MYNT

Smart Contract Audit Report

September 23, 2021

Introduction

1. About MYNT

-- Waiting for feedback from the team--

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 75+ 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.

Documentation Details

The MYNT team has provided the following doc for the purpose of audit:

1. --

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: MYNT
- Contracts Names: Drrt.sol
- Languages: Solidity(Smart contract), Typescript (Unit Testing)
- Github commit hash for audit: b8668164021d077d9d6fdcaafae62c5c8971d3ac
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck

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:

Admin/Owner Privileges can be misused either intentionally or unintentionally.

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	-	2
Closed	-	-	-

Admin/Owner Privileges

--

High severity issues

1. **mint() function logic is broken. Allows anyone to mint multiple tokens by paying for only one. Line no - 125 to 148**

   Explanation:
   As per the current design of the mint function in the contract, the function performs check at line **139** to ensure that the ether value passed is greater than the amount required to buy the desired tokens.

```
138
139         require(msg.value >= tokenPrice * tokenAmount, "Ether value sent is not correct");
140
```

   It then starts the token minting by calling the **_safeMint()** function. Moreover, once the required amount of token is minted, it also includes a **transfer()** function to return back the extra ETH amount(**msg.value**) that the caller of the function might have passed while calling the function.

```
141         for (uint i = 0; i < tokenAmount; i++) {
142             _safeMint(_msgSender(), totalSupply());
143         }
144
145         if (msg.value - tokenPrice > 0) {
146             payable(_msgSender()).transfer(msg.value - tokenPrice);
147         }
148     }
```

   However, the transfer of remaining ether back to the user is not done adequately and leads a major issue.

   The issue can be better understood in 2 different instances:

   A. **At Line 145** - In the condition of the **IF statement** , it is checked whether or not the "**msg.value - tokenPrice**" is greater than **zero**.
      This means the remaining amount of ether is calculated on the basis of just the "**tokenPrice**"

(which is a constant value) instead of the actual cost of the tokens, i.e., "**tokenPrice * tokenAmount**".

B. **At Line 146** - Once the execution enters the **IF statement**, an amount of ether equal to "**msg.value - tokenPrice**" is wrongly transferred to the user without taking into consideration the actual cost for the total tokens being bought.

Here is possible exploit scenario that might give a clear understanding of the above-mentioned issue: Considering the Token_Price to be equal to **70000000000000000**, a user wants to buy **5 tokens**.

The **mint()** function is called with an argument of 5 uint value and a **msg.value** of **350000000000000000**.

As all the require statements of the mint function is passed, the function moves to line 142 and mints 5 token for the **msg.sender()/caller** of the function.

Now, as the function execution moves to line **145**, the **if statement** condition is checked and qualified because:

350000000000000000(msg.value) - 70000000000000000(tokenPrice) > 0

Since the if statement is qualified, the execution moves to the next line at 146, and transfers an amount of wei equal to **280000000000000000** back to the caller.

350000000000000000 - 70000000000000000 = 280000000000000000

This results in a very unwanted scenario where, technically, the user only pays for one token instead of 5 tokens as the remaining amount of eth is transferred back to the user because of inadequate function design.

In the abobve-mentioned scenario, the user passed a **msg.value** of **350000000000000000** while calling the mint function but receives back **280000000000000000** amount. This wrong function design leads to a scenario where users can buy any amount of tokens for **70000000000000000** msg.value as the remaining amount will always be transferred back to the caller and the contract's balance will never increase.

Medium severity issues

No issues were found.

Low severity issues

1. **Fuctions can be marked as external**

   Explanation:
   Functions that are never called throughout the contract can 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:
   - name()
   - symbol()
   - balanceOf()
   - setBaseURI()
   - tokenOfOwnerByIndex()
   - tokenURI()

   Recommendation:
   If the **PUBLIC** visibility of the above-mentioned functions is not intended, then the **EXTERNAL** Visibility keyword should be preferred.

2. **Constant declaration should be preferred**
   Line no-50, 52, 56

   Explanation:
   State variables that are not supposed to change throughout the contract should be declared as **constant**.

   Recommendation:
   The following state variables need to be declared as constant, unless the current contract design is intended.
   - **name**
   - **symbol**
   - **_saleStart**

Recommendations

1. **NatSpec Annotations must be included**

   Description:
   The smart contracts do not include the NatSpec annotations adequately.

   Recommendation:
   Cover by NatSpec all Contract methods.

Unit Test

```
Compiling 13 files with 0.8.7
Compilation finished successfully


  Drrt contract
    ✓ get name
    ✓ get symbol
    ✓ mints one giveaway
    ✓ mints giveaway until limit is reached (110) (821ms)
    ✓ mints 2 times 1 presale and 1 normal sale NFTs. Check the owner
    ✓ does not mint NFT when sale is not started
    ✓ does not mint NFT if token amount is 0
    ✓ does not mint NFT if token amount is greater than 20
    ✓ does not mint NFT if Ether value is not correct
    ✓ returns token URI
    ✓ returns token URI does not exist if the token is not minted
    ✓ withdraw
    ✓ returns the owner of the token by index
    ✓ approve
    ✓ approve when approval to current owner
    ✓ approve when approve caller is not owner nor approved for all
    ✓ approve when approved query for nonexistent token
    ✓ approval for all
    ✓ approval for all when approve to caller
    ✓ transfer from
    ✓ transfer from when transfer caller is not owner nor approved
    ✓ safe transfer
    ✓ returns if sale is started
    ✓ reverts when balance of with zero address
    ✓ does not mint NFT if max supply reached (15481ms)


  25 passing (18s)
```

Automated Auditing

```
Compiled with solc
Number of lines: 1665 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 14 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 12
Number of informational issues: 55
Number of low issues: 2
Number of medium issues: 5
Number of high issues: 0

ERCs: ERC165, ERC721

+--------------------+-------------+---------------+------------+--------------+---------------+
|        Name        | # functions |      ERCS     | ERC20 info | Complex code |   Features    |
+--------------------+-------------+---------------+------------+--------------+---------------+
|      Strings       |      4      |               |            |     Yes      |               |
|      Address       |      11     |               |            |      No      |   Send ETH    |
|                    |             |               |            |              | Delegatecall  |
|                    |             |               |            |              |   Assembly    |
|      SafeMath      |      13     |               |            |      No      |               |
|   EnumerableSet    |      24     |               |            |      No      |   Assembly    |
|   EnumerableMap    |      16     |               |            |      No      |               |
| IERC721Enumerable  |      13     | ERC165,ERC721 |            |      No      |               |
|  IERC721Receiver   |      1      |               |            |      No      |               |
|        Drrt        |      54     | ERC165,ERC721 |            |      No      | Receive ETH   |
|                    |             |               |            |              |   Send ETH    |
+--------------------+-------------+---------------+------------+--------------+---------------+
```

Concluding Remarks

While conducting the audits of the MYNT smart contract, it was observed that the contracts contain High and Low severity issues.

Our auditors suggest that High and Low severity issues should be resolved by MYNT developers. 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 MYNT platform or its product nor 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