Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.01.26, the SlowMist security team received the DeSyn Protocol team's security audit application for DeSyn Phase6, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

Desyn is a web3 asset management platform that provides a decentralized asset management infrastructure for everyone around the world.

This time it is to audit its `batch-buy` and `desyn-modules-forge` modules. The `batch-buy` module uses the ERC4626 standard to provide users with a collective subscription service for ETFs. The `desyn-modules-forge` module implements the function of investing ETF funds into the yearn protocol to earn profits.

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Potential risk that the `getUnderlyingAsset` function can be manipulated	Others	Information	Acknowledged
N2	Potential risk of admin role acting maliciously	Authority Control Vulnerability Audit	Medium	Acknowledged
N3	Potential denial of service risk for batchDeposit operations	Denial of Service Vulnerability	Medium	Acknowledged
N4	Inconsistency issue between issue amount and contract asset balance	Design Logic Audit	Low	Acknowledged
N5	Incorrect `total` computation in `autoHelperRedeem`	Design Logic Audit	Critical	Fixed

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/Meta-DesynLab/desyn-modules-forge/tree/main/src

commit: 211120d631d1fb325c606009a8361a88671e6273

https://github.com/Meta-DesynLab/batch-buy/tree/main/contracts

commit: dc808f1699fc245392513239e6db79b30e50d782

Fixed Version:

https://github.com/Meta-DesynLab/batch-buy/tree/audit

commit: 9b2d795980af5f5561faba155c60b9f1fb0dbe1f

Audit Scope:

- desyn-modules-forge/src/*.sol
- batch-buy/contracts/ERC4626/*.sol
- batch-buy/contracts/interface/*.sol
- batch-buy/contracts/ERC4626Vault.sol
- batch-buy/contracts/ERC4626Factory.sol

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

InterestEnhanced			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
getAssetAmount	Public	-	-
getCurveLP	Public	-	-
getVaultShare	Public	-	-
getUnderlyingLP	Public	-	-
getUnderlyingAsset	External	-	-
_deposit	Internal	Can Modify State	-
_stake	Internal	Can Modify State	-
_unstake	Internal	Can Modify State	-
_redeem	Internal	Can Modify State	-
_updatePosition	Internal	Can Modify State	-
_checkTx	Internal	-	-

InterestEnhanced			
startToEarn	External	Can Modify State	-
clearPosition	External	Can Modify State	-
setFactory	External	Can Modify State	onlyOwner

Invoke			
Function Name	Visibility	Mutability	Modifiers
invokeApprove	Internal	Can Modify State	-
invokeTransfer	Internal	Can Modify State	-
strictInvokeTransfer	Internal	Can Modify State	-
invokeUnwrapWETH	Internal	Can Modify State	-
invokeWrapWETH	Internal	Can Modify State	-
invokeMint	Internal	Can Modify State	-
invokeUnbind	Internal	Can Modify State	-
invokeRebind	Internal	Can Modify State	-

YearnCall			
Function Name	Visibility	Mutability	Modifiers
getDepositCalldata	Internal	-	-
invokeDeposit	Internal	Can Modify State	-
getStakeCalldata	Internal	-	-
invokeStake	Internal	Can Modify State	-
getUnstakeCalldata	Internal	-	-
invokeUnstake	Internal	Can Modify State	-

YearnCall			
getRedeemCalldata	Internal	-	-
invokeRedeem	Internal	Can Modify State	-

ERC4626			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
_updateAsset	Internal	Can Modify State	-
_tryGetAssetDecimals	Private	-	-
decimals	Public	-	-
asset	Public	-	-
totalAssets	Public	-	-
convertToShares	Public	-	-
convertToAssets	Public	-	-
maxDeposit	Public	-	-
maxMint	Public	-	-
maxWithdraw	Public	-	-
maxRedeem	Public	-	-
previewDeposit	Public	-	-
previewRedeem	Public	-	-
deposit	Public	Can Modify State	-
redeem	Public	Can Modify State	-
_redeemForHelper	Internal	Can Modify State	-
_convertToShares	Internal	-	-

ERC4626			
_convertToAssets	Internal	-	-
_deposit	Internal	Can Modify State	-
_withdraw	Internal	Can Modify State	-
_decimalsOffset	Internal	-	-

