Security Audit Report for Halo Influencer Badge Contract

Date: June 19, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Halo
Target	Halo Influencer Badge Contract

Version History

Version	Date	Description
1.0	June 19, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-06-19 08:15:47+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 target of this audit is the code repository of Halo Influencer Badge Contract[1] of Halo.

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 MD5 values of the files 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
Halo Influencer Badge Contract	Version 1	4caa4b645991a50d38e75ecb0ec73c8a90ad8698
	Version 2	e14d0d7b881f77c26125d26aa35c5563b8f90a63

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.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

[1] https://github.com/halowalletdev/halo-influencer-badge-contract

- **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**, **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, **one** recommendation and **two** notes as follows:

- High Risk: 0
- Medium Risk: 2
- Low Risk: 0
- Recommendation: 1
- Note: 2

ID	Severity	Description	Category	Status
1	Medium	Inconsistent condition check	Defi Security	Confirmed
2	Medium	Lack of check in function buyFromPool()	Defi Security	Fixed
3	-	Lack of check in function setFeePercent()	Recommendation	Fixed
4	-	Potential centralization risk	Note	
5	-	Potential sandwich attack due to excessive maxPayIn	Note	

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Inconsistent condition check

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the contract `InfluencerBadge`, `isWhitelistKOL` is used to determine if an address is authorized to create a pool, while `isWhitelistPreminter` is used to determine if an address has `premint` privilege. Specifically, if `poolConfig.hasFinishPremint` is `false` (lines 224-227), only addresses for which `isWhitelistPreminter` returns true can invoke function `buyFromPool()` before ordinary users are allowed to mint. However, in the function `createBadgePool()`, `poolConfig.hasFinishPremint` is set based on the `msg.sender`'s status in `isWhitelistKOL` (line 193), which is inconsistent.

```
132    function createBadgePool(
133        address payToken,
134        uint256 constA,
135        uint256 constB,
136        uint256 revenueSharingPercent
137    )
138        external
139        callerIsUser
140        nonReentrant
141        whenNotPaused
142        returns (uint256 poolId)
143    {
144        // verify parameters
```

```
        if (isCheckCreator) {
            require(isWhitelistKOL[msg.sender], "NOT_IN_WL"); // in the whitelist
        }
        require(!isBlacklistKOL[msg.sender], "IN_BL"); // in the blacklist


        require(!hasCreatedPool[msg.sender], "HAS_CRED"); // has created


        if (isCheckConstA) {
            require(isWhitelistConstA[constA], "INV_CONSTA");
        }
        require(constB > 0, "INV_CONSTB");
        if (isCheckConstB) {
            require(isWhitelistConstB[constB], "INV_CONSTB");
        }
        // verify hmp
        if (isCheckHMPInCreation) {
            // check level
            require(getHMPLevel(msg.sender) >= hpmLevelThreshold, "INV_LEVEL");
        }
        require(isWhitelistPayToken[payToken], "NS_TOKEN"); // not supported token


        // get amountPerPayToken which is "10^n"
        uint256 amountPerPayToken;
        if (payToken == address(0)) {
            // native token
            amountPerPayToken = 10 ** 18;
        } else {
            uint256 decimals = IERC20WithDecimals(payToken).decimals();
            amountPerPayToken = 10 ** decimals;
        }


        require(revenueSharingPercent <= maxPercentInRevenueSharing, "INV_PCT");


        //---- verify success ---//
        hasCreatedPool[msg.sender] = true;
        // save pool's config
        uint256 newPoolId = ++currentIndex;


        BadgePoolConfig storage poolConfig = badgePoolConfigs[newPoolId];
        poolConfig.kol = msg.sender;
        poolConfig.payToken = payToken;
        poolConfig.amountPerPayToken = amountPerPayToken;
        poolConfig.tokenBalance = 0;
        poolConfig.constA = constA;
        poolConfig.constB = constB;
        poolConfig.varCoef1 = 1;
        poolConfig.varCoef2 = 1;
```

```
        poolConfig.revenueSharingPercent = revenueSharingPercent;
        if (isWhitelistKOL[msg.sender]) {
            poolConfig.hasFinishPremint = false; // false: need premint
        } else {
            poolConfig.hasFinishPremint = true; // do not need premint, just mark true
        }


        emit CreateBadgePool(msg.sender, newPoolId);
        return newPoolId;
    }
```

Listing 2.1: InfluencerBadge.sol

```
    function buyFromPool(
        uint256 poolId,
        uint256 buyAmount,
        uint256 maxPayIn
    )
        external
        payable
        callerIsUser
        nonReentrant
        whenNotPaused
        returns (uint256 payInAddFee)
    {
        // verify parameters
        BadgePoolConfig storage poolConfig = badgePoolConfigs[poolId];
        require(poolConfig.kol != address(0), "INV_ID");
        // verify msg.sender
        if (!poolConfig.hasFinishPremint) {
            require(isWhitelistPreminter[msg.sender], "NEED_PREMINT");
            poolConfig.hasFinishPremint = true;
        }


        require(
            buyAmount > 0 && maxPayIn > 0 && buyAmount <= maxLimitInBuyOrSell,
            "INV_AMT"
        );
        // verify hmp
        if (isCheckHMPInBuyOrSell) {
            // check level
            require(getHMPLevel(msg.sender) >= hpmLevelThreshold, "INV_LEVEL");
        }


        // calculate cost
        (uint256 buyPrice, uint256 protocolFee, uint256 kolFee) = getBuyPrice(
            poolId,
            buyAmount
        );
        // verify limit
```

```
        uint256 allFee = protocolFee + kolFee;
        payInAddFee = buyPrice + allFee;
        require(payInAddFee <= maxPayIn, "EX_AMT"); // exceeds max input amount


        //---- verify success ---//


        // pay: native or erc20
        address payToken = poolConfig.payToken;
        if (payToken == address(0)) {
            require(msg.value >= payInAddFee, "IF_AMT"); // insufficient payment amount
            // pay fees
            Address.sendValue(payable(protocolFeeTo), protocolFee);
            Address.sendValue(payable(poolConfig.kol), kolFee);
            // refund remaining
            uint256 refundAmount = msg.value - payInAddFee;
            if (refundAmount > 0) {
                Address.sendValue(payable(msg.sender), refundAmount);
            }
        } else {
            // msg.sender->this
            SafeERC20.safeTransferFrom(
                IERC20(payToken),
                msg.sender,
                address(this),
                buyPrice
            );
            // transfer fees
            // 1. msg.sender-> protocol fee recipient
            SafeERC20.safeTransferFrom(
                IERC20(payToken),
                msg.sender,
                protocolFeeTo,
                protocolFee
            );
            // 2. msg.sender-> kol
            SafeERC20.safeTransferFrom(
                IERC20(payToken),
                msg.sender,
                poolConfig.kol,
                kolFee
            );
        }
        // update pool balance
        poolConfig.tokenBalance += buyPrice;
        // // mint erc1155 to user
        _mint(msg.sender, poolId, buyAmount, "");
        emit Buy(
            poolId,
            msg.sender,
            buyAmount,
            payInAddFee,
```

```
300             buyPrice,
301             protocolFee,
302             kolFee,
303             poolConfig.tokenBalance
304         );
305     }
```

Listing 2.2: InfluencerBadge.sol

Impact The inconsistent condition check may not fit the protocol design.

Suggestion In line 193, replace `isWhitelistKOL` with `isWhitelistPreminter`.

Feedback from the project The addresses in `isWhitelistPreminter` are all halo's official addresses. That is to say that, only halo officials have premint privilege. We think that the pool created by the KOLs in the `isWhitelistKOL` whitelist is a high-quality pool. Halo official wants to mint part of badges before ordinary users, and later airdrop it to users as rewards. Therefore, in `createBadgePool()`, the judgment of whether need premint is based on `isWhitelistKOL`. And when buying in `buyFromPool()`, halo officials need to purchase it first and then open it to ordinary users.

2.1.2 Lack of check in function buyFromPool()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `InfluencerBadge`, users can mint `BADGE` by paying with the `payToken` set in `poolConfig` through the function `buyFromPool()`. However, the function `buyFromPool()` only supports one type of token as `payToken`. Specifically, if the `payToken` is an `ERC20` token and users mistakenly send native tokens when invoking the function, the contract does not refund these native tokens.

```
function buyFromPool(
    uint256 poolId,
    uint256 buyAmount,
    uint256 maxPayIn
)
    external
    payable
    callerIsUser
    nonReentrant
    whenNotPaused
    returns (uint256 payInAddFee)
{
    // verify parameters
    BadgePoolConfig storage poolConfig = badgePoolConfigs[poolId];
    require(poolConfig.kol != address(0), "INV_ID");
    // verify msg.sender
    if (!poolConfig.hasFinishPremint) {
        require(isWhitelistPreminter[msg.sender], "NEED_PREMINT");
        poolConfig.hasFinishPremint = true;
```

```
        }


        require(
            buyAmount > 0 && maxPayIn > 0 && buyAmount <= maxLimitInBuyOrSell,
            "INV_AMT"
        );
        // verify hmp
        if (isCheckHMPInBuyOrSell) {
            // check level
            require(getHMPLevel(msg.sender) >= hpmLevelThreshold, "INV_LEVEL");
        }


        // calculate cost
        (uint256 buyPrice, uint256 protocolFee, uint256 kolFee) = getBuyPrice(
            poolId,
            buyAmount
        );
        // verify limit
        uint256 allFee = protocolFee + kolFee;
        payInAddFee = buyPrice + allFee;
        require(payInAddFee <= maxPayIn, "EX_AMT"); // exceeds max input amount


        //---- verify success ---//


        // pay: native or erc20
        address payToken = poolConfig.payToken;
        if (payToken == address(0)) {
            require(msg.value >= payInAddFee, "IF_AMT"); // insufficient payment amount
            // pay fees
            Address.sendValue(payable(protocolFeeTo), protocolFee);
            Address.sendValue(payable(poolConfig.kol), kolFee);
            // refund remaining
            uint256 refundAmount = msg.value - payInAddFee;
            if (refundAmount > 0) {
                Address.sendValue(payable(msg.sender), refundAmount);
            }
        } else {
            // msg.sender->this
            SafeERC20.safeTransferFrom(
                IERC20(payToken),
                msg.sender,
                address(this),
                buyPrice
            );
            // transfer fees
            // 1. msg.sender-> protocol fee recipient
            SafeERC20.safeTransferFrom(
                IERC20(payToken),
                msg.sender,
```

```
            protocolFeeTo,
            protocolFee
        );
        // 2. msg.sender-> kol
        SafeERC20.safeTransferFrom(
            IERC20(payToken),
            msg.sender,
            poolConfig.kol,
            kolFee
        );
    }
    // update pool balance
    poolConfig.tokenBalance += buyPrice;
    // // mint erc1155 to user
    _mint(msg.sender, poolId, buyAmount, "");
    emit Buy(
        poolId,
        msg.sender,
        buyAmount,
        payInAddFee,
        buyPrice,
        protocolFee,
        kolFee,
        poolConfig.tokenBalance
    );
}
```

Listing 2.3: InfluencerBadge.sol

Impact Users may lose funds.

Suggestion Add a check to ensure that when the contract's `payToken` is an `ERC20` token, the user's `msg.value` is 0.

2.2 Additional Recommendation

2.2.1 Lack of check in function setFeePercent()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `InfluencerBadge`, the function `setFeePercent()` is used to set `protocolFeePercent` and `kolFeePercent`, but there is no validation for the input parameters.

```
function setFeePercent(
    uint256 newProtocolPercent,
    uint256 newKolFeePercent
) external onlyOwner {
    protocolFeePercent = newProtocolPercent;
    kolFeePercent = newKolFeePercent;
}
```

Listing 2.4: InfluencerBadge.sol

Suggestion Add checks to ensure the input parameters are less than `100`.

2.3 Note

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description In the protocol, various whitelist checks exist, and the contract owner can modify these whitelists through functions such as `addWhitelistKOLs()`, `addWhitelistPayTokens()`, etc. If the owner's private key is lost or maliciously exploited, it could lead to losses for the protocol.

Feedback from the project After the contract is deployed, we will transfer the ownership to a multisig wallet.

2.3.2 Potential sandwich attack due to excessive maxPayIn

Introduced by `Version 1`

Description According to the pricing formula for `BADGE`, the smaller the `totalSupply` corresponding to a `tokenId`, the lower the cost for minting the same quantity `BADGE`. Therefore, when ordinary users mint via the function `buyFromPool()`, malicious users can construct two transactions: 1. Mint the same quantity through `buyFromPool()`. 2. Sell the same quantity through `sellToPool()`. They can then use bribery to ensure that the ordinary user's transaction is executed between transactions 1 and 2, causing the user's expenditure to potentially exceed expectations. Hence, users should be cautious when setting the `maxPayIn`.

Feedback from the project In order to mitigate the impact of sandwich attacks, we will limit the maximum quantity of a single sell or buy. In addition, when users buy or sell badges through halo wallet, we will let them choose the slippage (maxPayIn= buyPrice*(1+ slippage)) and give a risk warning.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS