Uniqly

UniqDrop

Smart Contract Audit

Final Report

IMMUNE BYTES

Audits

uniqly.io

May 11, 2021

Introduction

1. About Uniqly

Uniqly is a platform created in response to the needs of the market. NFTs are becoming more and more popular, showing great potential, and yet looking at this trend as a bystander, it is hard to avoid being under the impression that what we see here is just a funny game, in which non-existent products of indeterminate value are being speculated on.
Uniqly.io creates the missing bridge. A bridge connecting the world of virtual NFT tokens with real, tangible products. We want to build an ecosystem that will eventually allow the NFTs to grow up to their true potential. This solution is revolutionary but also simple in its functioning. It allows us to introduce a real application of blockchain technology to the mainstream thanks to the possibility of the creation and tokenization of real, personalized products, including clothes, collectibles, and limited items.

Visit https://www.uniqly.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.

Documentation Details

Uniqly team has provided documentation for the purpose of conducting the audit. The documents are:

1. https://www.uniqly.io/whitepaper.pdf

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: Uniqly
- Contract Name: UniqDrop.sol
- Languages: Solidity(Smart contract), Javascript(Unit Testing)
- Github commit hash for audit: d9f77c0379e12e3de9b1dd6f7dc25a611751d7d3
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Solhint, VScode, Contract Library

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.

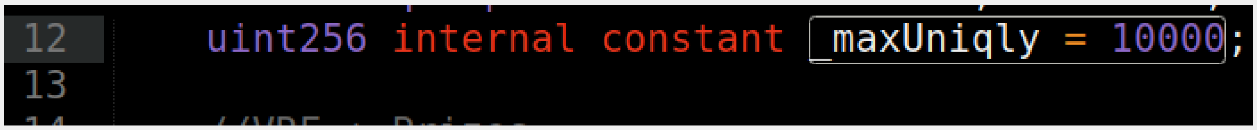
Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	-	-	2
Closed	3	1	2

High Severity Issues

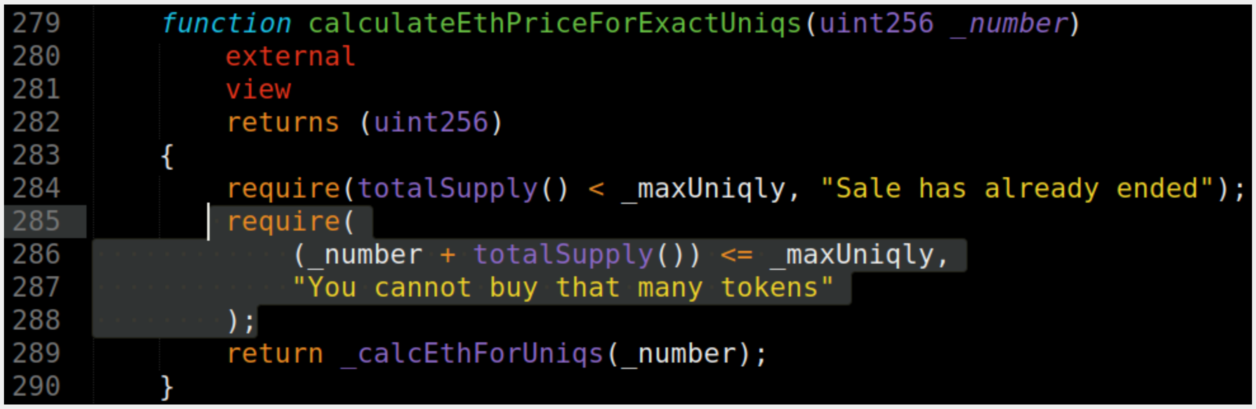
1. **More UniqlyNFT Tokens can be minted than the MAXIMUM ALLOWED Tokens(_maxUniqly)**
   Line no - 293 to 304

   Description:
   As per the current architecture of the contract, the maximum amount of tokens that should be minted is **10000**.



