BLOCKSEC

Security Audit
Report for HaloMembershipPass

Date: April 24th, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Halo
Target	HaloMembershipPass

Version History

Version	Date	Description
1.0	April 24th, 2024	First Release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-04-24 05:50:02+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 the `HaloMembershipPass.sol` file within the HaloMembershipPass of Halo [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.

Project	Version	Commit Hash
HaloMembershipPass	`Version 1`	`dfe5ee0d6ce5bc337eb2f41ac12a7912a720d879`
HaloMembershipPass	`Version 2`	`4028449577bd3c0c53d0b7edaed0ea61df9fa84c`

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-membership-pass/blob/main/contracts/HaloMembershipPass.sol

- **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 **seven** potential issue. Besides, we also have **one** recommendations.

- Medium Risk: 4
- Low Risk: 3
- Recommendation: 1
- Note: 0

ID	Severity	Description	Category	Status
1	Medium	Incorrect Rounding Direction	DeFi Security	Confirmed
2	Low	Lack of Check on the Number of Level5 and Level6 in Function initialMint()	DeFi Security	Confirmed
3	Low	Incorrect Logic in Function _chargeMint-Fee()	DeFi Security	Fixed
4	Low	Lack of Check on Parameters in Function setLevel5and6Proportion()	DeFi Security	Confirmed
5	Medium	Centralization risk	DeFi Security	Confirmed
6	Medium	Potential Overpayments by Users	DeFi Security	Confirmed
7	Medium	Upgrade Failed due to Burning Large Quantities of Low-Level NFTs	DeFi Security	Fixed
8	-	Incorrect Method of Transferring Native Tokens	Recommendation	Fixed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect Rounding Direction

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the `HaloMembershipPass` contract, users can mint NFTs through the function `initialMint()`. In line 118, the total cost is calculated based on the number of minted NFTs and the price. However, the required mint fee is rounded down, which is disadvantageous for the protocol.

```
87   function initialMint(
88     bytes32[] calldata proof,
89     uint8[] calldata nftLevels,
90     uint256 discount,
91     address payCurrency
92   ) external payable callerIsUser nonReentrant whenNotPaused {
93     // Verify parameters
94     require(
95         initialMintMerkleRoot != 0x0 && block.timestamp >= startTimestamp,
96         "Not in initial mint period"
```

```
97      );
98      require(
99          proof.length > 0 &&
100             nftLevels.length > 0 &&
101             discount <= SCALE_DECIMAL,
102         "Invalid parameters"
103     );
104     require(isCurrencyEnabled[payCurrency], "Invalid currency");
105
106     require(!isMinted[msg.sender], "Already Minted");
107     //  Merkle verify
108     bytes32 leaf = keccak256(abi.encode(msg.sender, nftLevels, discount));
109     require(
110         MerkleProof.verify(proof, initialMintMerkleRoot, leaf),
111         "Invalid proof"
112     );
113     // Mark it minted
114     isMinted[msg.sender] = true;
115
116     // Charge the mint fee
117     uint256 nftAmount = nftLevels.length;
118     uint256 payAmount = (nftAmount *
119         currencyAmountPerToken[payCurrency] *
120         discount) / SCALE_DECIMAL;
121     _chargeMintFee(payCurrency, payAmount);
122
123
124     // Mint tokens
125     for (uint256 i = 0; i < nftAmount; i++) {
126         uint256 newTokenId = ++currentIndex;
127         uint8 newTokenLevel = nftLevels[i];
128         // 1.set level 2.mint (can not change the order)
129         require(
130             newTokenLevel > 0 && newTokenLevel <= MAX_LEVEL,
131             "Invalid level"
132         );
133         levelOfToken[newTokenId] = newTokenLevel;
134         _safeMint(msg.sender, newTokenId);
135         emit InitialMinted(msg.sender, newTokenId, newTokenLevel);
136     }
137 }
```

Listing 2.1: HaloMembershipPass.sol

Impact Due to the use of rounding down in calculations, the fees paid by users are lower than expected.

Suggestion When calculating `payAmount`, rounding should be ensured to favor the protocol.

Feedback from the Project Currently, the decimals of ERC20 tokens are basically 6 or 18. When configuring `currencyAmountPerToken[payCurrency]`, we will also ensure that the value `currencyAmountPerToken[payCurrency]/SCALE_DECIMAL` will not produce decimals. Therefore, when calculating the payAmount, it will be divided and no decimals will be generated.

2.1.2 Lack of Check on the Number of Level5 and Level6 in Function initialMint()

Severity Medium

Status Confirmed

Introduced by Version 1

Description After obtaining `proof`, any user can mint NFTs with the function `initialMint()`, where the NFT's level is included in the `proof`. However, the function does not check whether the current number of level 5 and level 6 NFTs has reached the protocol's maximum limit. Specifically, after obtaining the correct proof, users can mint NFTs of any level through the function `initialMint()`. This may result in the quantity of level 5 and level 6 NFTs exceeding the upper limit. Since the function `upgradeMainProfile()` needs to verify whether the number of level 5 and level 6 NFTs has reached the limit.

The same issue also exists in the function `adminMint()`.

```
87   function initialMint(
88       bytes32[] calldata proof,
89       uint8[] calldata nftLevels,
90       uint256 discount,
91       address payCurrency
92   ) external payable callerIsUser nonReentrant whenNotPaused {
93       // Verify parameters
94       require(
95           initialMintMerkleRoot != 0x0 && block.timestamp >= startTimestamp,
96           "Not in initial mint period"
97       );
98       require(
99           proof.length > 0 &&
100              nftLevels.length > 0 &&
101              discount <= SCALE_DECIMAL,
102          "Invalid parameters"
103      );
104      require(isCurrencyEnabled[payCurrency], "Invalid currency");
105
106      require(!isMinted[msg.sender], "Already Minted");
107      //  Merkle verify
108      bytes32 leaf = keccak256(abi.encode(msg.sender, nftLevels, discount));
109      require(
110          MerkleProof.verify(proof, initialMintMerkleRoot, leaf),
111          "Invalid proof"
112      );
113      // Mark it minted
114      isMinted[msg.sender] = true;
115
116      // Charge the mint fee
117      uint256 nftAmount = nftLevels.length;
118      uint256 payAmount = (nftAmount *
119          currencyAmountPerToken[payCurrency] *
120          discount) / SCALE_DECIMAL;
121      _chargeMintFee(payCurrency, payAmount);
122
```

```
123     // Mint tokens
124     for (uint256 i = 0; i < nftAmount; i++) {
125         uint256 newTokenId = ++currentIndex;
126         uint8 newTokenLevel = nftLevels[i];
127         // 1.set level 2.mint (can not change the order)
128         require(
129             newTokenLevel > 0 && newTokenLevel <= MAX_LEVEL,
130             "Invalid level"
131         );
132         levelOfToken[newTokenId] = newTokenLevel;
133         _safeMint(msg.sender, newTokenId);
134         emit InitialMinted(msg.sender, newTokenId, newTokenLevel);
135     }
136 }
```

Listing 2.2: HaloMembershipPass.sol

```
258 function canUpgradeTo(uint8 toLevel) public view returns (bool) {
259     if (toLevel > 1 && toLevel < 5) return true;
260     if (toLevel == 5) {
261         return
262             totalSupply[5] <
263             (totalSupplyAll * level5UpperProportion) / SCALE_DECIMAL;
264     }
265     if (toLevel == 6) {
266         return
267             totalSupply[6] <
268             (totalSupplyAll * level6UpperProportion) / SCALE_DECIMAL;
269     }
270     return false;
271 }
```

Listing 2.3: HaloMembershipPass.sol

```
192 function upgradeMainProfile(
193     bytes32[] calldata proof,
194     uint256 campaignId,
195     uint8 toLevel
196 ) external nonReentrant whenNotPaused {
197     // Verify parameters
198     require(
199         proof.length > 0 &&
200             campaignMerkleRoot[campaignId] != 0x0 &&
201             toLevel <= MAX_LEVEL,
202         "Invalid parameters"
203     );
204
205     // Limit the maximum quantity
206     require(canUpgradeTo(toLevel), "Exceed the target proportion");
207
208     // the main profile nft is used by default
209     uint256 tokenId = userMainProfile[msg.sender];
210     require(
```

```
211             tokenId != 0 && ownerOf(tokenId) == msg.sender,
212             "Not user's main profile"
213         );
214         require(toLevel == levelOfToken[tokenId] + 1, "Invalid target level");
215
216         // Merkle verify
217         bytes32 leaf = keccak256(abi.encode(msg.sender, tokenId, toLevel));
218         require(
219             MerkleProof.verify(proof, campaignMerkleRoot[campaignId], leaf),
220             "Invalid proof"
221         );
222
223         //Upgrade:1.burn old token 2.mint new token
224         _burn(tokenId); // unbind main profile simultaneously
225         uint256 newTokenId = ++currentIndex;
226         levelOfToken[newTokenId] = toLevel;
227         _safeMint(msg.sender, newTokenId);
228         // bind the new token as main profile(because the old main profile has burnt)
229         userMainProfile[msg.sender] = newTokenId;
230         upgradedFrom[newTokenId] = tokenId;
231
232         emit NFTUpgraded(msg.sender, tokenId, newTokenId, toLevel);
233         emit MainProfileSet(msg.sender, newTokenId);
234     }
```

Listing 2.4: HaloMembershipPass.sol

```
309     function adminMint(
310         uint8[] calldata nftLevels,
311         address receiver
312     ) external onlyOwner {
313         uint256 amount = nftLevels.length;
314
315         for (uint256 i = 0; i < amount; i++) {
316             uint256 newTokenId = ++currentIndex;
317             uint8 newTokenLevel = nftLevels[i];
318             // 1.set level 2.mint (can not change the order)
319             require(
320                 newTokenLevel > 0 && newTokenLevel <= MAX_LEVEL,
321                 "Invalid level"
322             );
323             levelOfToken[newTokenId] = newTokenLevel;
324             _safeMint(receiver, newTokenId);
325             emit AdminMinted(receiver, newTokenId, newTokenLevel);
326         }
327     }
```

Listing 2.5: HaloMembershipPass.sol

Impact After the successful execution of the function `initialMint()`, the number of level 5 and level 6 NFTs may exceed the protocol's allowed maximum, affecting the normal logic of the contract.

Suggestion Add checks to ensure that the quantity of level 5 and level 6 NFTs remains within the limits allowed by the protocol.

Feedback from the Project When minting, there is no need to consider the maximum number limits of level5 and level6. Only needs to be considered when upgrading.

2.1.3 Incorrect Logic in Function _chargeMintFee()

Severity Low

Status Fixed in `Version2`

Introduced by `Version 1`

Description In the function `_chargeMintFee()`, if the passed `payAmount` is zero, the function will return directly without affecting the entire minting process of the NFTs. Therefore, if `currencyAmountPerToken[payCurrency]` is erroneously set to zero, or due to precision loss in calculations that results in a `payAmount` of zero, users can mint NFTs without paying any fees.

```
432   function _chargeMintFee(address payCurrency, uint256 payAmount) internal {
433       if (payAmount == 0) return;
434       if (payCurrency == address(0)) {
435           // native token
436           require(msg.value >= payAmount, "Insufficient payment amount");
437           payable(feeRecipient).transfer(msg.value);
438       } else {
439           // erc20 token
440           SafeERC20.safeTransferFrom(
441               IERC20(payCurrency),
442               msg.sender,
443               feeRecipient,
444               payAmount
445           );
446       }
447   }
```

Listing 2.6: HaloMembershipPass.sol

Impact Users can mint NFTs without paying any fees.

Suggestion Modify the check to revert when `payAmount` is zero.

Feedback from the Project In function `initialMint()` or function `publicMint()`, the discount may be 0, therefore `payAmount` may be 0, it is valid and don't need to be reverted. To prevent `currencyAmountPerToken[payCurrency]` from being erroneously set to 0, we have added the check `require(newPrice > 0, "Invalid price");` in function `enablePayCurrency()`.

2.1.4 Lack of Check on Parameters in Function setLevel5and6Proportion()

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `HaloMembershipPass` contract, the owner can set the upper limits for the quantity of level 5 and level 6 NFTs through the function `setLevel5and6Proportion()`. Since there is no parameter validation within the function, after updating, the current quantities of level 5 and level 6 NFTs may already exceed the limits. This could prevent other users' NFTs from being upgraded, which is incorrect.

```
342   function setLevel5and6Proportion(
343       uint256 newLevel5Proportion,
344       uint256 newLevel6Proportion
345   ) external onlyOwner {
346       require(
347           newLevel5Proportion <= 100 && newLevel6Proportion <= 100,
348           "Invalid proportion"
349       );
350       level5UpperProportion = newLevel5Proportion;
351       level6UpperProportion = newLevel6Proportion;
352   }
```

Listing 2.7: HaloMembershipPass.sol

Impact After the parameters are updated, the quantities of level 5 and level 6 may exceed the maximum allowed by the protocol, preventing other users' NFTs from upgrading normally.

Suggestion Add checks to ensure that after the parameters are updated, the quantities of level 5 and level 6 remain within the limits allowed by the protocol.

Feedback from the Project The situation mentioned by Impact are possible and acceptable. Of course, we will not make frequent adjustments and will notify users before making adjustments.

