Ethernity Chain

Smart Contract Audit Report

NFT MarketPlace

IMMUNE BYTES

Audits

April 26, 2021

IMMUNE BYTES

Introduction 3
- About Ethernity Chain 3
- About ImmuneBytes 3

Audit Process & Methodology 4

Audit Details 4

Documentation Details 4

Audit Goals 5

Security Level References 5
- High severity issues 6
- Medium severity issues 6
- Low severity issues 10

Recommendations 12

Automated Test Results 13

Concluding Remarks 14

Disclaimer 14

Introduction

1. About Ethernity Chain

Ethernity is a Decentralized Application (DAPP) Platform that allows artists to create and auction artwork inspired and backed by celebrities for charity.
The concept behind Ethernity is mutually beneficial for all actors involved:

1. **Public Figure:** by making it easier to create, store, back, and sell the artworks.
2. **Charity:** by getting 100% of the first sale proceeds (minus exchange fees). And the auction format maximizes the artwork value (increasing the charity's benefits) without the need of a promoter, leveraging the emotions that a bidding war involves.
3. **Collector:** by providing them with an easy, democratized platform to bid on these pieces of authentic digital art where they can thereafter take bids and auction their acquired artwork.

With ERN tokens collectors can acquire Ethernity's exclusive authenticated NFTs as payment method and also yield farming rewards. Part of the sales proceeds go to charity.

Visit https://ethernity.io/ to know more.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 15+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: Ethernity
- Languages: Solidity(Smart contract), Javascript(Unit Testing)
- Github commit hash for audit: 49d0433b94b38d9efad301a90709ecdf544b5755

Documentation Details

No documentation was provided by the Ethernity team for the purpose of conducting the audit.

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level References

Every issue in this report was assigned a severity level from the following:

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	1	7	5
Closed	-	-	-

High severity issues

1. **Multiplication is being performed on the result of Division**
   Line no - 208
   Explanation:
   The **buyByErn function** in the Collections.sol contract performs multiplication on the result of a Division.

```
205         } else {
206             amount = collection[id].ern;
207             uint256 discount = collection[id].discount;
208             uint256 discountAmount = (collection[id].ern / 100) * discount;
209             amount = amount - discountAmount;
210             seller = collection[id].seller;
211         }
212
```

   Integer Divisions in Solidity might truncate. Moreover, performing division before multiplication might lead to loss of precision.

   The following functions involve division before multiplication in the mentioned lines:
   - **buyByErn** at 208

   Recommendation:
   Solidity doesn't encourage arithmetic operations that involve division before multiplication. Therefore the above-mentioned function should be checked once and redesigned if they do not lead to expected results.

Medium severity issues

1. **Contract State Variables are being updated after External Calls.**
   Line no - 213-218, 256-257
   Explanation:
   The Collections contract includes quite a few functions that update some of the very imperative state variables of the contract after the external calls are being made.

   An external call within a function technically shifts the control flow of the contract to another contract for a particular period of time. Therefore, as per the Solidity Guidelines, any modification of the state variables in the base contract must be performed before executing the external call.
   Updating state variables after an external call might lead to a potential re-entrancy scenario.

The following function in the contract update the state variables after making an external call:

- **buyByErn** at Line 216 and 218.
- **buyByStones** at Line 257

```
212
213         distributeTokens(msg.sender, address(this), ernToken, amount, seller);
214
215         transferFrom(address(this), msg.sender, id, 1, "");
216         collection[id].seller = msg.sender;
217
218         sold += 1;
219
```

Recommendation:
Modification of any State Variables must be performed before making an external call.

2. **Return Value of an External Call is Not used Effectively**
   Line no - 192,256, 289-315,
   Explanation:
   The external calls made in the above-mentioned lines do return a boolean value that indicates whether or not the external call made was successful.
   These boolean return values can be used in the function as a check to ensure that the further execution of the function is only allowed if the external is successfully made.

```
191     function withdrawAmount(IERC20 token) external onlyOwner {
192         token.transfer(msg.sender, token.balanceOf(address(this)));
193     }
```

   However, the Collections contract never uses these return values throughout the contract.

   Recommendation:
   Effective use of all the return values from external calls must be ensured within the contract.

3. **Violation of Check_Effects_Interaction Pattern**
   Line no - 167, 220, 259, 152, 122
   Explanation:
   As per the Check_Effects_Interaction Pattern in Solidity, external calls should be made at the very end of the function and event emission, as well as any state variable modification, must be done before the external call is made.

However, the following functions in the Collections contract emit events after the external call has been made at the line number mentioned above:

- **addCard**
- **buyByErn**
- **buyByStones**
- **cancelSale**
- **cardTransfer**

Recommendation:
Check Effects Interaction Pattern must be followed while implementing external calls in a function

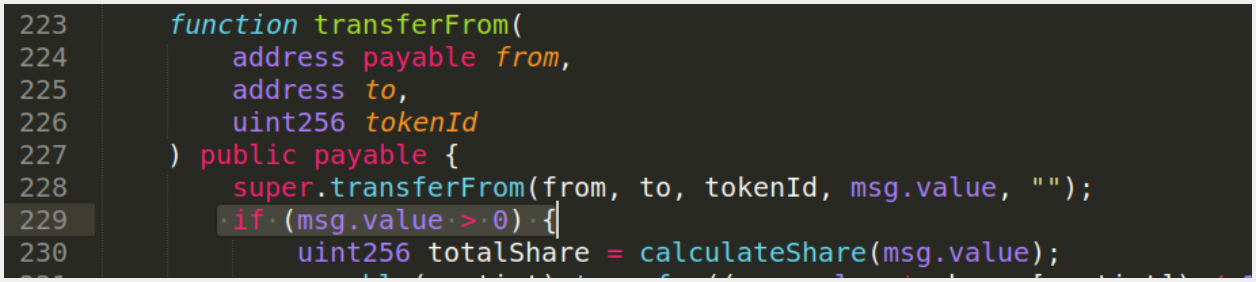
4. **Loops are extremely costly**
   Line no: 136, 182
   Description:
   The **for loops**, at the above-mentioned lines, in the contract includes state variables like **.length** of a non-memory array, in the condition of the for loops.
   As a result, these state variables consume a lot more extra gas for every iteration of the for loop.
   The following functions include such loops at the above-mentioned lines:
   - **cardTransferBatch function**
   - **addCardBatch**

   Recommendation:
   It's quite effective to use a local variable instead of a state variable like .length in a loop.
   For instance,

```
uint256 local_variable = ids.length;
for (uint24 i = 0; i <local_variable; i++) {
        addCard(sellers[i], ids[i], erns[i], stones[i], discounts[i]);
    }
```

5. **transferFrom function should include “require” statement instead of IF-Else Statement**
   Line no: 320-322
   Explanation
   The **transferFrom** function includes an **if statement** at the very beginning of the function to check whether or not the **msg.value** sent while calling this function, is greater than **ZERO**.
   Most importantly, the function body is only executed if this **IF statement** holds true.

In order to check for such **strict validations** in a function, **require statements** are more preferable and effective solidity. While it helps in gas optimizations it also enhances the readability of the code.



```
223     function transferFrom(
224         address payable from,
225         address to,
226         uint256 tokenId
227     ) public payable {
228         super.transferFrom(from, to, tokenId, msg.value, "");
229         if (msg.value > 0) {
230             uint256 totalShare = calculateShare(msg.value);
```

Recommendation:
Use **require statement** instead of **IF statement** in the above-mentioned function line.
For instance,
require(msg.value > 0, "Error MSG: msg.value should be more than ZERO");

6. **addShares function does not include Zero Address Validation**
   Line no: 60
   Explanation:
   The **addShares** function initializes some of the most imperative state variables in the Collections.sol contract and assigns their respective share amount.

   However, during the automated testing of the contract, it was found that the function doesn't implement any Zero Address Validation Check to ensure that no zero address is passed while calling this function.

   Recommendation:
   Since the **addFunction** initializes imperative addresses and assigns share amount to those addresses, it is quite crucial to implement zero address checks and ensure that only valid addresses are updated while calling this function.

7. **State Variables are being updated after External Calls. Violation of Check-Effects-Interaction Pattern**
   Line no - 26-29
   Explanation:
   The NFTFactory contract includes a function that updates a state variable after making an external call.

Moreover, as per the Check_Effects_Interaction Pattern in Solidity, external calls should be made at the very end of the function and event emission, as well as any state variable modification or event emission, must be done before the external call is made. However, the following function in the Collections contract updates a state variable and emits events after the external call has been made at the line number mentioned above:

- **createCollection** function at Line 27-29

```
19      function createCollection(
20          string memory uri,
21          uint256 _id,
22          address _toAddress
23      ) public onlyOwner returns (Collections) {
24          require(_ids.add(_id), "id should be unique");
25          Collections child = new Collections(uri, _toAddress);
26          child.transferOwnership(owner());
27          children.push(child);
28
29          emit CollectionCreated(owner(), address(child));
30          return child;
31      }
```

Recommendation:
Modification of any State Variables must be performed before making an external call. Check Effects Interaction Pattern must be followed while implementing external calls in a function

Low severity issues

1. **External Visibility should be preferred**
   Explanation:
   Those functions that are never called throughout the contract should be marked as **external** visibility instead of **public** visibility.
   This will effectively result in Gas Optimization as well.

   Therefore, the following function must be marked as **external** within the contract:
   - **addShares**
   - **addCardBatch**
   - **transferFrom**

   Recommendation:
   External Visibility should be preferred for the above-mentioned functions.

2. **Comparison to boolean Constant**
   Line no: 146
   Description:
   Boolean constants can directly be used in conditional statements or require statements. Therefore, it's not considered a better practice to explicitly use **TRUE or FALSE** in the **require** statements.

```
141     function cancelSale(uint256 id) external {
142         require(
143             collection[id].seller == msg.sender,
144             "this card is not yours to cancel the sale"
145         );
146         require(winners[id].transferred == false, "Card already transferred");
147         require(winners[id].seller != address(0), "There is no sale");
148
```

   Recommendation:
   The equality to boolean constants must be removed from the above-mentioned line.

3. **Functions with similar names should be avoided**
   Line no - 223
   Description:
   The Collections.sol contract includes two with exactly similar names.

   Since every function has different behavior, it is considered a better practice to avoid similar names for 2 different functions to eliminate any dilemma and enhance the readability of the code.

   Mentioned below are the function(s) with similar names but different behavior and arguments:
   - **transferFrom - Collections.sol contract #Line223**
   - **transferFrom - ERC1155.sol contract #Line148**

   Recommended:
   It is recommended to avoid using a similar name for different functions.

4. **11 Order of layout**
   Description:
   1. As per the Solidity Style Guide, the order of elements and statements should be according to the following layout:
   2. Pragma statements
   3. Import statements
   4. Interfaces

5. Libraries
6. Contracts
   Inside each contract, library or interface, use the following order:
1. Type declarations
2. State variables
3. Events
4. Functions

The following documentation links can be used as a reference to understand the correct order: - https://solidity.readthedocs.io/en/v0.8.0/style-guide.html#order-of-layout https://solidity.readthedocs.io/en/v0.8.0/style-guide.html#order-of-functions

5. **External Visibility should be preferred**
   Explanation:
   Those functions that are never called throughout the contract should be marked as **external** visibility instead of **public** visibility.
   This will effectively result in Gas Optimization as well.
   Therefore, the following function must be marked as external within the contract:
   - **createCollection**

   Recommendation:
   External Visibility should be preferred for the above-mentioned functions.

Recommendations

1. **Coding Style Issues**
   Code readability of a Smart Contract is largely influenced by the Coding Style issues and in some specific scenarios may lead to bugs in the future.
   Therefore, it is highly recommended to fix the issues like naming convention, indentation, and code layout issues in a smart contract.

2. **NatSpec Annotations must be included**
   Description:
   The smart contracts do not include the NatSpec annotations adequately.

   Recommendation:
   Cover by NatSpec all Contract methods.

Automated Test Results

```
Reentrancy in Collections.buyByErn(uint256) (FlatCollections.sol#1695-1721):
        External calls:
        - distributeTokens(msg.sender,address(this),ernToken,amount,seller) (FlatCollections.sol#1713)
                - token.transferFrom(from,to,amount) (FlatCollections.sol#1789)
                - token.transfer(_artist,(amount * shares[_artist]) / 100) (FlatCollections.sol#1792)
                - token.transfer(_celebrity,(amount * shares[_celebrity]) / 100) (FlatCollections.sol#1796)
                - token.transfer(_agent,(amount * shares[_agent]) / 100) (FlatCollections.sol#1800)
                - token.transfer(_charityOne,(amount * shares[_charityOne]) / 100) (FlatCollections.sol#1804)
                - token.transfer(_charityTwo,(amount * shares[_charityTwo]) / 100) (FlatCollections.sol#1808)
                - token.transfer(_toAddress,(amount * shares[_toAddress]) / 100) (FlatCollections.sol#1812)
                - token.transfer(seller,amount - totalShare) (FlatCollections.sol#1815)
        - transferFrom(address(this),msg.sender,id,1,) (FlatCollections.sol#1715)
                - IERC1155Receiver(to).onERC1155Received(operator,from,id,amount,data) (FlatCollections.sol#1033-1051)
        State variables written after the call(s):
        - collection[id].seller = msg.sender (FlatCollections.sol#1716)
```

```
Collections.constructor(string,address).toAddress (FlatCollections.sol#1556) lacks a zero-check on :
                - _toAddress = toAddress (FlatCollections.sol#1557)
Collections.addShares(address,uint256,address,uint256,address,uint256,address,uint256,address,uint256,uint256).artist (FlatCollections.sol#1561) lack
 a zero-check on :
                - _artist = artist (FlatCollections.sol#1573)
Collections.addShares(address,uint256,address,uint256,address,uint256,address,uint256,address,uint256,uint256).celebrity (FlatCollections.sol#1563) l
cks a zero-check on :
                - _celebrity = celebrity (FlatCollections.sol#1574)
Collections.addShares(address,uint256,address,uint256,address,uint256,address,uint256,address,uint256,uint256).agent (FlatCollections.sol#1565) lacks
a zero-check on :
                - _agent = agent (FlatCollections.sol#1575)
Collections.addShares(address,uint256,address,uint256,address,uint256,address,uint256,address,uint256,uint256).charityOne (FlatCollections.sol#1567)
acks a zero-check on :
                - _charityOne = charityOne (FlatCollections.sol#1576)
Collections.addShares(address,uint256,address,uint256,address,uint256,address,uint256,address,uint256,uint256).charityTwo (FlatCollections.sol#1569)
acks a zero-check on :
                - _charityTwo = charityTwo (FlatCollections.sol#1577)
Collections.transferFrom(address,address,uint256).from (FlatCollections.sol#1724) lacks a zero-check on :
                - from.transfer(msg.value - totalShare) (FlatCollections.sol#1745)
```

```
Reentrancy in Collections.cancelSale(uint256) (FlatCollections.sol#1641-1653):
        External calls:
        - transferFrom(address(this),msg.sender,id,1,) (FlatCollections.sol#1650)
                - IERC1155Receiver(to).onERC1155Received(operator,from,id,amount,data) (FlatCollections.sol#1033-1051)
        Event emitted after the call(s):
        - SaleCancelled(msg.sender,id) (FlatCollections.sol#1652)
Reentrancy in Collections.cardTransfer(uint256,uint256,address,address) (FlatCollections.sol#1608-1623):
        External calls:
        - transferFrom(msg.sender,address(this),id,1,) (FlatCollections.sol#1620)
                - IERC1155Receiver(to).onERC1155Received(operator,from,id,amount,data) (FlatCollections.sol#1033-1051)
        Event emitted after the call(s):
        - WinnerAdded(winner,amount,id) (FlatCollections.sol#1622)
```

```
Reentrancy in NFTFactory.createCollection(string,uint256,address) (FlatFactory.sol#1908-1920):
        External calls:
        - child.transferOwnership(owner()) (FlatFactory.sol#1915)
        State variables written after the call(s):
        - children.push(child) (FlatFactory.sol#1916)
```

Concluding Remarks

While conducting the audits of Ethernity NFT smart contracts, it was observed that the contracts contain High, Medium, and Low severity issues, along with several areas of recommendations.

Our auditors suggest that High, Medium, Low severity issues should be resolved by Ethernity developers. Resolving the areas of recommendations are up to the team's discretion. The recommendations given will improve the operations of the smart contract.

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the Ethernity platform or its product neither this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes Pvt Ltd.