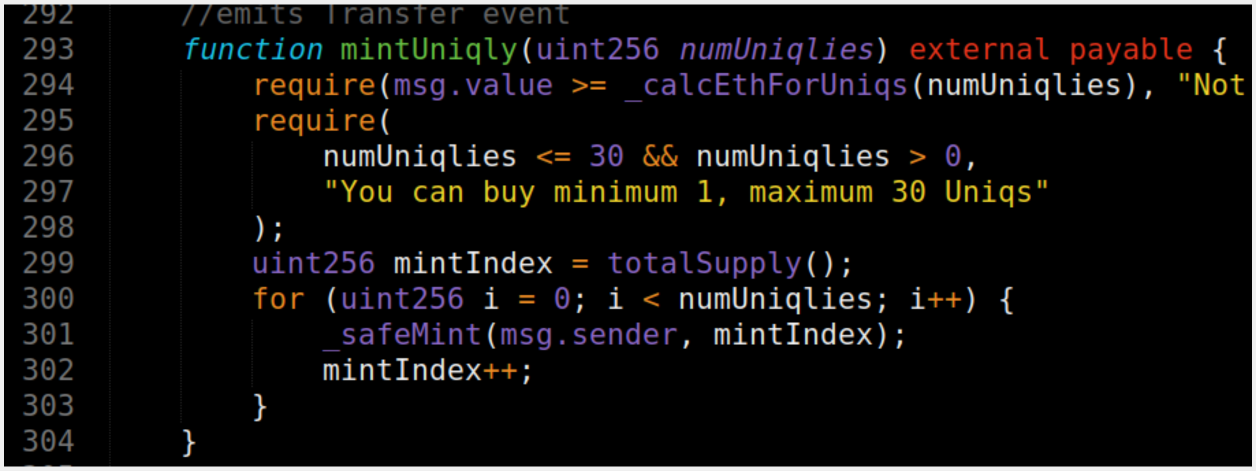
```
12      uint256 internal constant _maxUniqly = 10000;
13
```

In order to ensure that no tokens above this amount is minted, adequate validations have been included in the calculateEthPriceForExactUniqs function(Line 284-288).



```
279     function calculateEthPriceForExactUniqs(uint256 _number)
280         external
281         view
282         returns (uint256)
283     {
284         require(totalSupply() < _maxUniqly, "Sale has already ended");
285         require(
286             (_number + totalSupply()) <= _maxUniqly,
287             "You cannot buy that many tokens"
288         );
289         return _calcEthForUniqs(_number);
290     }
```

However, no such validations have been implemented in the mintUniqly function to ensure that the **totalSupply** does not exceed the **_maxUniqly** amount.



```
292         //emits Transfer event
293         function mintUniqly(uint256 numUniqlies) external payable {
294             require(msg.value >= _calcEthForUniqs(numUniqlies), "Not
295             require(
296                 numUniqlies <= 30 && numUniqlies > 0,
297                 "You can buy minimum 1, maximum 30 Uniqs"
298             );
299             uint256 mintIndex = totalSupply();
300             for (uint256 i = 0; i < numUniqlies; i++) {
301                 _safeMint(msg.sender, mintIndex);
302                 mintIndex++;
303             }
304         }
```

Although the mintUniqly function includes a require statement to check that the function argument **numUniqlies** should not be between 0 to 30(Line 295-298), it does not take into consideration the following scenario:

- If the total Supply is already at 9990 and a user wishes to mint 30 new tokens, then the total supply will exceed the _maxUniqly amount.

Recommendation:
If the above mentioned scenario is not intended, then a **require** statement must be implemented to ensure total supply always remains within the predefined limit.

