#HashEx
BLOCKCHAIN SECURITY

Embers NFT

smart contracts
final audit report

March 2022

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn’t say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Embers team to perform an audit of the Embers and ERC721A contracts. The audit was conducted between 2022-03-09 and 2022-03-10.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code was available provided via gist with md5 sum of 5b8c2d1d70283cfc7cb338e38130a445 and qwertyalpha/ember-contracts-audit GitHub repository. The Embers contract was audited after 6cfb51d commit. The original ERC721A contract is available at @chiru-labs/ERC721A GitHub repository.

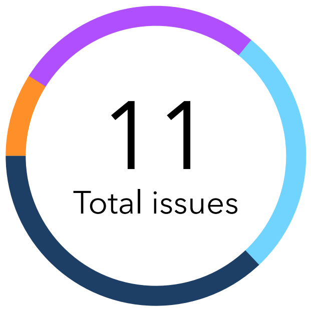
2.1 Summary

Project name	Embers NFT
URL	https://embersnft.com
Platform	Ethereum
Language	Solidity

2.2 Contracts

Name	Address
ERC721A	
Embers	

3. Found issues



●	High	1 (9%)
●	Medium	3 (27%)
●	Low	3 (27%)
●	Info	4 (37%)

C1. ERC721A

ID	Severity	Title	Status
C1-01	● Medium	Unsafe math	Open
C1-02	● Info	Only sequential minting is possible	Open
C1-03	● Info	Inconsistent comment	Open
C1-04	● Info	Revert without a reason	Open
C1-05	● Info	tokenURI() return data	Open

C2. Embers

ID	Severity	Title	Status
C2-01	● High	Reentrancy vulnerability	Open
C2-02	● Medium	Functions finishAuction may not finish the action	Open

C2-03	● Medium	Owner can change base token URI	ⓘ Open
C2-04	● Low	Lack of events	ⓘ Open
C2-05	● Low	Gas savings	ⓘ Open
C2-06	● Low	Excessive payment ethers are not returned to the buyer	ⓘ Open

4. Contracts

C1. ERC721A

Overview

ERC721 standard token, a fork of chiru-labs version with a customized token numeration (starting from 1 instead of 0). According to the documentation, the code is inspired by the ERC721Enumerable extension from the OpenZeppelin library, but with several gas optimizations.

Issues

C1-01 Unsafe math ● Medium ⓘ Open

The minting function uses `unchecked` math and therefore should have proper warnings about the `quantity` parameter. Any user-interacting functions with internal calls to `_mint()` must check the parameters for over- and underflow.

Recommendation

We recommend specifying this remark in the NatSpec description and in the project documentation. Any project that forks this code should limit the `quantity` parameter to some range.

C1-02 Only sequential minting is possible ● Info ⓘ Open

Anyone who wants to implement the random or a user-defined minting order would face difficulties with ERC721A and should probably use OpenZeppelin versions.

C1-03 Inconsistent comment ● Info ⓘ Open

The comment about minting order in L38 mentions that the default starting ID is 0, but the `_startTokenId()` function is modified in OgamiStudios version:

```
function _startTokenId() internal view virtual returns (uint256) {
  return 1;
}
```

C1-04 Revert without a reason ● Info ⓘ Open

Reentrancy protection revert at L409 should have a reason (or an error message):

```
if (_currentIndex != startTokenId) revert();
```

C1-05 tokenURI() return data ● Info ⓘ Open

The `tokenURI()` returned string is always ended with `tokenId` without an extension. `tokenURI()` returns zero length string if `_baseURI()` hasn't been set.

Assumes serials are sequentially minted starting at `_startTokenId()`.

Assumes that an owner cannot have more than `2**64 - 1` (max value of `uint64`) of supply.

Assumes that the maximum token id cannot exceed `2**256 - 1` (max value of `uint256`).

C2. Embers

Overview

The Embers contract is an ERC721 contract with a sale functionality. Users can buy Embers tokens for native currency via whitelisted purchase for a special price, or via a public sale.

Issues

C2-01 Reentrancy vulnerability ● High Open

The function `whitelistBuy()` is susceptible to a reentrancy attack. An attacker can mint up to the maximum supply number of tokens by using a callback in the `_safeMint()` call.

```
function whitelistBuy(bytes calldata signature) public payable { //@audit external
   require(!claimed[msg.sender], "ALREADY_CLAIMED");
   ...
   _safeMint(msg.sender, 1);
   claimed[msg.sender] = true;
 }
```

Recommendation

Switch L55 and L56 or use the ReentrantGuard contract from OpenZeppelin.

```
function whitelistBuy(bytes calldata signature) public payable { //@audit external
   require(!claimed[msg.sender], "ALREADY_CLAIMED");
   ...
   claimed[msg.sender] = true;
   _safeMint(msg.sender, 1);
 }
```

C2-02 Functions finishAuction may not finish the action ● Medium Open

The function `finishAuction()` sets the auction start price to zero but this does not guarantee that the auction will be finished.

```
    function finishAuction() external onlyOwner {
      embersConfig.auctionStartPrice = 0;
    }
```

C2-03 Owner can change base token URI ● Medium Open

The `_baseTokenURI` can be changed by the owner without any limitations. This may break the functionality of the contract as all the token links will be broken if a wrong base token URI is set.

