Security Audit Report for Deal Contract

Date: Jan 07, 2022

Version: 1.1

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by BlockSec
Audit Team
Date: 2023.01.07 21:01:08
+08'00'

Contents

Report Manifest

Item	Description
Client	Deal.Art
Target	Deal Contract

Version History

Version	Date	Description
1.0	December 26, 2022	First Release
1.1	January 07, 2022	Second audit for new commits

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 5 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 audit target is the *Deal contract* [1]. 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. Note that, `Version 3` is the target of the second audit, while there exist some commits between `Version 2` and `Version 3`. For the sake of simplicity, the issues introduced before `Version 3` are aggregated to `Version 3` in this report.

Project	Version	Commit Hash
Deal Contract	Version 1	8dec93b25ffc0eb6769998cdab063e73c1d26da8
	Version 2	250e1962531e72612ac89ccaf85d4a7c1304e781
	Version 3	c52d36eb5cd576c673ece50d7f7f9d65e15a140b
	Version 4	79e21afc5da045724b48999096dc5fb61e7ab71b

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 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] https://github.com/deal-art/Swap-Contract

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	*High*	High	Medium
	Low	Medium	Low
		High	*Low*
		Likelihood	

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

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

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

Chapter 2 Findings

In total, we find **two** potential issues. We also have **two** recommendations.

- High Risk: 2
- Medium Risk: 0
- Low Risk: 0
- Recommendation: 2

ID	Severity	Description	Category	Status
1	High	Reentrancy in the swap function	Software Security	Fixed
2	High	Lack of sanity checks for ERC721 and ERC1155 token addresses	Software Security	Fixed
3	-	Consider the offer has duplicated ERC20 tokens	Recommendation	Fixed
4	-	Add approval checks before invoking the _swap function	Recommendation	Acknowledged

The details are provided in the following sections.

2.1 Software Security

2.1.1 Reentrancy in the swap function

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description There is a reentrancy in ERC721's safeTransferFrom function so that the attacker can use this issue to exit and rejoin a room with a different offer, which makes the room owner lose accepts a different offer.

Here is how the attack works. Suppose the host makes an offer using 10 ERC20 token A and 1 ERC721 token B. And the attacker joins the room by submitting the offer with 1 ERC 721 token C. The host thinks the deal is great and invokes swap(roomID, attacker) to finish the trade.

When invoking _swtapTokens(roomid, host, attacker), the ERC721 token will be transferred to the attacker using safeTransferFrom. The safeTransferFrom of ERC721 has an external call back to to (in this case is the attacker) (line 219). Then the attacker will invoke exitRoom and then joinRoom (or directly invoke updateOffer) with another offer (with useless ERC721 tokens) that is different from the original offer. After that, when the _swapTokens(roomID, counterparty, rooms[roomId].host) is invoked (line 197), the offer fetched from room.offers[attacker] is a different offer, which may make the host lose the money.

```
195    function _swap(bytes32 roomId, address counterparty) internal {
196        _swapTokens(roomId, rooms[roomId].host, counterparty);
197        _swapTokens(roomId, counterparty, rooms[roomId].host);
198        _saveDealRecords(roomId, counterparty);
199        _closeRoom(roomId);
200    }
201
```

```
function _swapTokens(
bytes32 roomId,
address from,
address to
) internal {
    Room storage room = rooms[roomId];
    Offer memory offer = room.offers[from];
    if (room.nonce != offer.offerNonce) revert EOfferExpired();
    for (uint256 i = 0; i < offer.erc20Tokens.length; i++) {
        offer.erc20Tokens[i].transferFrom(
        from,
        address(this),
        offer.erc20TokenAmounts[i]
        );
        offer.erc20Tokens[i].transfer(to, offer.erc20TokenAmounts[i]);
    }
    for (uint256 i = 0; i < offer.erc721Tokens.length; i++) {
        offer.erc721Tokens[i].safeTransferFrom(
        from,
        to,
        offer.erc721TokenIds[i]
        );
    }
    for (uint256 i = 0; i < offer.erc1155Tokens.length; i++) {
        offer.erc1155Tokens[i].safeBatchTransferFrom(
        from,
        to,
        offer.erc1155TokenIds[i],
        offer.erc1155TokenAmounts[i],
        ""
        );
    }
}
```

Listing 2.1: Deal.sol

Impact The counterpart's offer can be changed during the execution.

Suggestion Use reentrancy guard and add nonReentrant before external functions.

2.1.2 Lack of sanity checks for ERC721 and ERC1155 token addresses

Severity High

Status Fixed in `Version 4`

Introduced by `Version 3`

Description In the `_checkIfOfferMatchesIdealOffer` function, there is a new feature to support auto-swapping for any token from the collection (line 324 - 370). However, there does not exist the sanity checks for ERC721 and ERC1155 token addresses, which allows the attackers to use fake tokens to perform the swapping.

```
317 function _checkIfOfferMatchesIdealOffer(bytes32 roomId, Offer memory offer)
318     internal
```

```
        view
        returns (bool)
    {
        Offer memory idealOffer = rooms[roomId].idealOffer;
        if (cannotUseKeccakComparison[roomId]) {
            if (
                keccak256(
                    abi.encode(
                        idealOffer.erc20Tokens,
                        idealOffer.erc20TokenAmounts
                    )
                ) !=
                keccak256(
                    abi.encode(offer.erc20Tokens, offer.erc20TokenAmounts)
                ) ||
                idealOffer.erc721Tokens.length != offer.erc721Tokens.length ||
                idealOffer.erc1155Tokens.length != offer.erc1155Tokens.length
            ) {
                return false;
            }
            for (uint256 i = 0; i < idealOffer.erc721Tokens.length; i++) {
                if (idealOffer.erc721TokenIds[i] == MAX_UINT256) {
                    continue;
                }
                if (idealOffer.erc721TokenIds[i] != offer.erc721TokenIds[i]) {
                    return false;
                }
            }
            for (uint256 i = 0; i < idealOffer.erc1155Tokens.length; i++) {
                for (
                    uint256 j = 0;
                    j < idealOffer.erc1155TokenIds[i].length;
                    j++
                ) {
                    if (
                        idealOffer.erc1155TokenAmounts[i][j] !=
                        offer.erc1155TokenAmounts[i][j]
                    ) {
                        return false;
                    }
                    if (idealOffer.erc1155TokenIds[i][j] == MAX_UINT256) {
                        continue;
                    }
                    if (
                        idealOffer.erc1155TokenIds[i][j] !=
                        offer.erc1155TokenIds[i][j]
                    ) {
                        return false;
                    }
                }
            }
            return true;
        } else {
```

```
        idealOffer.offerNonce = offer.offerNonce;
        return
            keccak256(abi.encode(offer)) ==
            keccak256(abi.encode(idealOffer));
    }
}
```

Listing 2.2: Deal.sol

Impact The attackers could use fake tokens to swap valuable ones.

Suggestion Add sanity checks accordingly.

2.2 Additional Recommendation

2.2.1 Consider the offer has duplicated ERC20 tokens

Status Fixed in `Version 3`

Introduced by `Version 1`

Description If the ERC20 tokens can be duplicated, then the allowance checked here (line 285 - 286) is not correct.

```
278 function _checkIfAllTokensApproved(Offer memory offer, address from)
279     internal
280     view
281     returns (bool)
282 {
283     for (uint256 i = 0; i < offer.erc20Tokens.length; i++) {
284         if (
285             offer.erc20Tokens[i].allowance(from, address(this)) !=
286             offer.erc20TokenAmounts[i]
287         ) return false;
288     }
289     for (uint256 i = 0; i < offer.erc721Tokens.length; i++) {
290         if (
291             offer.erc721Tokens[i].getApproved(offer.erc721TokenIds[i]) !=
292             address(this) &&
293             !offer.erc721Tokens[i].isApprovedForAll(from, address(this))
294         ) return false;
295     }
296     for (uint256 i = 0; i < offer.erc1155Tokens.length; i++) {
297         if (!offer.erc1155Tokens[i].isApprovedForAll(from, address(this)))
298             return false;
299     }
300     return true;
301 }
```

Listing 2.3: Deal.sol

Impact N/A

Suggestion Check duplicated tokens.

Feedback from the project For this one, we are relying on our frontend to send the correct data. Checking for duplicates on-chain in an array could become costly and using an OpenZeppelin Set is also expensive. Since circumventing our frontend and forcing a duplicate value doesn't accomplish anything for anyone, it's in everyone's interest to input proper data.

2.2.2 Add approval checks before invoking the `_swap` function

Status Acknowledged

Introduced by Version 1

Description In the `_updateOffer` function, there exists the approval checks before invoking the `_swap` function.

```
    function _updateOffer(bytes32 roomId, Offer calldata offer) internal {
        rooms[roomId].offers[_msgSender()] = offer;
        emit OfferUpdated(roomId, _msgSender());
        if (_checkIfOfferMatchesIdealOffer(roomId, offer)) {
            if (
                _checkIfAllTokensApproved(offer, _msgSender()) &&
                _checkIfAllTokensApproved(
                    rooms[roomId].offers[rooms[roomId].host],
                    rooms[roomId].host
                )
            ) {
                _swap(roomId, _msgSender());
            }
        }
    }
}
```

Listing 2.4: Deal.sol

To keep consistency, it is recommended to add the similar checks in the `swap` function.

```
    function swap(bytes32 roomId, address counterparty)
        external
        whenNotPaused
        hostOnly(roomId)
    {
        _swap(roomId, counterparty);
    }
```

Listing 2.5: Deal.sol

Impact N/A

Suggestion Add the approval checks.

Feedback from the project I think it doesn't matter because if the check is added, then ideally we should revert if not everything is approved. But, as the code currently works, if not everything is approved the call will revert as well.