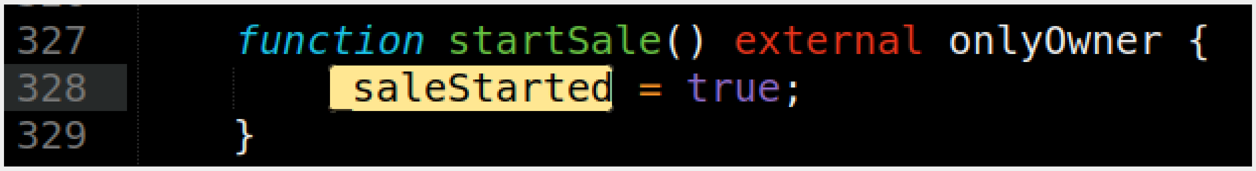
An effective way of doing so is to use **calculateEthPriceForExactUniqs** function instead of **_calcEthForUniqs** function in the **require** statement of **mintUniqly** function at **Line 294**. This is because the calculateEthPriceForExactUniqs already implements all the significant **require** statements in this case and ensures that the maximum **totalSupply** of tokens is always as expected.

Moreover, if there are no such restrictions on the **totalSupply** of the token and tokens can be minted upto any maximum amount, then the contract should be updated accordingly.

Amended (May 11th 2021): Issue was fixed by Uniqly team and is no longer present in the code. GitHub Commit number: d9f77c0379e12e3de9b1dd6f7dc25a611751d7d3

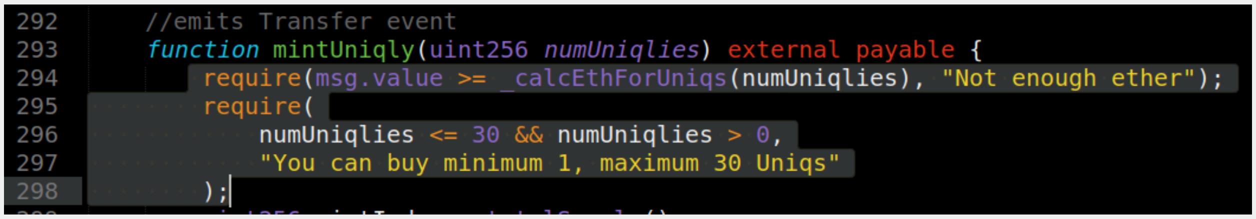
2. **New Uniqly Tokens can be Minted even before the SALE has Started**
   Line no - 293 to 304

   Description:
   The current contract design allows the owner to start the sale(Line 327-239) but doesn't use this information effectively in the contract.



```
327        function startSale() external onlyOwner {
328            saleStarted = true;
329        }
```

   For instance, the **mintUniqly** function can be accessed and users can start investing as well as minting new tokens even before the owner triggers the startSale function and actually initiates the sale process.
   This is mainly because the **mintUniqly function** doesn't include any **require statements** to validate whether or not the **saleStarted** boolean is **TRUE**.



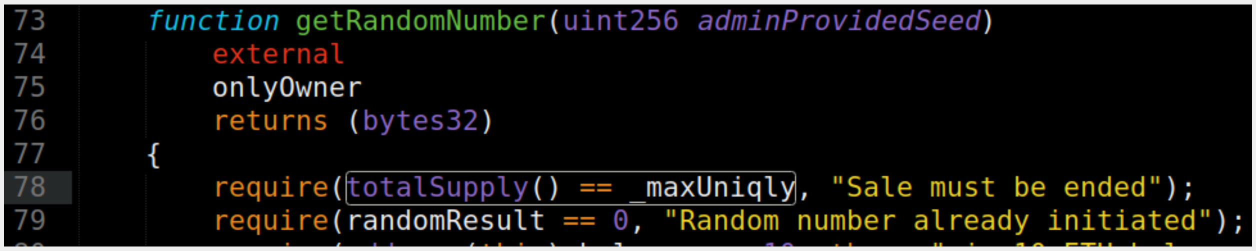
```
292     //emits Transfer event
293     function mintUniqly(uint256 numUniqlies) external payable {
294         require(msg.value >= _calcEthForUniqs(numUniqlies), "Not enough ether");
295         require(
296             numUniqlies <= 30 && numUniqlies > 0,
297             "You can buy minimum 1, maximum 30 Uniqs"
298         );
```