ERC4626Factory			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
newERC4626Vault	External	Can Modify State	-
setMaxTransfer	External	Can Modify State	onlyOwner

ERC4626Vault			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC4626 ERC20
batchDeposit	Public	Can Modify State	onlyOwner
smartBatchDeposit	Public	Can Modify State	onlyOwner
deposit	Public	Can Modify State	-
redeem	Public	Can Modify State	-
helperRedeem	Public	Can Modify State	onlyOwner
autoHelperRedeem	Public	Can Modify State	onlyOwner

4.3 Vulnerability Summary

[N1] [Information] Potential risk that the `getUnderlyingAsset` function can be manipulated

Category: Others

Content

In the InterestEnhanced contract, the getUnderlyingAsset function is used to calculate the amount of Curve pool liquidity that can be withdrawn based on the yearnVault shares held by the bPool. It first gets the current LP amount held by the bPool via getUnderlyingLP, then calculates the withdrawable liquidity amount from the Curve pool via the `calc_withdraw_one_coin` function. However, note that `calc_withdraw_one_coin` allows computing single-sided liquidity withdrawals when the pool is imbalanced. This means the value returned by `calc_withdraw_one_coin` may be incorrect if the pool is maliciously manipulated.

Currently getUnderlyingAsset is only a view function, and no other protocol functions use it, so there is no risk. But in the future, care should be taken not to incorporate getUnderlyingAsset into critical business logic.

Code location: src/InterestEnhanced.sol#L61

```
function getUnderlyingAsset() external view returns (uint256) {
  uint256 underlyingLP = getUnderlyingLP();
  uint256 expected = tricryptoUSDC.calc_withdraw_one_coin(underlyingLP, 0);

  return expected;
}
```

Solution

It is recommended to use getUnderlyingAsset cautiously, only as a frontend view function, and avoid it for critical logic.

Status

Acknowledged; After communicating with the project team, the project team stated that this function will only be used on the front end.

[N2] [Medium] Potential risk of admin role acting maliciously

Category: Authority Control Vulnerability Audit

Content

In the InterestEnhanced contract, the admin role can enable/disable investing in yearn by calling the startToEarn and clearPosition functions. When starting to invest, the bPool first adds liquidity to the Curve pool to get LP shares, then

stakes the LP into yearn for farming. When closing the position, the protocol reverses the operations to return funds to the bPool. However, note that liquidity is removed from Curve using the `remove_liquidity_one_coin` function. As is well known, `remove_liquidity_one_coin` is highly vulnerable to manipulation of imbalanced pools when the `minAmount` is 0. This introduces potential admin griefing risks.

For example: A malicious admin could first initiate `startToEarn`, then use flashloans or large amounts of funds to imbalance the Curve pool extremely. Finally calling `clearPosition` would cause the bPool to remove liquidity single-sided from the imbalanced pool. This results in the bPool receiving far less funds than expected, with the missing amount left in the pool. The attacker profits greatly from reverse swapping the funds originally belonging to the bPool.

Code location: src/InterestEnhanced.sol#L115

```
function _redeem(uint256 i, uint256 minAmount) internal returns (uint256) {
  uint256 share = getCurveLP();
  etf.invokeRedeem(address(tricryptoUSDC), share, i, minAmount);

  uint256 returnedAsset = getAssetAmount();
  return returnedAsset;
}
```

Solution

One is to use `remove_liquidity` instead of `remove_liquidity_one_coin` for removing liquidity. Because `remove_liquidity` will return more of the other token types in the pool when imbalanced. The bPool can swap those tokens to the originally desired token to mitigate admin griefing risks.

The other is to use an LP price oracle. Calculate the actual single-sided token amount that could be obtained based on the LP price, and use that as the `minAmount` to prevent the risk.

Using `remove_liquidity` or setting a proper `minAmount` via an LP price feed would help ensure the bPool receives the expected value. This protects against potential griefing by a malicious admin exploiting imbalanced pools.

Status

Acknowledged; After communicating with the project team, the project team indicated that the admin role is trusted

and will ensure the passed in `minAmount` is correct. However, this still has the risk of excessive privileges for the admin role.

[N3] [Medium] Potential denial of service risk for batchDeposit operations

Category: Denial of Service Vulnerability

Content

In the ERC4626Vault contract, the batchDeposit function called by the owner is used to deposit user funds into the Pool via the joinSmartPool function of the actions contract. After deposit, it checks if the asset balance in the contract is less than or equal to 1 via `_asset.balanceOf(address(this)) <= 1`. However, note that when the actions contract has token balance remaining after `joinPool` completes, it will send the remaining tokens to the ERC4626Vault contract. This will result in the ERC4626Vault having an asset token balance greater than 1, causing the owner to be unable to successfully perform batchDeposit. An attacker can exploit this for a DoS attack, or it can happen if users incorrectly send assets to the actions contract.

Code location: contracts/ERC4626Vault.sol#L70

```
    function batchDeposit(uint poolAmountOut) public onlyOwner {
        ...
        if (bal > 0) {
            ...
            require(_asset.balanceOf(address(this)) <= 1, "ERR_ASSET_NOT_ENOUGH");
            ...
        }
    }
```

Solution

It is recommended to directly check if the pool share balance of the ERC4626Vault contract is greater than the expected `poolAmountOut` after `joinPool`, instead of checking the asset balance. But note this requires the `poolAmountOut` passed in by the owner to be able to use the full asset balance of the contract for adding liquidity.

Status

Acknowledged; After communicating with the project team, the project team stated that in the event of a malicious DoS attack, the `smartBatchDeposit` function will be used to achieve normal batch-buy functions.

[N4] [Low] Inconsistency issue between issue amount and contract asset balance

Category: Design Logic Audit

Content

In the ERC4626Vault contract, the smartBatchDeposit function is used to deposit user funds into the ETF via the actions contract's autoJoinSmartPool function. It is called by the owner and passes in the corresponding issueAmount and minPoolAmountOut. However, it does not check if the asset balance of the contract equals the issueAmount. If the issueAmount passed in by the owner is much less than the contract's asset balance, it will result in the vault obtaining much less pool shares than expected. This can confuse vault users, since the pool share amount they receive on redeem does not match the raiseAsset amount they deposited.

Code location: contracts/ERC4626Vault.sol#L78

```
    function smartBatchDeposit(
        uint issueAmount,
        ...
    ) public onlyOwner {
        IERC20 _asset = IERC20(asset());

        _asset.approve(address(actions), issueAmount);
        IDesynActions(actions).autoJoinSmartPool(pool, kol, issueAmount,
minPoolAmountOut, handleToken, swapBase, swapDatas);
        ...
    }
```

Solution

It is recommended to not pass issueAmount externally, but directly use the current asset balance of the vault contract as the issueAmount for the autoJoinSmartPool operation.

Status

Acknowledged; After communicating with the project team, the project team stated that the owner role is trustworthy and will ensure that the incoming `issueAmount` is as expected.

[N5] [Critical] Incorrect `total` computation in `autoHelperRedeem`

Category: Design Logic Audit

Content

In the ERC4626Vault contract, the autoHelperRedeem function is used to distribute pool shares to users who have

deposits in the vault. It calls `_redeemForHelper` in a for loop to redeem for each user. To avoid out-of-gas, the max loop count is maxTransfer, otherwise it is the difference between waitLiquidationUsers and hasLiquidationUser. The loop index starts from hasLiquidationUser and ends at total. However, note that total is capped at maxTransfer. When hasLiquidationUser exceeds maxTransfer, the loop start index will be greater than total. This will cause the for loop to fail, leading to inability to perform `autoHelperRedeem`.

Code location: contracts/ERC4626Vault.sol#L122

```
    function autoHelperRedeem() public onlyOwner {
        require(crpFactory.isCrp(asset()), "ERR_NOT_ETF");

        uint total = waiitLiquidationUsers - hasLiquidationUser >
factory.maxTransfer() ? factory.maxTransfer() : waiitLiquidationUsers -
hasLiquidationUser;

        for (uint i = hasLiquidationUser; i < total; i++) {
            _redeemForHelper(liquidationUsers[i]);
        }
        hasLiquidationUser += total;
        ...
    }
```

Solution

It is recommended that after total calculation, hasLiquidationUser should be added as the termination index of the for loop.

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002401310003	SlowMist Security Team	2024.01.26 - 2024.01.31	Medium Risk

Summary conclusion: The SlowMist security team uses a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 1 critical risk, 2 medium risks, and 1 low risk. All the findings were fixed. The

code was not deployed to the mainnet. Due to N3 issues, the conclusion is still the medium risk.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist