Security Audit Report for Windranger Auction Contract

Date: Feb 25, 2022

Version: 1.2

Contact: contact@blocksecteam.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.02.25 18:25:16 +08'00'

Contents

Report Manifest

Item	Description
Client	Windranger Auction
Target	Windranger Auction Contract

Version History

Version	Date	Description
1.0	Feb 19, 2022	First Release
1.1	Feb 21, 2022	Status Update
1.2	Feb 25, 2022	New Commit

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. 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 auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values of the repo [1] during the audit are shown in the following.

Contract Name	Stage	Commit SHA
windranger-auction	Initial	823c23966b9c16d2a999a184f8ebe354b2ead8c4
windranger-auction	Final	b6ff3a26644e9e6a148033d5bb900f456a6d27a1

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.

This audit report is not an endorsement of any particular project or team, and the report do 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).

[1] https://github.com/windranger-io/auction-contracts

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
- Access control
- 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]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

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

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

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

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

Chapter 2 Findings

In total, we find `five` potential issues in the smart contract. We also have `three` recommendations, as follows:

- High Risk: 2
- Medium Risk: 1
- Low Risk: 2
- Recommendations: 3

ID	Severity	Description	Category	Status
1	Medium	Incorrect Check of Parameters for the `selectWinner()` Function	Software Security	Fixed
2	High	Incorrect Initialization Pattern	Software Security	Fixed
3	High	Unsafe Signatures	Software Security	Fixed
4	Low	Potential Denial-of-Service Attack	DeFi Security	Fixed
5	Low	Inconsistent Auction Design	DeFi Security	Confirmed
6	-	Remove Unused State Variable	Recommendation	Fixed
7	-	Remove Unused `receive()` Function	Recommendation	Fixed
8	-	Remove Unused Inherited Contract	Undetermined	Fixed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Incorrect Check of Parameters for the `selectWinner()` Function

Status Fixed

Description The `selectWinner()` function in both `Auction` and `AuctionWithTime` contracts accept a series of signatures in three arrays which specify the raw data (i.e., `R, S, V`). However, there is a duplicate check for the length of `sigsV` and `sigsR`, while leaving `sigS` unchecked.

```
57      function selectWinners(
58              address[] calldata bidders,
59              uint256[] calldata bids,
60              bytes32[] calldata sigsR,
61              bytes32[] calldata sigsS,
62              uint8[] memory sigsV,
63              uint256 startID
64          ) external onlyOperator returns (uint256) {
65              require(bidders.length <= items, "Too much winners");
66              require(bidders.length == bids.length, "Incorrect number of bids");
67              require(
68                  bidders.length == sigsV.length &&
69                      sigsV.length == sigsR.length &&
70                      sigsV.length == sigsR.length,
71                  "Incorrect number of signatures"
```

```
72                 );
```

Listing 2.1: selectWinners():Auction.sol

Impact `sigsS` will not be checked as one may expect.

Suggestion Fix the incorrect check.

2.1.2 Incorrect Initialization Pattern

Status Fixed

Description In the `AuctionWithTime` contract, the initialization procedure is done in the `constructor()` function with the `initializer` modifier, which is not the recommended way of implementing the initialization logic in the upgradeable context. Instead, the correct implementation is to define a standalone `initialize()` function with the `initializer` modifier, and put all the related initialization logic there.

```
29    constructor(
30        uint256 startTime_,
31        uint256 endTime_,
32        IERC721 nft_,
33        uint256 items_,
34        IERC20Upgradeable weth_
35    ) initializer {
```

Listing 2.2: constructor():AuctionWithTime.sol

Impact The related variables cannot be properly initialized.

Suggestion Adopt the correct pattern to implement the initialization logic.

2.1.3 Unsafe Signatures

Status Fixed

Description The signatures used in the `selectWinners()` function only sign on the bid amount of each bidder in the auction. These signatures are subject to replay attacks. Specifically, these signatures can be used multiple times in this contract to maliciously extract `WETH` from the bidders. Furthermore, these signatures are generated from simple elements, which may also result in the signature reuse problem. Specifically, user signatures in other projects may be used in the `Auction` contract due to the limited information of the signatures.

```
57  function selectWinners(
58    address[] calldata bidders,
59    uint256[] calldata bids,
60    bytes32[] calldata sigsR,
61    bytes32[] calldata sigsS,
62    uint8[] memory sigsV,
63    uint256 startID
64  ) external onlyOperator returns (uint256) {
65    require(bidders.length <= items, "Too much winners");
66    require(bidders.length == bids.length, "Incorrect number of bids");
67    require(
68        bidders.length == sigsV.length &&
```

```
69          sigsV.length == sigsR.length &&
70          sigsV.length == sigsR.length,
71      "Incorrect number of signatures"
72  );
73  uint256 minted = 0;
74  for (uint256 i = 0; i < bidders.length; i++) {
75      if (
76          ecrecover(
77              keccak256(
78                  abi.encodePacked(
79                      "\x19Ethereum Signed Message:\n32",
80                      keccak256(abi.encodePacked(bids[i]))
81                  )
82              ),
83              sigsV[i],
84              sigsR[i],
85              sigsS[i]
86          ) != bidders[i]
87      ) {
```

Listing 2.3: selectWinners():Auction.sol

Impact The signatures may be used multiple times to extract `WETH` from the bidders.

Suggestion Include more information to prevent replay attacks.

Feedback from the Developers The signatures are unchanged. We add a new state variable named `used` to ensure no address can provide more than one signature to the `selectWinners()` function.

2.1.4 Potential Denial-of-Service Attack

Status Fixed

Description In the `selectWinners()` function of the `AuctionWithTime` contract, for each bidder, the auction procedure is implemented as follows:

- verifying the signatures provided by the bidder.
- transferring WETH from the bidder to the beneficiary address.
- minting NFT to the bidder.

However, the logic implemented for the `AuctionWithTime` contract is not *fail-safe*. Specifically, the checks for the first two steps are implemented using the `require` statements. Therefore, the entire transaction will revert if the first two steps of any bidder fail to execute. This design is subject to Denial-of-Service attacks.

```
66  function selectWinners(
67      address[] calldata bidders,
68      uint256[] calldata bids,
69      bytes32[] calldata sigsR,
70      bytes32[] calldata sigsS,
71      uint8[] calldata sigsV,
72      uint256[] memory ids
73  ) external onlyOwner {
74      require(block.timestamp > endTime, "Auction hasn't ended");
75      require(bidders.length <= items, "Too much winners");
76      require(
```

```
77            bidders.length == ids.length && bidders.length == bids.length,
78            "Incorrect number of ids"
79        );
80        require(
81            bidders.length == sigsV.length &&
82                sigsV.length == sigsR.length &&
83                sigsV.length == sigsR.length,
84            "Incorrect number of signatures"
85        );
86        for (uint256 i = 0; i < bidders.length; i++) {
87            require(
88                ecrecover(
89                    keccak256(
90                        abi.encodePacked(
91                            "\x19Ethereum Signed Message:\n32",
92                            keccak256(abi.encodePacked(bids[i]))
93                        )
94                    ),
95                    sigsV[i],
96                    sigsR[i],
97                    sigsS[i]
98                ) == bidders[i],
99                "Incorrect signature"
100           );
101           weth.safeTransferFrom(bidders[i], beneficiaryAddress, bids[i]);
102           nft.mint(bidders[i], ids[i]);
103       }
104       items -= bidders.length;
105   }
```

Listing 2.4: selectWinners():AuctionWithTime.sol

Impact The `selectWinners()` function may subject to Denial-of-Service attacks.

Suggestion Use conditional checks rather than the `require()` statements.

2.2 DeFi Security

2.2.1 Inconsistent Auction Design

Status Confirmed

Description The auction procedure consists of two steps. Firstly, the bidders submit their signatures to the project. Secondly, the project invokes the `selectWinners()` function of the `Auction` contract with corresponding bids. For each bid to succeed, the bidder must also have sufficient `WETH` and approve to the `Auction` contract. The issue is that the bidders can control the `WETH`-related condition after the signatures are provided (i.e., the auction kicks off), which may lead to the following problems:

1. The bidders can force to cancel the auction after signatures are submitted, with or without front-running the selectWinners transaction.
2. A malicious bidder may batch submit false (higher) bids with multiple addresses (i.e., the signatures are correct, but the bid address has insufficient balance or allowance for `WETH`). This way may cause a

fake bloom of the NFT auction with a specific auction display interface, and turn down the enthusiasm of other competitors to undermine the valid offers. As a result, this malicious bidder is able to hide his/her true bids and purchase the items with a relatively lower price (even down to 0).

3. It results in the mismatching between the valid offers and the shown offers, which may break the confidence of the community.

Impact N/A

Suggestion Revise the auction design.

Feedback from the Developers

1. Only we ourselves can execute the `selectWinners()` function, so there won't be any DoS attacks. The users funds cannot be compromised either.
2. We don't allow making false bids (i.e., with zero bid amount) in UI strictly and only push bidders with bid amount larger than zero.
3. We do check to make sure that the allowance exceeds or is equal to the bid at the time the bid is made before saving it to the database. We do the check inside the contract to still be able to process the whole batch and just skip ones that revoked approval or withdrawn balance without demanding with requirement.

2.3 Additional Recommendation

2.3.1 Remove Unused State Variable

Status Fixed

Description The `startTime` state variable in the `AuctionWithTime` contract is not used.

Impact Unnecessary gas consumption.

Suggestion Remove the unused state variables.

2.3.2 Remove Unused `receive()` Function

Status Fixed

Description The `receive()` function of the `Auction` and `AuctionWithTime` contracts are unused, and there is no actual logic implemented. Besides, there is no way to withdraw the Ether sent to these contracts. Thus, the `receive()` function in both contracts should be removed.

Impact N/A

Suggestion Remove the unused `receive()` function.

2.3.3 Remove Unused Inherited Contract

Status Fixed

Description Both `Auction` and `AuctionWithTime` contracts inherit from the `PausableUpgradeable` contract of OpenZeppelin. However, the `PausableUpgradeable` contract has no public interface to pause and unpause the contract. Specifically, the `_pause()` and `_unpause()` functions of the `PausableUpgradeable` contract are private.

Impact Unnecessary gas consumption.

Suggestion Remove the unused inheritance.