C2-04 Lack of events ● Low Open

Functions `finishAuction()`, `setStartTime()`, `setBaseURI()` do not emit events. We recommend adding events to simplify the offchain tracking of changes.

C2-05 Gas savings ● Low Open

Multiple reads from the storage in the `getPrice()` function. `auctionConfig.startTime` should be read once to a local variable.

Functions `whitelistBuy()` and `auctionBuy()` can be declared external.

The `team[]` array should be removed. The `payees[]` array should be used instead.

The memory type of the `_PROVENANCE_HASH` parameter in the `setProvenanceHash()` functions should be set to `calldata`.

C2-06 Excessive payment ethers are not returned to the buyer ● Low Open

The functions `whitelistBuy()` and `auctionBuy()` check that the user sends at least the minimum amount to buy. If a user sends more than needed, the excess will not be returned back.

```
    function whitelistBuy(bytes calldata signature) public payable { //@audit external
      ...
      require(msg.value >= embersConfig.whitelistPrice, "NOT_ENOUGH_ETHER"); //@audit
exact price
      ...
    }
```

```
    function auctionBuy(uint256 quantity) public payable { //@audit external
      ...
      require(msg.value >= cost, "NOT_ENOUGH_ETHER");
      ...
    }
```

Recommendation

Return back the sent excess of the native currency.

5. Conclusion

1 high and 3 medium severity issues were found during the audit. The ERC721A contracts are well tested (in the @chiru-labs/ERC721A repository). ERC721A contract conforms to the ERC721 token standard.

This audit includes recommendations on code improvement and preventing potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

8. Slither ERC721 check

```
# Check ERC721A

## Check functions
[+] balanceOf(address) is present
        [+] balanceOf(address) -> () (correct return value)
        [+] balanceOf(address) is view
[+] ownerOf(uint256) is present
        [+] ownerOf(uint256) -> () (correct return value)
        [+] ownerOf(uint256) is view
[+] safeTransferFrom(address,address,uint256,bytes) is present
        [+] safeTransferFrom(address,address,uint256,bytes) -> () (correct return type)
        [+] Transfer(address,address,uint256) is emitted
[+] safeTransferFrom(address,address,uint256) is present
        [+] safeTransferFrom(address,address,uint256) -> () (correct return type)
        [+] Transfer(address,address,uint256) is emitted
[+] transferFrom(address,address,uint256) is present
        [+] transferFrom(address,address,uint256) -> () (correct return type)
        [+] Transfer(address,address,uint256) is emitted
[+] approve(address,uint256) is present
        [+] approve(address,uint256) -> () (correct return type)
        [+] Approval(address,address,uint256) is emitted
[+] setApprovalForAll(address,bool) is present
        [+] setApprovalForAll(address,bool) -> () (correct return type)
        [+] ApprovalForAll(address,address,bool) is emitted
[+] getApproved(uint256) is present
        [+] getApproved(uint256) -> () (correct return value)
        [+] getApproved(uint256) is view
[+] isApprovedForAll(address,address) is present
        [+] isApprovedForAll(address,address) -> () (correct return value)
        [+] isApprovedForAll(address,address) is view
[+] supportsInterface(bytes4) is present
        [+] supportsInterface(bytes4) -> () (correct return value)
        [+] supportsInterface(bytes4) is view
[+] name() is present
        [+] name() -> () (correct return value)
        [+] name() is view
[+] symbol() is present
        [+] symbol() -> () (correct return value)
[+] tokenURI(uint256) is present
```

```
        [+] tokenURI(uint256) -> () (correct return value)

## Check events
[+] Transfer(address,address,uint256) is present
        [+] parameter 0 is indexed
        [+] parameter 1 is indexed
        [+] parameter 2 is indexed
[+] Approval(address,address,uint256) is present
        [+] parameter 0 is indexed
        [+] parameter 1 is indexed
        [+] parameter 2 is indexed
[+] ApprovalForAll(address,address,bool) is present
        [+] parameter 0 is indexed
        [+] parameter 1 is indexed
```

contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx
BLOCKCHAIN SECURITY