BLOCKSEC

Security Audit

Report for Allstake Solidity Client Contracts

Date: July 31, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Allstake
Target	Allstake Solidity Client Contracts

Version History

Version	Date	Description
1.0	July 31, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-07-31 07:51:16+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is on the Allstake Solidity Client Contracts[1] of the Allstake. The Allstake is the first Omnichain Meshed Restaking protocol that brings restaking to all chains, enabling Actively Validated Services (AVS)/App chain to derive security from multiple assets simultaneously. By focusing on modularization and decoupling consensus and execution, Allstake extends beyond Ethereum, establishing connections between restaking hubs and all chains. It provides a comprehensive framework for AVS to utilize a diverse range of restaked assets, including ETH, BTC, NEAR, and SOL, etc..

Please note that the audit scope is limited to the following smart contracts:

```
1 src/Strategy.sol
2 src/StrategyManager.sol
```

Listing 1.1: Audit Scope for this Report

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Allstake Solidity Client Contracts	`Version 1`	`0b6cf9c95c4c0d1f8a856e20cffed2a89abb5c0d`
	`Version 2`	`3effb24cf58250fae7589c11404c1561c920f8c9`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

[1] https://github.com/allstake/allstake-solidity-client-contracts

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism

- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**,

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Medium, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we did not find potential security issues. Besides, we have **four** recommendations and **two** notes.

- Recommendation: 4
- Note: 2

ID	Severity	Description	Category	Status
1	-	Lack of early termination in Function `_assertStrategyValid()`	Recommendation	Fixed
2	-	Improper error messages in functions `deposit()` and `queueWithdraw()`	Recommendation	Fixed
3	-	Lack of invoking function `_disableInitializers()`	Recommendation	Fixed
4	-	Lack of out-of-bounds checks in functions `userPendingWithdrawalRequestIds()` and `listStrategies()`	Recommendation	Confirmed
5	-	Potential centralization risks	Note	-
6	-	Potential incorrect versioned initialization for proxy's upgrade	Note	-

The details are provided in the following sections.

2.1 Additional Recommendation

2.1.1 Lack of early termination in Function `_assertStrategyValid()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `StrategyManager`, there is no need to continue the loop in the function `_assertStrategyValid()` when `strategies[i] == strategy`.

```
182   function _assertStrategyValid(IStrategy strategy) private view {
183       uint8 i;
184       bool valid = false;
185       for (i = 0; i < strategiesLen(); i++) {
186           if (strategies[i] == strategy) {
187               valid = true;
188           }
189       }
190       require(valid, "StrategyManager: Invalid strategy address");
191   }
```

Listing 2.1: src/StrategyManager.sol

Suggestion Terminate the loop when `strategies[i] == strategy`.

2.1.2 Improper error messages in functions `deposit()` and `queueWithdraw()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Functions `deposit()` and `queueWithdraw()` both emit error messages "Strategy: Zero shares" when the number of shares to deposit or withdraw is zero. This could lead to confusion for users since they get the same messages for different operations.

```
63  function deposit(
64      uint256 amount,
65      address user
66  ) external onlyStrategyManager returns (uint256) {
67      require(amount >= minDeposit, "Strategy: Min deposit amount not met");
68
69      uint256 shares = balanceToShares(amount);
70      require(shares > 0, "Strategy: Zero shares");
71
72      token.safeTransferFrom(user, address(this), amount);
73      totalShares += shares;
74      userShares[user] += shares;
75
76      return shares;
77  }
78
79
80  /// @inheritdoc IStrategy
81  function queueWithdraw(
82      uint256 shares,
83      address user
84  ) external onlyStrategyManager returns (uint32) {
85      require(userShares[user] >= shares, "Strategy: Invalid shares");
86      require(shares > 0, "Strategy: Zero shares");
87
88      userShares[user] -= shares;
89
90      WithdrawalRequest memory request;
91      request.user = user;
92      request.shares = shares;
93      request.timestamp = block.timestamp;
94      request.pending = true;
95
96      withdrawalRequests.push(request);
97      uint32 requestId = uint32(withdrawalRequests.length - 1);
98      pendingWithdrawals[user].add(requestId);
99
100     return requestId;
101 }
```

Listing 2.2: src/Strategy.sol

Suggestion Use different error messages in functions `deposit()` and `queueWithdraw()`.

2.1.3 Lack of invoking function `_disableInitializers()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Currently, there is no invocation of function `_disableInitializers()`, which is also not implemented, in both the contracts `Strategy` and `StrategyManager`. In this case, the implementation contracts can be initialized and can result in potential risks.

```
51  function initialize(
52      IERC20 _token,
53      uint256 _withdrawDelay,
54      uint256 _minDeposit
55  ) external versionedInitializer {
56      token = _token;
57      withdrawDelay = _withdrawDelay;
58      minDeposit = _minDeposit;
59      strategyManager = msg.sender;
60  }
```

Listing 2.3: src/Strategy.sol

```
34  function initialize(
35      uint256 _minWithdrawDelay,
36      address _owner
37  ) external versionedInitializer {
38      _transferOwnership(_owner);
39      minWithdrawDelay = _minWithdrawDelay;
40  }
```

Listing 2.4: src/StrategyManager.sol

```
18  abstract contract VersionedInitializable {
19      /**
20       * @dev Indicates that the contract has been initialized.
21       */
22      uint256 private lastInitializedRevision = 0;
23
24      /**
25       * @dev Indicates that the contract is in the process of being initialized.
26       */
27      bool private initializing;
28
29      /**
30       * @dev Modifier to use in the initializer function of a contract.
31       */
32      modifier versionedInitializer() {
33          uint256 revision = getRevision();
34          require(
35              initializing ||
36                  isConstructor() ||
37                  revision > lastInitializedRevision,
38              "Contract instance has already been initialized"
```

```
39          );
40
41          bool isTopLevelCall = !initializing;
42          if (isTopLevelCall) {
43              initializing = true;
44              lastInitializedRevision = revision;
45          }
46
47          _;
48
49          if (isTopLevelCall) {
50              initializing = false;
51          }
52      }
53
54      /**
55       * @notice Returns the revision number of the contract
56       * @dev Needs to be defined in the inherited class as a constant.
57       * @return The revision number
58       */
59      function getRevision() internal pure virtual returns (uint256);
60
61      /**
62       * @notice Returns true if and only if the function is running in the constructor
63       * @return True if the function is running in the constructor
64       */
65      function isConstructor() private view returns (bool) {
66          // extcodesize checks the size of the code stored in an address, and
67          // address returns the current address. Since the code is still not
68          // deployed when running a constructor, any checks on its code size will
69          // yield zero, making it an effective way to detect if a contract is
70          // under construction or not.
71          uint256 cs;
72          //solium-disable-next-line
73          assembly {
74              cs := extcodesize(address())
75          }
76          return cs == 0;
77      }
78
79      // Reserved storage space to allow for layout changes in the future.
80      uint256[50] private ______gap;
81  }
```

Listing 2.5: src/utils/VersionedInitializable.sol

Suggestion Implement the function `_disableInitializers()` in the implementation contracts and invoke the function `_disableInitializers()` in the constructors of `Strategy` and `StrategyManager`.

2.1.4 Lack of out-of-bounds checks in functions `userPendingWithdrawalRequestIds()` and `listStrategies()`

Status Confirmed

Introduced by `Version 1`

Description Functions `userPendingWithdrawalRequestIds()` and `listStrategies()` will return elements indexed by the `offset` and `limit` parameters. However, there is no check on `offset` + `limit`, and may trigger out-of-bound revert.

```
161   function userPendingWithdrawalRequestIds(
162       address user,
163       uint32 offset,
164       uint32 limit
165   ) external view returns (uint32[] memory) {
166       uint32[] memory ids = new uint32[](limit);
167       uint32 i;
168       for (i = 0; i < limit; i++) {
169           ids[i] = uint32(pendingWithdrawals[user].at(offset + i));
170       }
171
172       return ids;
173   }
```

Listing 2.6: src/Strategy.sol

```
48   function listStrategies(
49       uint256 offset,
50       uint256 limit
51   ) external view returns (IStrategy[] memory) {
52       IStrategy[] memory result = new IStrategy[](limit);
53       uint256 i;
54       for (i = 0; i < limit; i++) {
55           result[i] = strategies[i + offset];
56       }
57       return result;
58   }
```

Listing 2.7: src/StrategyManager.sol

Suggestion Add checks to ensure `offset` plus `limit` is in range.

Feedback from the Project These functions are meant to be called by UI component after it reads the array length on chain. So in practice it won't run out-of-bounds. Besides, even if there is a check for array bound, it would just be another assert and causes revert.

2.2 Note

2.2.1 Potential centralization risks

Introduced by `Version 1`

Description In the project, there is a privileged account owner, which can create a strategy, disable a strategy, and upgrade contracts. If the owner's private key is lost or maliciously exploited, it could potentially cause losses to the protocol and users.

Feedback from the Project The ownership will be transferred to a multisig DAO soon after the contract is deployed.

2.2.2 Potential incorrect versioned initialization for proxy's upgrade

Introduced by `Version 1`

Description The `Strategy` contract uses a versioned initialization method to enable reinitialization after the proxy upgrade. The function `initialize()` will set the `strategyManager` as the `msg.sender` which is the strategy's manager during the first initialization. If the function `initialize()` is called again during the proxy upgrade, the `msg.sender` will become the proxy's admin rather than the strategy's manager, which is incorrect.

```
51  function initialize(
52      IERC20 _token,
53      uint256 _withdrawDelay,
54      uint256 _minDeposit
55  ) external versionedInitializer {
56      token = _token;
57      withdrawDelay = _withdrawDelay;
58      minDeposit = _minDeposit;
59      strategyManager = msg.sender;
60  }
```

Listing 2.8: src/Strategy.sol

Feedback from the Project When upgrading the contract, we will directly update the existing contracts instead of inheriting from it. So this initialized function will be rewritten accordingly, which won't try to update the address of the strategy manager again.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS