BLOCKSEC

Security Audit

Report for Halo-token-earn-contract and HaloMembershipPass

Date: January 2, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Halo
Target	Halo-token-earn-contract and HaloMembershipPass

Version History

Version	Date	Description
1.0	January 2, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-01-02 12:34:49+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

This audit focuses on the code repositories of the halo-token-earn-contract [1] and HaloMembershipPass.sol [2] 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 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
halo-token-earn-contract	`Version 1`	`05733631f676529f3095b75bbdbd8289cce6a8bb`
	`Version 2`	`c258fbfca83de43daaef32606417838132150e72`
HaloMembershipPass.sol	`Version 1`	`94fc54ddf8aae66ce1d3a6e82f28dc884ef6b9f8`
	`Version 2`	`66ef3c6e727144c6d5707c404ee96fdfd7315ec6`

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/halowalletdev/halo-token-earn-contract/tree/main/contracts

[2] https://github.com/halowalletdev/halo-membership-pass/blob/main/contracts/HaloMembershipPass.sol

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 and Common Weakness Enumeration. 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.

Chapter 2 Findings

In total, we found **four** potential security issues. Besides, we have **three** recommendations and **three** notes.

- High Risk: 1
- Low Risk: 3
- Recommendation: 3
- Note: 3

ID	Severity	Description	Category	Status
1	Low	Potential loss of influencer airdrop in function `setInfluencerInfos()`	DeFi Security	Confirmed
2	Low	Configuration overwrites and lack of validations in function `setAirdropDetail()`	DeFi Security	Confirmed
3	Low	Potential incorrect reward distribution in function `updateRewardRate()`	DeFi Security	Confirmed
4	High	Reuse of AdminSig enables upgrading multiple NFTs of users	DeFi Security	Fixed
5	-	Lack of non-zero check for key parameters	Recommendation	Confirmed
6	-	Lack of comparison check in function `setJustClaimPct()`	Recommendation	Confirmed
7	-	Lack of check in function `setClaimStartAt()`	Recommendation	Confirmed
8	-	Potential centralization risk	Note	-
9	-	HGP burn verification reliance on off-chain mechanisms	Note	-
10	-	Potential unavailability of `claimRewardsAndStake()` function due to `StakeToken` and `RewardToken` inconsistency	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential loss of influencer airdrop in function `setInfluencerInfos()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The function `setInfluencerInfos()` allows the privileged owner to update the amount of tokens claimable for influencers. According to the design, once the airdrop is open, influencers should be able to claim their tokens at any time. However, if the owner invokes the function to update values before the influencers claimed their tokens, the unclaimed token amounts will be overwritten with the new values, which is incorrect.

```
283 function setInfluencerInfos(
284   address[] calldata influencers,
285   uint256[] calldata amounts
286) external onlyOwner {
287   address influencer;
288   uint256 amount;
289   for (uint256 i = 0; i < influencers.length; i++) {
290       influencer = influencers[i];
291       amount = amounts[i];
292       influencerClaimableAmt[influencer] = amount;
293   }
294}
```

Listing 2.1: contracts/HaloAirdrop.sol

Impact The influencers may lose unclaimed tokens.

Suggestion Revise the logic to accumulate the claimable token amount when updating instead of overwriting.

Feedback from the project The number of influencers is fixed at approximately 60 and will be configured once before the airdrop begins. Under normal circumstances, each address will be configured only once, and the token amounts will remain unchanged thereafter.

2.1.2 Configuration overwrites and lack of validations in function `setAirdropDetail()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The `setAirdropDetail()` function allows the privileged owner to configure the `merkle tree root`, the immediate claimable token percentage, and the number of phases required to fully unlock the remaining tokens for `Halo Membership Pass (HMP)` and `Halo Genesis Pass (HGP)` holders. However, this function permits the owner to invoke it repeatedly, overwriting previously set configurations, which contradicts the design intent stating that "there is no time limit for holders to claim." Moreover, the function does not ensure that the `merkle tree roots` for these two airdrops are distinct, which is also incorrect.

```
300 function setAirdropDetail(
301   bytes32 root_,
302   uint256 imdClaimPct_,
303   uint256 totalUnlockPhases_,
304   bool isMP
305 ) external onlyOwner {
306   require(imdClaimPct_ <= 100, "INV_ARG");
307   if (isMP) {
308       airdropMP.root = root_;
309       airdropMP.imdClaimPct = imdClaimPct_;
310       airdropMP.totalUnlockPhases = totalUnlockPhases_;
311   } else {
```

```
312         airdropGP.root = root_;
313         airdropGP.imdClaimPct = imdClaimPct_;
314         airdropGP.totalUnlockPhases = totalUnlockPhases_;
315     }
316 }
```

Listing 2.2: contracts/HaloAirdrop.sol

Impact This issue risks misconfiguration, causing inconsistencies and unexpected results.

Suggestion Add necessary checks and restrict the function to be invoked only once to prevent the configured parameters from being overwritten.

Feedback from the project In order to prevent configuration errors from being unable to be modified, we will not add stricter verification. Instead, we manually check the parameters for correctness.

2.1.3 Potential incorrect reward distribution in function `updateRewardRate()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `HaloStakeVault` contract, users can deposit `Halo` tokens to earn rewards. The rewards are calculated based on the `rewardRatePerBlock` and staking time. However, the privileged owner is allowed to dynamically update the `rewardRatePerBlock` via the function `updateRewardRate()`. If `rewardRatePerBlock` is updated during the user's staking period, their rewards may not be distributed correctly, as new `rewardRatePerBlock` can be incorrectly applied to the earlier time intervals.

```
1185 function updateRewardRate(
1186   uint256 newRatePerBlock_,
1187   bool _withUpdate
1188) external onlyOwner {
1189   // whether check 0
1190   // require(newRatePerBlock_ > 0, "INV_RATE");
1191   if (_withUpdate) {
1192       updatePool();
1193   }
1194   rewardRatePerBlock = newRatePerBlock_;
1195   emit RewardRateChanged(newRatePerBlock_);
1196}
```

Listing 2.3: contracts/HaloStakeVault.sol

```
110 function updatePool() public {
111     if (block.number <= poolInfo.lastRewardBlock) {
112         return;
113     }
114     if (poolInfo.totalStaked > 0) {
115         uint256 multiplier = block.number - poolInfo.lastRewardBlock;
116         uint256 haloReward = multiplier * rewardRatePerBlock;
```

```
117             poolInfo.accHaloPerShare += Math.mulDiv(
118                 haloReward,
119                 ACC_PRECISION,
120                 poolInfo.totalStaked
121             );
122         }
123         poolInfo.lastRewardBlock = block.number;
124         // event
125         emit UpdatePool(
126             poolInfo.lastRewardBlock,
127             poolInfo.totalStaked,
128             poolInfo.accHaloPerShare
129         );
130     }
```

Listing 2.4: contracts/HaloStakeVault.sol

Impact The user's reward is calculated with the new `rewardRatePerBlock`, which is incorrect.

Suggestion Revise the logic to ensure that the function `updatePool()` must be invoked when updating `rewardRatePerBlock`.

Feedback from the project This method can only be called by the `owner`. By default, the parameter `_withUpdate` will be set to true.

2.1.4 Reuse of AdminSig enables upgrading multiple NFTs of users

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The user can upgrade the level of a specified `NFT` through the function `upgradeMainProfileWithToken()`, provided that the `NFT` is the user's current `userMainProfile` and the user has obtained the protocol admin's signature. However, the function's signature validation does not include the `NFT`'s `token_id`. In this case, users can repeatedly replay the signature before it expires to upgrade their other `NFT`s, as long as the `NFT`s meet the level requirement specified in the signature, which is incorrect.

```
241 function upgradeMainProfileWithToken(
242     uint8 toLevel,
243     address payCurrency,
244     uint256 payAmount,
245     uint256 sigExpiredAt,
246     bytes calldata adminSig
247 ) external payable nonReentrant whenNotPaused {
248     // Verify parameters
249     require(adminSigner != address(0), "Invalid signer");
250     require(
251         adminSig.length > 0 &&
252             sigExpiredAt > block.timestamp &&
253             toLevel <= MAX_LEVEL,
254         "Invalid parameters"
```

```
255     );
256     require(isCurrencyEnabled[payCurrency], "Invalid currency");
257     require(
258         payAmount >= minPayAmtToUpgrade[toLevel][payCurrency],
259         "Invalid amount"
260     );
261     // Verify signature
262     require(
263         verifyAdminSig(
264             keccak256(
265                 abi.encode(
266                     msg.sender,
267                     toLevel,
268                     payCurrency,
269                     payAmount,
270                     sigExpiredAt
271                 )
272             ),
273             adminSig
274         ),
275         "Invalid signature"
276     );
277
278
279     // Limit the maximum quantity
280     require(canUpgradeTo(toLevel), "Exceed the target proportion");
281
282
283     // the main profile nft is used by default
284     uint256 tokenId = userMainProfile[msg.sender];
285     require(
286         tokenId != 0 && ownerOf(tokenId) == msg.sender,
287         "Not user's main profile"
288     );
289
290
291     require(toLevel == levelOfToken[tokenId] + 1, "Invalid target level");
292     // Charge the mint fee
293     _chargeMintFee(payCurrency, payAmount);
294
295
296     // Upgrade:1.burn old token 2.mint new token
297     _burn(tokenId); // unbind main profile simultaneously
298     uint256 newTokenId = ++currentIndex;
299     levelOfToken[newTokenId] = toLevel;
300     _safeMint(msg.sender, newTokenId);
301     // bind the new token as main profile(because the old main profile has burnt)
302     userMainProfile[msg.sender] = newTokenId;
303     upgradedFrom[newTokenId] = tokenId;
304
305
306     emit NFTUpgraded(msg.sender, tokenId, newTokenId, toLevel);
307     emit MainProfileSet(msg.sender, newTokenId);
```

```
308   }
```

Listing 2.5: contracts/HaloMembershipPass.sol

```
306 function bindMainProfile(uint256 tokenId) external {
307   require(ownerOf(tokenId) == msg.sender, "Not token owner");
308   userMainProfile[msg.sender] = tokenId;
309   emit MainProfileSet(msg.sender, tokenId);
310}
```

Listing 2.6: contracts/HaloMembershipPass.sol

Impact The user can upgrade the levels of multiple NFTs with a single signature.

Suggestion Revise the logic, incorporating the `token_id` into the signature validation process to ensure that a single signature is only valid for one specific `token_id`.

2.2 Additional Recommendation

2.2.1 Lack of comparison check in function `setJustClaimPct()`

Status Confirmed

Introduced by Version 1

Description In the `claimOrLockForMP()` function, users can select from two claiming modes: direct and partial with subsequent locking. In direct mode, users receive a predefined percentage (i.e., `justClaimPct`) of tokens immediately, while the remaining tokens are transferred to the treasury. In partial mode, users claim an immediate percentage (i.e., `imdClaimPct`) with the rest being locked and gradually unlocked over time for future claims. Ideally, the `justClaimPct` should be higher than the `imdClaimPct` to fairly compensate for the immediate loss in direct claiming, where users forfeit part of their airdrop. If `justClaimPct` is lower, users may prefer partial claiming for its higher upfront payout, disrupting the intended design.

```
322   function setJustClaimPct(uint256 newPct_) external onlyOwner {
323       require(newPct_ <= 100, "INV_ARG");
324       justClaimPct = newPct_;
325   }
```

Listing 2.7: contracts/HaloAirdrop.sol

```
70   function claimOrLockForMP(
71       bytes32[] calldata proof,
72       uint256 amount,
73       bool isLock
74   ) external nonReentrant whenNotPaused {
75       // verify parameters
76       require(
77           block.timestamp > claimStartAt && airdropMP.root != 0x0,
78           "NOT_START"
79       );
80       require(proof.length > 0 && amount > 0, "INV_PARAM");
```

```
81        require(!isClaimedMP[msg.sender], "HAS_CLAIMED");
82        // merkle verify
83        bytes32 leaf = keccak256(abi.encode(msg.sender, amount));
84        require(MerkleProof.verify(proof, airdropMP.root, leaf), "INV_PROOF");
85        // mark it claimed
86        isClaimedMP[msg.sender] = true;
87
88
89        if (isLock) {
90            // lock: claim part + lock others
91            uint256 toUserAmount = (amount * airdropMP.imdClaimPct) / 100;
92            SafeERC20.safeTransfer(IERC20(HALO), msg.sender, toUserAmount);
93            uint256 lockAmount = amount - toUserAmount;
94            userInfoMP[msg.sender] = UserLockInfo({
95                lockStartAt: block.timestamp,
96                totalAmount: lockAmount,
97                claimedAmount: 0
98            });
99            emit ClaimAndLock(
100               msg.sender,
101               toUserAmount,
102               lockAmount,
103               AIRDROP_FOR_MP
104           );
105       } else {
106           // just claim part
107           uint256 toUserAmount = (amount * justClaimPct) / 100;
108           uint256 toTreasuryAmount = amount - toUserAmount;
109           // transfer: address(this)-> 1. to user + 2. to treasury
110           SafeERC20.safeTransfer(IERC20(HALO), msg.sender, toUserAmount);
111           SafeERC20.safeTransfer(IERC20(HALO), treasury, toTreasuryAmount);
112           // event
113           emit ClaimOnlyForMP(msg.sender, toUserAmount, toTreasuryAmount);
114       }
115   }
```

Listing 2.8: contracts/HaloAirdrop.sol

Suggestion Revise the logic to ensure that `justClaimPct` is greater than `imdClaimPct`.

Feedback from the project Since `justClaimPct` and `imdClaimPct` are not configured synchronously, we will verify them manually.

2.2.2 Lack of non-zero check for key parameters

Status Confirmed

Introduced by `Version 1`

Description In the `HaloAirdrop`, `HaloSocialMining`, `HaloStakeVault`, and `HaloMembershipPass` contracts, some key parameters lack non-zero validation, which could lead to unexpected behaviors. Specifically, in the following code segment:

1. The `constructor()` function lacks zero address checks, which may lead to critical contract addresses being incorrectly initialized.

2. The `setAirdropDetail()` function lacks zero check for the `totalUnlockPhases_` parameter. If `totalUnlockPhases_` is set to 0, it will lead to a division by zero error when performing the division operation.

```
56  constructor(
57      address owner_,
58      IERC20 HALO_,
59      address treasury_,
60      uint256 claimStartAt_,
61      uint256 justClaimPct_
62  ) Ownable(owner_) {
63      HALO = HALO_;
64      treasury = treasury_;
65      claimStartAt = claimStartAt_;
66      justClaimPct = justClaimPct_;
67  }
```

Listing 2.9: contracts/HaloAirdrop.sol

```
25  constructor(
26      address owner_,
27      IERC20 HALO_,
28      address rewardVault_
29  ) Ownable(owner_) {
30      HALO = HALO_;
31      rewardVault = rewardVault_;
32  }
```

Listing 2.10: contracts/HaloSocialMining.sol

```
40  constructor(
41      address owner_,
42      IERC20 stakeToken_,
43      IERC20 rewardToken_,
44      uint256 cooldownSeconds_,
45      uint256 unstakeSeconds_,
46      uint256 rewardRatePerBlock_,
47      address rewardVault_,
48      uint256 startBlock // the block number when reward starts
49  ) Ownable(owner_) {
50      stakeToken = stakeToken_;
51      rewardToken = rewardToken_;
52      cooldownSeconds = cooldownSeconds_;
53      unstakeSeconds = unstakeSeconds_;
54      rewardRatePerBlock = rewardRatePerBlock_;
55      rewardVault = rewardVault_;
56      poolInfo = PoolInfo({
57          accHaloPerShare: 0,
58          lastRewardBlock: Math.max(startBlock, block.number),
59          totalStaked: 0
60      });
61      // for restake( when stakeToken=rewardToken )
62      rewardToken.approve(address(this), type(uint256).max);
```

```
63  }
```

Listing 2.11: contracts/HaloStakeVault.sol

```
67  function initialize(
68      string memory name_,
69      string memory symbol_,
70      address feeRecipient_,
71      uint256 level6UpperProportion_
72  ) public initializer {
73      feeRecipient = feeRecipient_;
74      level5UpperProportion = 100;
75      level6UpperProportion = level6UpperProportion_;
76
77
78      __ReentrancyGuard_init();
79      __Pausable_init();
80      __Ownable2Step_init();
81      __ERC721_init(name_, symbol_);
82  }
```

Listing 2.12: contracts/HaloMembershipPass.sol

```
300 function setAirdropDetail(
301     bytes32 root_,
302     uint256 imdClaimPct_,
303     uint256 totalUnlockPhases_,
304     bool isMP
305 ) external onlyOwner {
306     require(imdClaimPct_ <= 100, "INV_ARG");
307     if (isMP) {
308         airdropMP.root = root_;
309         airdropMP.imdClaimPct = imdClaimPct_;
310         airdropMP.totalUnlockPhases = totalUnlockPhases_;
311     } else {
312         airdropGP.root = root_;
313         airdropGP.imdClaimPct = imdClaimPct_;
314         airdropGP.totalUnlockPhases = totalUnlockPhases_;
315     }
316 }
```

Listing 2.13: contracts/HaloAirdrop.sol

```
200 function getUnlockInfo(
201     address user
202 )
203     public
204     view
205     returns (
206         uint256 unlockableAmtForMP,
207         uint256 unlockableAmtForGP,
208         uint256 nextUnlockTimeForMP,
209         uint256 nextUnlockTimeForGP
```

```
210         )
211     {
212         // for hmp
213         UserLockInfo memory userInfoForMP = userInfoMP[user];
214         if (userInfoForMP.lockStartAt > 0) {
215             // else: lockStartAt=0 ==> unlockableAmtForMP = 0, nextUnlockTimeForMP=0
216             uint256 currentPhases = (block.timestamp -
217                 userInfoForMP.lockStartAt) / DURATION_PER_PHASE;
218             uint256 maxUnlockPhases = Math.min(
219                 airdropMP.totalUnlockPhases,
220                 currentPhases
221             );
222             uint256 maxUnlockAmount = (maxUnlockPhases *
223                 userInfoForMP.totalAmount) / airdropMP.totalUnlockPhases;
224             unlockableAmtForMP = maxUnlockAmount - userInfoForMP.claimedAmount;
225             // next unlock time
226             if (currentPhases < airdropMP.totalUnlockPhases) {
227                 nextUnlockTimeForMP =
228                     userInfoForMP.lockStartAt +
229                     (currentPhases + 1) *
230                     DURATION_PER_PHASE;
231             }
232         }
233         // for gp
234         UserLockInfo memory userInfoForGP = userInfoGP[user];
235         if (userInfoForGP.lockStartAt > 0) {
236             uint256 currentPhases = (block.timestamp -
237                 userInfoForGP.lockStartAt) / DURATION_PER_PHASE;
238
239
240             uint256 maxUnlockPhases = Math.min(
241                 airdropGP.totalUnlockPhases,
242                 currentPhases
243             );
244             uint256 maxUnlockAmount = (maxUnlockPhases *
245                 userInfoForGP.totalAmount) / airdropGP.totalUnlockPhases;
246             unlockableAmtForGP = maxUnlockAmount - userInfoForGP.claimedAmount;
247             // next unlock time
248             if (currentPhases < airdropGP.totalUnlockPhases) {
249                 nextUnlockTimeForGP =
250                     userInfoForGP.lockStartAt +
251                     (currentPhases + 1) *
252                     DURATION_PER_PHASE;
253             }
254         }
255     }
```

Listing 2.14: contracts/HaloAirdrop.sol

Suggestion Add a check to ensure that key parameters are not zero.

Feedback from the project We will manually check the parameters in `constructor()`.

2.2.3 Lack of check in function `setClaimStartAt()`

Status Confirmed

Introduced by `Version 1`

Description In the `constructor()` function, the global variable `claimStartAt` lacks proper validation. Specifically, it should be greater than or equal to the current timestamp. Additionally, the extra implementation of the `setClaimStartAt()` function, which allows the owner to update `claimStartAt`, is unnecessary, as this variable only determines whether users can start claiming the airdrop.

```
56 constructor(
57   address owner_,
58   IERC20 HALO_,
59   address treasury_,
60   uint256 claimStartAt_,
61   uint256 justClaimPct_
62) Ownable(owner_) {
63   HALO = HALO_;
64   treasury = treasury_;
65   claimStartAt = claimStartAt_;
66   justClaimPct = justClaimPct_;
67}
```

Listing 2.15: contracts/HaloAirdrop.sol

```
296 function setClaimStartAt(uint256 newStartAt) external onlyOwner {
297   claimStartAt = newStartAt;
298}
```

Listing 2.16: contracts/HaloAirdrop.sol

Suggestion Add check to ensure that `claimStartAt` is greater than or equal to the current timestamp, and remove the redundant implementation.

Feedback from the project When deploying the contract, we will manually check it.

2.3 Note

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description In the current implementation, several privileged roles are set to govern and regulate the system-wide operations (e.g., parameter setting, pause/unpause). Additionally, the `owner` of the contract `HaloMembershipPass` can mint `NFT`s of any quantity and any level through the function `adminMint()`. If the private keys of these privileged roles are lost or maliciously exploited, it could potentially lead to losses for users.

2.3.2 HGP burn verification reliance on off-chain mechanisms

Introduced by Version 1

Description According to the documentation, users are required to burn the corresponding HGP tokens in the HGPBurn contract before claiming the airdrop. However, the HaloAirdrop contract does not have relevant implementation to verify whether these NFT tokens have been burned. This verification should be ensured off-chain, which is beyond the scope of our audit.

2.3.3 Potential unavailability of claimRewardsAndStake() function due to StakeToken and RewardToken inconsistency

Introduced by Version 1

Description In the HaloStakeVault contract, stakeToken and rewardToken are initialized as immutable variables in the constructor and are designed to be set only once. Separate contracts will be deployed for each unique combination of stakeToken and rewardToken. Additionally, only stake vaults where the stakeToken and rewardToken are the same support the functionality of restaking rewards.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS