB.5. Front-Run Vulnerability in Contract Initialization	MEDIUM	Mitigated
SHB.6. Lack of Two-Step Ownership Transfer Mechanism	MEDIUM	Acknowledged

SHB.7. Insecure Token Transfer Implementation	MEDIUM	Fixed
SHB.8. Missing Value Verification in changeFee Function	LOW	Acknowledged
SHB.9. Missing Address Verification	LOW	Acknowledged

3 Finding Details

SHB.1 Transfer to Zero Address in withdrawCollectedFee Function

- Severity: CRITICAL
- Status: Fixed
- Likelihood: 3
- Impact: 3

Description:

In the NexusFee contract, the withdrawCollectedFee function contains a critical issue where it attempts to transfer Ether to the zero address (address(0)) if the _tokenAddress is set to the zero address. This approach is fundamentally flawed, as transferring Ether to the zero address will result in the loss of funds and effectively make them unrecoverable. The contract should instead transfer the Ether to the specified _receiver address, as intended. This oversight not only jeopardizes user funds but also undermines the contract's integrity and reliability.

Files Affected:

SHB.1.1: NexusFee.sol

```
59      function withdrawCollectedFee(address _tokenAddress,uint256 _value,
            ↪ address _receiver) external onlyOwner{
60          if (_tokenAddress==address(0)){
61              (bool success,) = _tokenAddress.call{value:_value}("");
62              if (!success){
63                  revert CallFailed();
64              }
65          }
```

Recommendation:

To address this issue, it is recommended to ensure that Ether is always sent to the _receiver address when _tokenAddress is zero.

Updates

The team has addressed the issue in the withdrawCollectedFee function by implementing a transfer to the _receiver address instead of the _tokenAddress. They have also added a zero-address check to prevent transfers to an unintended address. The updated code uses: (bool success,) = _receiver.callvalue: _value("");. This change ensures that the transfer correctly targets the _receiver and avoids potential security risks associated with zero-address transfers.

SHB.2 Unimplemented Withdrawal Functionality in DepositL1 Contract

- Severity : CRITICAL
- Status : Fixed
- Likelihood : 3
- Impact : 3

Description:

In the DepositL1 contract, the messageReceivedL2 function is designed to handle messages received from Layer 2, allowing for cross-chain transactions via the LayerZero protocol. However, the actual withdrawal of funds is not fully implemented, as the _withdrawFunds function is marked as a placeholder and does not contain any logic. This lack of implementation means that while the contract can receive messages and identify valid transaction types, there is no mechanism in place to facilitate the transfer of tokens or assets to users. Consequently, this could lead to significant functionality gaps, as users would be unable to withdraw their funds, resulting in potential loss of user trust and usability issues.

Files Affected:

SHB.2.1: DepositL1.sol

```
191     function messageReceivedL2(uint256 _id, address _receiver, uint256
            ↪ _value) external override onlyMessageApp {
192         if(_id==3){
193             uint256 _valueETHWithdraw = _value*BASE_POINT/sharePrice;
194             _withdrawFunds(_receiver, _valueETHWithdraw);
195         }
196         else {
197             revert IncorrectTypeID(_id,msg.sender);
198         }
199     }
200
201     // TODO to implement later on
202     function _withdrawFunds(address _reciever, uint256 _value) internal
            ↪ {
203     }
```

Recommendation:

Consider implementing the _withdrawFunds function to enable proper withdrawal behavior. The function should include logic to handle token transfers securely, ensuring that users can receive their funds as intended.

Updates

The team has implemented the previously empty _withdrawFunds function within the DepositL1 contract. This function now performs the necessary steps for recording a withdrawal, including generating a unique withdrawalHash and storing the withdrawal details in the withdrawals mapping. The function also emits a WithdrawalRequestCreated event to log the withdrawal request.

SHB.3 Potential Loss of Deposited Token Records in addStrategy Function

- Severity : HIGH
- Status : Fixed
- Likelihood : 2
- Impact : 3

Description:

In the StrategyManager contract, the addStrategy function enables the contract owner to add a new strategy by mapping the associated token address to the corresponding strategy contract. However, if there are existing deposits associated with a previous strategy for the _tokenAddress in the strategyDeposits mapping, these funds will not be transferred or accounted for when a new strategy is added. As a result, the new strategy will start with a zero balance, leading to potential confusion regarding the actual available funds managed by each strategy.
This oversight can impact the accuracy of user balances and performance tracking, creating a lack of transparency and potentially undermining user trust in the contract's functionality.

Files Affected:

SHB.3.1: StrategyManager.sol

```
21      mapping(address=>address) public strategies;
22
23      mapping(address=>StrategyStruct) public strategyDeposits;
```

SHB.3.2: StrategyManager.sol

```
54      function addStrategy(address _strategy) external onlyOwner{
55          address _tokenAddress = IStrategy(_strategy).TOKEN_ADDRESS();
56          strategies[_tokenAddress] = _strategy;
```

```
57          (bool success,bytes memory returndata)=depositL1.call{gas:50000}(
                ↪ abi.encodeWithSignature("whitelistToken(address)",
                ↪ _tokenAddress));
58          if(!success){
59              if (returndata.length == 0) revert CallFailed();
60              assembly {
61                  revert(add(32, returndata), mload(returndata))
62              }
63          }
64          emit StrategyAdded(_tokenAddress,_strategy);
65      }
```

Recommendation:

To address this issue, it is recommended to implement logic within the addStrategy function to ensure that existing deposits from previous strategies are properly managed.

Updates

The team has resolved the issue in the addStrategy function by ensuring that the balance from the previous strategy is recorded and associated with the new strategy for the same token. Now, when a new strategy is added, the function retrieves and records the previous strategy's balance. This update prevents any loss of balance information during strategy updates, maintaining accurate records of deposited tokens and ensuring that the new strategy begins with the correct starting balance.

SHB.4 Lack of Result Verification After _lzSend Call in sendMessage Function

- Severity: MEDIUM
- Likelihood: 2
- Status: Fixed
- Impact: 2

Description:

In the Messaging/MessagingBera contracts, the sendMessage function allows users to send messages across the LayerZero protocol. However, this implementation lacks a check for the result of the _lzSend function call, which is responsible for executing the message transmission. The _lzSend function returns a MessagingReceipt structure that contains important information regarding the message sent, such as a unique identifier and the associated fees. Without verifying the result of this function call, the contract may proceed under the assumption that the message was successfully sent, potentially leading to unrecognized failures or message delivery issues.

Files Affected:

SHB.4.1: Messaging.sol

```
47      function sendMessage(bytes memory _data, uint32 _destId, uint256
            ↪ _lzFee) external override payable onlyDeposit{
48          if(!destIdAvailable(_destId)) revert NotWhitelisted(_destId);
49          _lzSend(
50              _destId,
51              _data,
52              optionsDestId[_destId],
53              MessagingFee(_lzFee, 0),
54              payable(msg.sender)
55          );
56          emit MessageSent(_destId,_data);
```

```
57     }
```

SHB.4.2: MessageBera.sol

```
36     function sendMessage(bytes memory _data, uint32 _destId, uint256
           ↪ _lzFee) external override payable onlyDeposit{
37         if(!destIdAvailable(_destId)) revert NotWhitelisted(_destId);
38         _lzSend(
39             _destId,
40             _data,
41             optionsDestId[_destId],
42             // Fee in native gas and ZRO token.
43             MessagingFee(_lzFee, 0),
44             // Refund address in case of failed source message.
45             payable(msg.sender)
46         );
47         emit MessageSent(_destId,_data);
48     }
```

Recommendation:

It is recommended to implement result verification for the _lzSend function within the sendMessage function. This can be done by capturing the returned MessagingReceipt and checking its properties to ensure that the message was sent successfully.

Updates

The team has resolved this issue by adding the result of the _lzSend function (the MessagingReceipt) to the emitted MessageSent event. The updated implementation ensures that when the sendMessage function is called, relevant details of the message transmission, including the guid, are logged for better tracking and verification.

SHB.5 Front-Run Vulnerability in Contract Initialization

- Severity : MEDIUM
- Status : Mitigated
- Likelihood : 1
- Impact : 3

Description:

The initialize function in the project contracts presents a front-run attack risk. As this function is designed to be called only once during the contract's initialization, the lack of additional security measures creates a window of vulnerability. An attacker could exploit this by front-running the initialization transaction, allowing them to claim ownership and manipulate the contract's state. This vulnerability can lead to unauthorized control over the contract, posing significant risks to both the project and its users.

Files Affected:

SHB.5.1: DepositL1.sol

```
43      function initialize() public initilizeOnce {
44          _ownableInit(msg.sender);
45          sharePrice = 1e18;
46      }
```

SHB.5.2: StrategyManager.sol

```
30      function initialize() public initilizeOnce {
31          _ownableInit(msg.sender);
32      }
```

SHB.5.3: DepositBera.sol

```
24      function initialize() public initilizeOnce {
25          _ownableInit(msg.sender);
26      }
```

Recommendation:

To mitigate the risk, it is recommended to implement a deployment script that utilizes a factory pattern. This approach will allow for the atomic deployment and initialization of contracts, ensuring that both processes occur within a single transaction.

Updates

The team has mitigated the risk of front-run vulnerability in contract initialization by ensuring that the initialization function is called by the proxy during the setup process in the constructor. This guarantees that the function will only execute once in the context of the proxy contract, thereby reducing the potential for front-running attacks.

SHB.6 Lack of Two-Step Ownership Transfer Mechanism

- Severity : MEDIUM
- Status : Acknowledged
- Likelihood : 1
- Impact : 3

Description:

In the NexusOwnable contract, the transferOwnership function allows the current owner to transfer ownership directly to a specified address. While there is a safeguard in place to prevent the transfer to a zero address, the function does not implement a two-step ownership transfer process. This absence increases the risk of mistakenly transferring ownership to an unintended or incorrect address, which could result in a permanent loss of control over the contract. Such scenarios may expose the contract to unauthorized access or operational failures.

Files Affected:

SHB.6.1: NexusOwnable.sol

```
40      function transferOwnership(address newOwner) external onlyOwner{
```

```
41          if (newOwner == address(0)) revert IncorrectAddress();
42
43          emit OwnerChanged(owner, newOwner);
44          owner = newOwner;
45      }
```

Recommendation:

It is recommended to modify the transferOwnership function to set the new owner as pending, requiring the new owner to call an acceptOwnership function to finalize the transfer. This two-step process would add an additional layer of security, allowing the current owner to verify the intended recipient before the ownership is irrevocably changed, thus reducing the risk of transferring ownership to an incorrect address.

Updates

The team has acknowledged the issue, stating that a timelock will be integrated in the future.

SHB.7 Insecure Token Transfer Implementation

- Severity : MEDIUM
- Likelihood : 2
- Status : Fixed
- Impact : 2

Description:

In the NexusFee contract, the withdrawCollectedFee function uses the IERC20 interface's transfer function to send tokens from the contract to a specified _receiver address. However, this approach does not verify if the transfer operation succeeded, as the transfer function returns a boolean value indicating success.

This is problematic since some tokens (e.g., USDT, BNB) may not behave as expected, either by returning false or by not returning any value at all, leading to compatibility issues and potential transfer failures without detection. This can result in funds being locked or incorrectly transferred, exposing the contract to unexpected behavior.

Files Affected:

SHB.7.1: NexusFee.sol

```
66          else{
67              IERC20(_tokenAddress).transfer(_receiver, _value);
68          }
```

Recommendation:

It is recommended to use OpenZeppelin's SafeERC20 library, specifically the safeTransfer function, which handles such cases by ensuring that the token transfer either succeeds or reverts if it fails. This approach adds an extra layer of safety and compatibility for tokens that may not conform to the standard transfer behavior.

Updates

The team has resolved the issue by incorporating the safeTransfer function from the SafeERC20 library into the withdrawCollectedFee function.

SHB.8 Missing Value Verification in changeFee Function

- Severity : LOW
- Status : Acknowledged
- Likelihood : 1
- Impact : 2

Description:

In the NexusFee contract, the changeFee function lacks validation checks to ensure that the provided fee values (_onboardingFee, _withdrawalFee, and _maintenanceFee) are within acceptable ranges. Currently, these values are assigned directly without verifying if they are greater than zero and within a maximum allowable limit. This could lead to unintended behavior, such as setting fees to zero or excessively high values, which may disrupt the contract's intended fee structure or allow for potential manipulation.

Files Affected:

SHB.8.1: NexusFee.sol

```
42      function changeFee(uint256 _onboardingFee,uint256 _withdrawalFee,
            ↪ uint256 _maintenanceFee) external onlyOwner(){
43          onboardingFee=_onboardingFee;
44          withdrawalFee=_withdrawalFee;
45          maintenanceFee=_maintenanceFee;
46          emit NexusFeeChanged( _onboardingFee, _withdrawalFee,
                ↪ _maintenanceFee);
47      }
```

Recommendation:

It is recommended to implement validation checks within the changeFee function to ensure that all fee values are greater than zero and do not exceed a specified maximum threshold. This will help maintain the integrity of the fee structure and prevent misuse or unintentional assignment of invalid values.

Updates

The team has acknowledged the issue, stating that only the owner can modify the fees using this function. Additionally, it has been noted that ownership of this functionality will eventually transition to a timelock mechanism in the future.

SHB.9 Missing Address Verification

- Severity : LOW
- Status : Acknowledged
- Likelihood : 1
- Impact : 2

Description:

In several setter, initializer functions, and constructors across the main contracts, there is an absence of validation checks for the address parameters being provided. These address parameters are directly assigned without verification, which poses a security risk, such as the potential assignment of invalid or zero addresses. This lack of validation can compromise the functionality and reliability of the contract, leading to vulnerabilities that may be exploited maliciously or result in unintended behavior.

Files Affected:

SHB.9.1: DepositL1.sol

```
59      function setMessagingApp(address _messageApp) external onlyOwner{
60          messageApp=_messageApp;
61          emit MessagingContractAddressAdded(_messageApp);
62      }
63
64      function setStrategyAddress(address _strategyManager) external
            ↪ onlyOwner{
65          strategyManager=_strategyManager;
66          emit StrategyContractAddressAdded(_strategyManager);
```

```
67      }
68
69      function setStrategyExecutor(address _executor) external onlyOwner{
70          strategyExecutor = _executor;
71          emit StrategyExecutorAddressAdded(_executor);
72      }
```

SHB.9.2: Messaging.sol

```
33      constructor(address _endpoint,address _deposit) OApp(_endpoint, msg.
            ↪ sender) Ownable(msg.sender) {
34          depositL1 = _deposit;
35      }
```

SHB.9.3: StrategyManager.sol

```
49      function setDeposit(address _deposit) external onlyOwner{
50          depositL1=_deposit;
51          emit DepositAddressSet(_deposit);
52      }
```

SHB.9.4: DepositBera.sol

```
35      function addnETHAddress(address _nETH) external onlyOwner{
36          nETH=_nETH;
37      }
38
39      function addMessageApp(address _messageAddress) external onlyOwner{
40          messageApp=_messageAddress;
41      }
```

SHB.9.5: MessageBera.sol

```
22      constructor(address _endpoint,address _deposit) OApp(_endpoint, msg.
            ↪ sender) Ownable(msg.sender) {
23          depositBera = _deposit;
24      }
```

SHB.9.6: nETH.sol

```
17      constructor(address _deposit) ERC20("Native Nexus ETH","nETH"){
18          depositAddress = _deposit;
19      }
```

Recommendation:

It is recommended to implement validation checks for all address parameters within the relevant functions. Ensure that these checks verify that the addresses are valid and non-zero before they are assigned, mitigating the risk of assigning invalid addresses.

Updates

The team has acknowledged the issue, stating that the functions for setting and updating addresses are protected by the onlyOwner modifier. As a result, the authorized owner is responsible for making these changes, which mitigates the risk associated with this oversight.

4 Best Practices

BP.1 Avoid Redundant Variable Initialization to Default Values

Description:

In the NexusFee contract, the state variables onboardingFee, withdrawalFee, and maintenanceFee are explicitly initialized to zero. However, since zero is the default value for uint256 in Solidity, this explicit initialization is unnecessary and can lead to redundant code. It is advisable to remove the explicit initializations for these variables. This practice simplifies the code, making it cleaner and more readable while still retaining the intended functionality. By avoiding unnecessary initializations, the code becomes clearer and reduces potential confusion regarding the purpose of the declarations.

Files Affected:

BP.1.1: NexusFee.sol

```
20      uint256 public onboardingFee=0;
21      uint256 public withdrawalFee=0;
22      uint256 public maintenanceFee=0;
```

Status - Fixed

BP.2 Consistent Naming Convention for Contract and File Names

Description:

In the Ownable contract, the contract name should ideally match the file name, which in this case is NexusOwnable.sol. Maintaining consistent naming conventions for contract and file names enhances code organization and improves readability, making it easier for developers to locate and understand the contract's purpose.

It is advisable to rename the contract to NexusOwnable to match the file name. This practice not only adheres to standard conventions but also promotes clarity within the codebase, helping to avoid confusion, especially in larger projects where multiple contracts may be involved. The same remark applies to the NexusProxy file, where the contract name should also align with the file name for consistency and clarity.

Files Affected:

BP.2.1: NexusOwnable.sol

```
9  contract Ownable {
```

BP.2.2: NexusProxy.sol

```
10 contract Proxy{
```

Status - Fixed

BP.3 Combine Duplicate Logic for Strategy Execution Types

Description:

In the executeStrategies function, there are two separate conditions handling different execution types (100 and 101) that perform similar operations for updating the strategy state. Both conditions construct a StrategyStruct object and call updateStrategy with the corresponding value, leading to redundancy and increased maintenance complexity. To enhance code readability and maintainability, it is advisable to consolidate these duplicate logic branches into a single code block. This can be achieved by using a unified approach to determine the type and updating the StrategyStruct accordingly, thereby simplifying the code and reducing the likelihood of introducing errors in future modifications.

BP.3.1: DepositL1.sol

```
168 IStrategyManager.StrategyStruct memory change;
169 if (\_type == 100) {
170 change.ethDeposited = \_executionData[i].value;
```

```
171 } else if (\_type == 101) {
172 change.ethWithdrawn = \_executionData[i].value;
173 } else {
174 revert IncorrectValue();
175 }
176 // Update strategy balance only once
177 IStrategyManager(strategyManager).updateStrategy(\_executionData[i].
      ↪ strategy, change);
```

Files Affected:

BP.3.2: DepositL1.sol

```
167                 uint256 _type=abi.decode(returndata,(uint256));
168                 if (_type==100){
169                     IStrategyManager.StrategyStruct memory change;
170                     change.ethDeposited = _executionData[i].value;
171                     IStrategyManager(strategyManager).updateStrategy(
                           ↪ _executionData[i].strategy,change);
172                 }else if(_type==101){
173                     IStrategyManager.StrategyStruct memory change;
174                     change.ethWithdrawn = _executionData[i].value;
175                     IStrategyManager(strategyManager).updateStrategy(
                           ↪ _executionData[i].strategy,change);
176                 }
177                 else{
178                     revert IncorrectValue();
179                 }
```

Status - Acknowledged

BP.4 Public Functions Can Be Declared as External

Description:

When defining functions in Solidity, consider using the external visibility modifier instead of public where applicable to reduce gas costs. External functions are more restricted, as they cannot be called internally and can only be called by other contracts and externally-owned accounts. This restriction allows the compiler to optimize the function's bytecode, leading to lower gas costs. Review the project contracts to identify public functions that do not need to be called internally and change their visibility to external to benefit from potential gas savings. Note : The provided code snippet is just one example of a public function. However, there are multiple instances throughout the codebase where public functions could be declared as external, particularly the updateProxy functions.

Files Affected:

BP.4.1: DepositL1.sol

```
48      function updateProxy(
49          address _newImplemetation
50      ) public onlyOwner {
51          if (_newImplemetation == address(0)) revert IncorrectAddress();
52          updateCodeAddress(_newImplemetation);
53      }
```

Status - Acknowledged

5 Tests

Foundry Tests:

- ✓ test_adminActions() (gas: 29885)
- ✓ test_depositAndMint() (gas: 900934)
- ✓ test_withdrawal() (gas: 1583042)
- ✓ testDeposit() (gas: 113732)
- ✓ testRewards() (gas: 149652)
- ✓ testSupervisorAccess() (gas: 77648)
- ✓ testTokenMintAndBurn() (gas: 95727)
- ✓ testWithdrawal() (gas: 134459)
- ✓ test_ERCDeposit() (gas: 1909466)
- ✓ test_ETHDeposit() (gas: 347259)
- ✓ test_adminActions() (gas: 30050)
- ✓ test_sendMessage() (gas: 428759)
- ✓ test_whitelistDestID() (gas: 14625)

Hardhat Tests:

- ✓ set deposit address in strategymanager (704ms)
- ✓ set strategymanager address in deposit (362ms)
- ✓ set message address in deposit (351ms)
- ✓ should set executor address (725ms)
- ✓ should whitelist and set the option for destID (752ms)
- ✓ should whitelist strategy (1214ms)

- ✓ should give weth value
- ✓ should take ETH deposit (24197ms)
- ✓ should not take random Token deposit (542ms)
- ✓ should take Whitelist Token deposit (4327ms)
- ✓ should not take random dest id deposit (404ms)
- ✓ should execute single strategy (2606ms)
- ✓ should execute multiple strategies (64ms)
- ✓ should throw error is strategy not available (425ms)
- ✓ reward changes sharePrice (1075ms)

Coverage:

→ Foundry Tests (13 passed,0 failed)

- Statements Coverage : 45.90%
- Branches Coverage : 19.35%
- Functions Coverage : 62.77%
- Lines Coverage : 49.20%

→ Hardhat Tests (15 passed,0 failed)

- Statements Coverage : 54.74%
- Branches Coverage : 29.89%
- Functions Coverage : 52.13%
- Lines Coverage : 55.64%

Conclusion:

The project offers a testing mechanism to improve the correctness of smart contracts; nonetheless, the test coverage percentage is low; it must be increased to cover all functionalities and test cases in order to guarantee the integrity of the code and the functionality of the protocol.

6 Conclusion

In this audit, we examined the design and implementation of Native Yield contract and discovered several issues of varying severity. Nexus Network team addressed 5 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised Nexus Network Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
src/DepositL1.sol	fbffcdb7f7e557ebd1e1708db9aad3a9
src/Messaging.sol	1343a673287b7a74dc0969077f542cdf
src/NexusFee.sol	898b0d45e2509f0f8789f81c2201a023
src/StrategyManager.sol	a6c6392ccd2c469400e04a9db9f8f675
src/utils/Helpers.sol	dcc095c07c6a6e7ac3ccc88fbdf9c16f
src/utils/NexusOwnable.sol	efbbe69e301a36b6dd33c020802bc103
src/utils/NexusProxy.sol	354f7a98f31da9dfcafc4ff7cdd1b592
src/utils/UUPSUpgreadable.sol	b399b0cce4018c9ec7f8731860698b0b
src/strategies/LidoStrategy.sol	7460bc26646d544042b87bb82e2bbc74
src/strategies/WETHStrategy.sol	2492034e69bb5b2330fb6d4476fefd4f
src/berachain/DepositBera.sol	5982cdc29e919e4f18b1c67e76360cb5
src/berachain/MessageBera.sol	d58773161256de041a81e000018615f8
src/berachain/nETH.sol	ced04ea061f99736e9a9356258fba710

7.2 Re-Audit

Files	MD5 Hash
src/DepositL1.sol	0ec29b29369ae859ccaf5ac2115542eb

src/Messaging.sol	4f3d36f380665e2560d9b0c36f27d800
src/NexusFee.sol	9666600c50d357a1f2f407e6fb71bdaa
src/StrategyManager.sol	9001770d707755becc752c56a09e0ea4
src/utils/Helpers.sol	3dc2a4e9ff89c70da31adcc9024ac0a5
src/utils/NexusOwnable.sol	fe1741ea60df14d6a9cbcf7110ef90f0
src/utils/NexusProxy.sol	3e14e6f12d216d55419ec560d1fb7d0f
src/utils/UUPSUpgreadable.sol	b399b0cce4018c9ec7f8731860698b0b
src/strategies/LidoStrategy.sol	7460bc26646d544042b87bb82e2bbc74
src/strategies/WETHStrategy.sol	2492034e69bb5b2330fb6d4476fefd4f
src/berachain/DepositBera.sol	7e727e3db79d68f9af7fefef40256bb4
src/berachain/MessageBera.sol	a2ec8c0ecf800d0c1b0b17348c394c01
src/berachain/nETH.sol	ced04ea061f99736e9a9356258fba710

8 Disclaimer

Shellboxes reports should not be construed as "endorsements" or "disapprovals" of particular teams or projects. These reports do not reflect the economics or value of any "product" or "asset" produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

For a Contract Audit, contact us at contact@shellboxes.com