verichains

SECURITY AUDIT OF

ASHWARD SMART CONTRACTS

Public Report

Apr 14, 2022

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report prepared by Verichains Lab on Apr 14, 2022. We would like to thank the ASHWARD for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the ASHWARD Smart Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified one vulnerable issue in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY **5**

1.1. About ASHWARD Smart Contracts **5**

1.2. Audit scope **5**

1.3. Audit methodology **5**

1.4. Disclaimer **6**

2. AUDIT RESULT **7**

2.1. Overview **7**

2.2. Findings **8**

2.2.1. OpenboxGenesis.sol - User can reuse serverHash and serverSig in buyBox function MEDIUM 8

2.3. Additional notes and recommendations **9**

2.3.1. VestingLiquidity.sol - Sum of releasing token percentages not equal 100% INFORMATIVE .. 9

2.3.2. StakingRewards.sol - The getReward function call may cost larger than the gas limit INFORMATIVE 10

2.3.3. Vesting.sol - The GRANT_ROLE user may withdraw all tokens from vesting contract INFORMATIVE 11

3. VERSION HISTORY **12**

1. MANAGEMENT SUMMARY

1.1. About ASHWARD Smart Contracts

Ashward is a virtual world where players can own, and monetize their gaming experiences in the Binance Smart Chain (BSC) network using ASC, the platform's utility token. Players will be able to explore the fantasy medieval land, gather the bravest warriors to defeat aggressive bosses or engage in fierce battles against other players.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the smart contracts of ASHWARD Smart Contracts. It was conducted on commit a589c98bb89b7d939f8c2c8e5c72a90dbcc59327 from git repository *https://github.com/ashward-game/contract-ashward*.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- 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:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The ASHWARD Smart Contracts was written in Solidity language, with the required version to be ^0.8.9.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
bb6a2e80230ad150ea0ce97a33f0b15e0dfa32686aed27b0bd1710c7dd570bc3	**Antibots.sol**
0ccf36d615b7b4193a4f71ff9bfd0958e09f059f8536d5b87735d7c0d67df8e6	**Vesting.sol**
a2f369f20b697643047e38167e36e84c0e13e4eda32c177327e1805f27fae1d7	**OwnerTransferable.sol**
cd68aad88870e0b39e620a4747f4e8ee12cb1307b706a2eb10aaac71c8d2873d	**MakeRand.sol**
61c4cbb07a1ccfbaafc75acdf73532ae5b0d02a7a8699b7f5865a4b3989b96e0	**Marketplace.sol**
f7e8f7c0a250dfe5de2e3e5406f34aee5569b58caf0b673cca12c83c16686322	**MarketplaceBNB.sol**
7780cbf3985a8bc3cbeeb145f5d1fe02343f11cb89b6e261efe9dd8e72c85ad5	**NFT.sol**
b4805b2f6a3c32743c06433de94d3956f437fff738df57b73a1425dbf4059c28	**OpenboxGenesis.sol**
a608e6e7dafba4d7b6e139c970749f1210f7dd15eefbc89553ec05d14ad0d457	**StakingRewards.sol**
42b81eae676c33713b8293e1586d05dfa00f1ce14c708b8eb43a51f38e2b6352	**VestingAdvisory.sol**
5dbaceb9f61a145a828189164f36b99afcde4645e7bfda9dc5c1ab3a75501ea2	**VestingIDO.sol**
d6d3be398b321485477de7203420b4753cb539222ef4ffb1d559d4be509eed79	**VestingLiquidity.sol**
933cd29babef14aec27f0ba72b6668a226819e1c40aad79533ef50d2106038ec	**VestingMarketing.sol**
b39aba94e97f8bc8a4f758220113a4d7cd2a9ba21b9b68b9613dfb41c441d54d	**VestingPlay2Earn.sol**
fbf1cce873360b65a1869d302addd47b2adc051f9382168d533a23609dcd082d	**VestingPrivate.sol**
3bb29527f904eb4d710265cdea9e543842b0ed3e4df333e3414c80d5403b760b	**VestingReserve.sol**
1e2f13ac81e99b6d989834173cc35523e788c0609ddeddccc6958f615082ee65	**VestingStaking.sol**
a7c1adf17cb525a50990713279f7143ae8fc3994f565d0f6ce91d38c7cf58601	**VestingStrategicPartner.sol**
4db32c4766291c2e919e1f2970af648e13c39ade811a7efade6b8fa07a405fed	**VestingTeam.sol**

2.2. Findings

During the audit process, the audit team found one vulnerability issue in the given version of ASHWARD Smart Contracts.

2.2.1. OpenboxGenesis.sol - User can reuse serverHash and serverSig in buyBox function MEDIUM

The buyBox function uses the _commit function to verify serverHash and serverSig values and have been used or not. But there is a flow in the _commit function that allows users to reuse the serverHash and serverSig

```
function buyBox(
        BoxGrade grade,
        bytes32 serverHash,
        bytes memory serverSig,
        bytes32 clientRandom
    ) external payable canBuy {
    ...
    _commit(serverHash, serverSig, clientRandom);
    _buyBox(msg.sender, grade, serverHash, clientRandom);
}
```

Snippet 1. OpenboxGenesis.sol - the verify statement in the `buyBox` function

```
function _commit(
        bytes32 _serverHash,
        bytes memory _signature,
        bytes32 _clientRandom
    ) internal isFreshCommit(_serverHash) verified(_serverHash, _signatur…
  e) {
    _commitments[_serverHash] = _clientRandom;
    }
```

Snippet 2. MakeRand.sol - the`_commit` function

```
modifier isFreshCommit(bytes32 hashBytes) {
        require(
            _commitments[hashBytes] == 0,
            "MakeRand: hash value already exists"
        );
        _;
    }
```

Snippet 3. MakeRand.sol - The modifier checks `@serverHash`

The _commit function uses isFreshCommit modifier to verify hashBytes. After a _commit function is called, the mapping value of serverHash will be set by the clientRandom value which the user may control. If the mapping value of hashBytes is different from 0, the transaction will revert. Therefore, the user may pass the clientRandom with 0 in the buyBox function to reuse the serverHash and serverSig.

Besides, the logic using serverHash and clientRandom values isn't clear. Users still control the clientRandom value, we don't know how it is used in the server, it may be a risk.

RECOMMENDATION

We suggest adding a require statement in the buyBox function ensures the clientRandom value is different from 0. The ASHWARD team should review the using logic of serverHash and clientRandom values carefully.

UPDATES

- *Apr 14,2022*: This issue has been acknowledged and fixed by the ASHWARD team.

2.3. Additional notes and recommendations

2.3.1. VestingLiquidity.sol - Sum of releasing token percentages not equal 100% INFORMATIVE

The VestingLiquidity contract is used to release the token following milestones. For each milestone, the releasePercent value is set in the contructor. However, we found that the sum of releasePercents in all mileStone is not equal 100%. It only equals 80%.

Maybe, the ASHWARD team missed setting the _tge_percent value or the mileStone number is incorrect.

```
constructor(address token) Vesting(token) {
        _tge_percent = 0; // 0.00%
        _claimableMilestones = [
            1647621000,
            1658161800,
            1660840200,
            1663518600,
            1666110600,
            1668789000,
            1671381000,
            1674059400,
            1676737800,
            1679157000
```

```
        ];

        _claimablePercents[1658161800] = 889; // 2022-07-18 16:30:00UTC
        _claimablePercents[1660840200] = 889; // 2022-08-18 16:30:00UTC
        _claimablePercents[1663518600] = 889; // 2022-09-18 16:30:00UTC
        _claimablePercents[1666110600] = 889; // 2022-10-18 16:30:00UTC
        _claimablePercents[1668789000] = 889; // 2022-11-18 16:30:00UTC
        _claimablePercents[1671381000] = 889; // 2022-12-18 16:30:00UTC
        _claimablePercents[1674059400] = 889; // 2023-01-18 16:30:00UTC
        _claimablePercents[1676737800] = 889; // 2023-02-18 16:30:00UTC
        _claimablePercents[1679157000] = 888; // 2023-03-18 16:30:00UTC

    }
```

RECOMMENDATION

The ASHWARD team should review those values and update the contructor.

UPDATES

- *Apr 14, 2022*: This issue has been acknowledged by the ASHWARD team.

2.3.2. StakingRewards.sol - The getReward function call may cost larger than the gas limit INFORMATIVE

The getRewards function use an internal function to calculate the tokens that the investors may receive.

```
function getRewards() public onlyStaker {
        uint256 amount = _getRewards(msg.sender);
        if (amount == 0) return;

        totalRewarded += amount;
        rewardsToken().safeTransfer(msg.sender, amount);
        emit RewardsPaid(msg.sender, amount);
    }
```

Snippet 4. StakingReward.sol - The getRewards function

The internal function - _getRewards function uses a for loop to calculate. Therefore, the cost may be over the gas limit.

```
function _getRewards(address staker) private returns (uint256) {
        uint256 index = _stakes[staker];
```

```
        uint256 amount = 0;
        for (uint256 i = 0; i < _stakeholders[index].stakes.length; i++) …
  {
            amount += _calculateStakeRewards(_stakeholders[index].stakes[…
  i]);
            _stakeholders[index].stakes[i].since = block.timestamp;
        }
        return amount;
    }
```

Snippet 5. StakingRewards.sol - The for-loop in _getRewards function may cause an issue

If the length of stake array is too large, the cost may be over the gas limit which cause the transaction is reverted.

RECOMMENDATION

We suggest changing the stake struct or setting a stake.length limited to avoid this issue.

UPDATES

- *Apr 14, 2022*: This issue has been acknowledged by the ASHWARD team.

2.3.3. Vesting.sol - The GRANT_ROLE user may withdraw all tokens from vesting contract INFORMATIVE

The Vesting contract implements collectToken function which allow the GRANT_ROLE user to withdraw all tokens of the contract.

Besides, the GRANT_ROLE user may also withdraw tokens through creating a new beneficiary by addBeneficiaries function.

The contracts which inherit Vesting contract are VestingAdvisory, VestingIDO, VestingLiquidity, VestingMarketing, VestingPlay2Earn, VestingPrivate, VestingReserve, VestingStaking, VestingStrategicPartner and VestingTeam.

We submit this issue to notice the ASHWARD team to avoid some risks in the future.

UPDATES

- *Apr 14, 2022*: This issue has been acknowledged by the ASHWARD team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Mar 23, 2022*	Private Report	Verichains Lab
1.1	*Apr 14, 2022*	Public Report	Verichains Lab

Table 2. Report versions history