   Recommendation:
   In order to avoid the above-mentioned scenario, **a require statement must be included in the mintUniqly function** at the very beginning, as follows:
   require(_saleStarted, "ERROR MSG: Sale has not Started Yet")

   Amended (May 11th 2021): Issue was fixed by Uniqly team and is no longer present in the code. GitHub Commit number: d9f77c0379e12e3de9b1dd6f7dc25a611751d7d3

3. **getRandomNumber function includes a Strict Equality Check**
   Line no - 78

   Description:
   The getRandomNumber function includes a strict equality check between totalSupply and _maxUniqly at the very start of the function body.
   In order to execute this function, this require statement must be satisfied.



```
    function getRandomNumber(uint256 adminProvidedSeed)
        external
        onlyOwner
        returns (bytes32)
    {
        require(totalSupply() == _maxUniqly, "Sale must be ended");
        require(randomResult == 0, "Random number already initiated");
```

However, since there is a strict equality check, the function becomes completely inaccessible even if the totalSupply() value is slightly above the _maxUniqly.

Recommendation:
If the above-mentioned scenario is not intended, the require statement can be modified as follows:
require(totalSupply() >= _maxUniqly, "Sale must be ended");

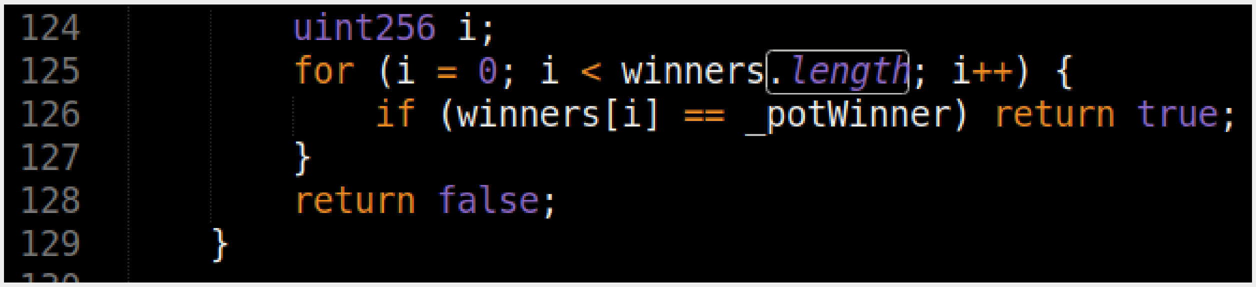
Amended (May 11th 2021): Issue was fixed by Uniqly team and is no longer present in the code. GitHub Commit number: [d9f77c0379e12e3de9b1dd6f7dc25a611751d7d3](d9f77c0379e12e3de9b1dd6f7dc25a611751d7d3)

Medium Severity Issues

1. **Loops are extremely costly**

   Line no - 61, 66

Description:
The **UniqDrop** contract has a **for loop** in the contract that includes state variables like .length of a non-memory array, in the condition of the for loops.



```
124            uint256 i;
125            for (i = 0; i < winners.length; i++) {
126                if (winners[i] == _potWinner) return true;
127            }
128            return false;
129        }
```

As a result, these state variables consume a lot more extra gas for every iteration of the for loop.
The following function includes such loops at the mentioned lines:

- **_isAlreadyRececeivedPrize at Line 125**

Recommendation:
It's quite effective to use a local variable instead of a state variable like .length in a loop. This will be a significant step in optimizing gas usage.
For instance,

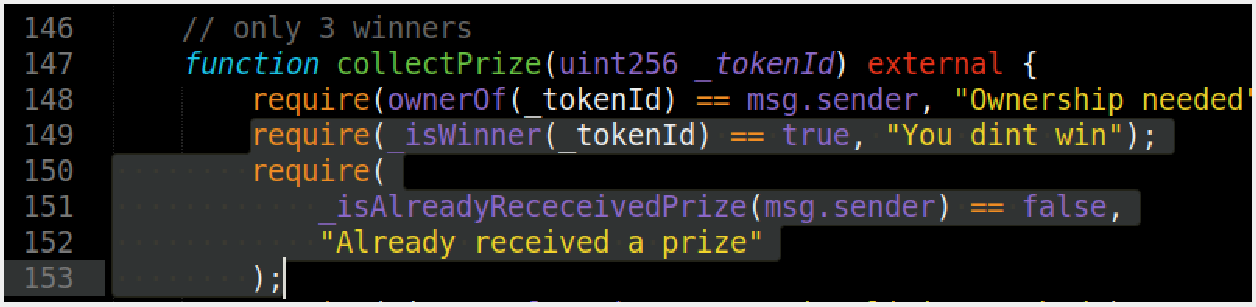
```
local_variable = winners.length;
 for (i = 0; i <local_variable; i++) {
           if (winners[i] == _potWinner) return true;
        }
```

Amended (May 11th 2021): Issue was fixed by Uniqly team and is no longer present in the code. GitHub Commit number: [d9f77c0379e12e3de9b1dd6f7dc25a611751d7d3](d9f77c0379e12e3de9b1dd6f7dc25a611751d7d3)

Low severity issues

1. **Comparison to boolean Constant**
   Line no: 149 to 153

   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.



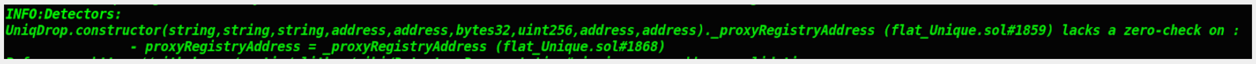
```
146     // only 3 winners
147     function collectPrize(uint256 _tokenId) external {
148         require(ownerOf(_tokenId) == msg.sender, "Ownership needed"
149         require(_isWinner(_tokenId) == true, "You dint win");
150         require(
151             _isAlreadyRececeivedPrize(msg.sender) == false,
152             "Already received a prize"
153         );
```

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

2. **constructor does not include Zero Address Validation**
   Line no: 250-253

   Description:
   The **constructor** initializes one of the most imperative state variables, i.e., **proxyRegistryAddress** in the UniqDrop contract.

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

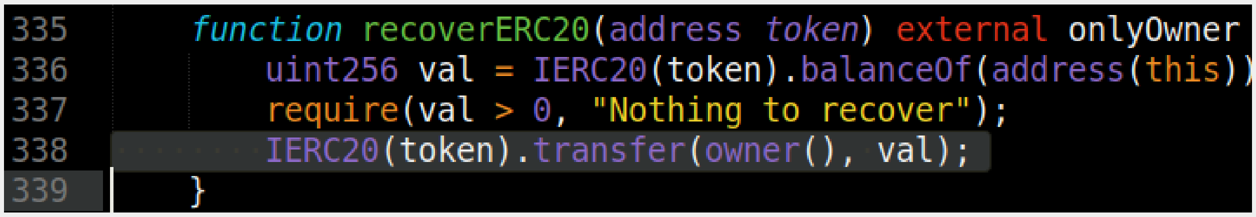


```
INFO:Detectors:
UniqDrop.constructor(string,string,string,address,address,bytes32,uint256,address,address)._proxyRegistryAddress (flat_Unique.sol#1859) lacks a zero-check on :
                - proxyRegistryAddress = _proxyRegistryAddress (flat_Unique.sol#1868)
```

Amended (May 11th 2021): Issue was fixed by Uniqly team and is no longer present in the code. GitHub Commit number: [d9f77c0379e12e3de9b1dd6f7dc25a611751d7d3]()

3. **Return Value of an External Call is never used Effectively**
   Line no - 338

Description:
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.
However, the UniqDrop contract never uses these return values throughout the contract.



```
335     function recoverERC20(address token) external onlyOwner
336         uint256 val = IERC20(token).balanceOf(address(this))
337         require(val > 0, "Nothing to recover");
338         IERC20(token).transfer(owner(), val);
339     }
```

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

Amended (May 11th 2021): **"**Point you cannot check the return value because USDT does not return it and would throw an error"
While auditing the contract our team assumption was for a standard ERC20 token.

Issue found during the Final Audit

4. **calculateEthPriceForExactUniqs function can be assigned an external visibility**
   Line no - 330

   Description:
   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 as improve the code readability.

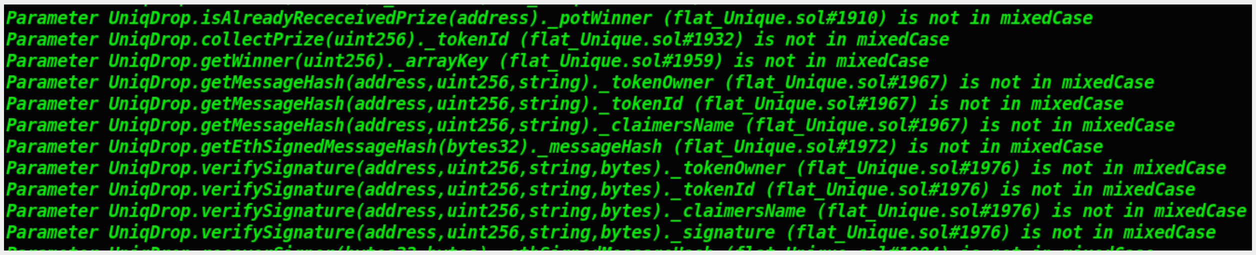
   Recommendation:
   If the **PUBLIC** visibility of **calculateEthPriceForExactUniqs** function is not intended, **EXTERNAL** visibility should be assigned to it.

Recommendations

1. Coding Style Issues in the Contract

Description:
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.

During the automated testing, it was found that the UniqDrop contract had quite a few code style issues.



```
Parameter UniqDrop.isAlreadyRececeivedPrize(address)._potWinner (flat_Unique.sol#1910) is not in mixedCase
Parameter UniqDrop.collectPrize(uint256)._tokenId (flat_Unique.sol#1932) is not in mixedCase
Parameter UniqDrop.getWinner(uint256)._arrayKey (flat_Unique.sol#1959) is not in mixedCase
Parameter UniqDrop.getMessageHash(address,uint256,string)._tokenOwner (flat_Unique.sol#1967) is not in mixedCase
Parameter UniqDrop.getMessageHash(address,uint256,string)._tokenId (flat_Unique.sol#1967) is not in mixedCase
Parameter UniqDrop.getMessageHash(address,uint256,string)._claimersName (flat_Unique.sol#1967) is not in mixedCase
Parameter UniqDrop.getEthSignedMessageHash(bytes32)._messageHash (flat_Unique.sol#1972) is not in mixedCase
Parameter UniqDrop.verifySignature(address,uint256,string,bytes)._tokenOwner (flat_Unique.sol#1976) is not in mixedCase
Parameter UniqDrop.verifySignature(address,uint256,string,bytes)._tokenId (flat_Unique.sol#1976) is not in mixedCase
Parameter UniqDrop.verifySignature(address,uint256,string,bytes)._claimersName (flat_Unique.sol#1976) is not in mixedCase
Parameter UniqDrop.verifySignature(address,uint256,string,bytes)._signature (flat_Unique.sol#1976) is not in mixedCase
```

Recommendation:
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.

Concluding Remarks

While conducting the audits of Uniqly smart contracts, it was observed that the contracts contain only High, Medium and Low severity issues, along with a few areas of recommendations.

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

Note: Uniqly team has fixed the High and Medium issues based on the auditor's recommendation.

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 Uniqly 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.