2.1.5 Centralization risk

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the `HaloMembershipPass` contract, the `owner` can mint NFTs of any quantity and any level through the function `adminMint()`. If the private key of the `owner` is leaked, it can cause severe damage to the protocol.

```
309   function adminMint(
310       uint8[] calldata nftLevels,
311       address receiver
312   ) external onlyOwner {
313       uint256 amount = nftLevels.length;
314
315       for (uint256 i = 0; i < amount; i++) {
316           uint256 newTokenId = ++currentIndex;
317           uint8 newTokenLevel = nftLevels[i];
318           // 1.set level 2.mint (can not change the order)
319           require(
320               newTokenLevel > 0 && newTokenLevel <= MAX_LEVEL,
```

```
321             "Invalid level"
322         );
323         levelOfToken[newTokenId] = newTokenLevel;
324         _safeMint(receiver, newTokenId);
325         emit AdminMinted(receiver, newTokenId, newTokenLevel);
326     }
327 }
```

Listing 2.8: HaloMembershipPass.sol

Impact If the owner's private key is leaked, it can infinitely mint NFTs of any level without any cost, causing losses to the protocol.

Suggestion Use multisig or DAO to manage the owner account.

Feedback from the Project After the formal contract is deployed, we will set the owner to the Safe MultiSig address.

2.1.6 Potential Overpayments by Users

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the function `_chargeMintFee()`, the check on Line 436 ensures that the user's `msg.value` is greater than or equal to `payAmount`. However, when the `msg.value` sent by the user exceeds `payAmount`, the function does not correctly handle the refunding of the excess amount, which is incorrect.

```
432 function _chargeMintFee(address payCurrency, uint256 payAmount) internal {
433     if (payAmount == 0) return;
434     if (payCurrency == address(0)) {
435         // native token
436         require(msg.value >= payAmount, "Insufficient payment amount");
437         payable(feeRecipient).transfer(msg.value);
438     } else {
439         // erc20 token
440         SafeERC20.safeTransferFrom(
441             IERC20(payCurrency),
442             msg.sender,
443             feeRecipient,
444             payAmount
445         );
446     }
447 }
```

Listing 2.9: HaloMembershipPass.sol

Impact When the native token sent by the user exceeds the `payAmount`, the excess amount is not refunded.

Suggestion Modify the function logic to ensure that if the user pays more native token than the `payAmount`, the excess amount should be refunded.

Feedback from the Project In order to prevent reentrancy attacks when ETH is refunded, we are not considering refunding directly. If an overpayment does occur, users can contact us and we will handle the refund manually.

2.1.7 Upgrade Failed due to Burning Large Quantities of Low-Level NFTs

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Any user can acquire a large number of low-level NFTs through the function `publicMint()` or by purchasing them on the market, and then they can continuously burn their low-level NFTs, reducing the `totalSupplyAll` count, resulting in the quantity of level 5 and level 6 NFTs reaching their maximum limit. In this case, it may become impossible for other users to normally upgrade their NFTs to level 5 or level 6.

```
249   function burn(uint256 tokenId) public {
250       require(
251           _isApprovedOrOwner(msg.sender, tokenId),
252           "Not token owner or approved"
253       );
254       _burn(tokenId);
255   }
```

Listing 2.10: HaloMembershipPass.sol

```
258   function canUpgradeTo(uint8 toLevel) public view returns (bool) {
259       if (toLevel > 1 && toLevel < 5) return true;
260       if (toLevel == 5) {
261           return
262               totalSupply[5] <
263               (totalSupplyAll * level5UpperProportion) / SCALE_DECIMAL;
264       }
265       if (toLevel == 6) {
266           return
267               totalSupply[6] <
268               (totalSupplyAll * level6UpperProportion) / SCALE_DECIMAL;
269       }
270       return false;
271   }
```

Listing 2.11: HaloMembershipPass.sol

Impact Users may be unable to successfully upgrade their NFTs as expected.

Suggestion Remove the function `burn()` or ensure the cost for users to acquire a large number of low-level NFTs is sufficiently high.

2.2 Additional Recommendation

2.2.1 Incorrect Method of Transferring Native Tokens

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `_chargeMintFee()`, the method `transfer()` is used to send native tokens, which has gas limitations compared to the method `call()`. If the `feeRecipient` is a contract account, it may result in DoS.

```
432   function _chargeMintFee(address payCurrency, uint256 payAmount) internal {
433       if (payAmount == 0) return;
434       if (payCurrency == address(0)) {
435           // native token
436           require(msg.value >= payAmount, "Insufficient payment amount");
437           payable(feeRecipient).transfer(msg.value);
438       } else {
439           // erc20 token
440           SafeERC20.safeTransferFrom(
441               IERC20(payCurrency),
442               msg.sender,
443               feeRecipient,
444               payAmount
445           );
446       }
447   }
```

Listing 2.12: HaloMembershipPass.sol

Suggestion Replace `transfer()` with the function `sendValue()` from the OpenZeppelin Library.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS