Uniqly

Smart Contract Audit Report

April 12, 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
- Languages: Solidity(Smart contract), Javascript(Unit Testing)
- Github commit hash for audit:
- Testnet Code (Testnet):
    - UniqToken: 0x76D92BDaab4Fb45c3cb8aA9198C53d181C886Ea9
    - UniqVesting: 0xc589498f462FF538DDeb4A53c5E67A01fdBa390f
- 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	-	-	4
Closed	-	-	-

Low severity issues

1. In the UniqVesting contract on line 185 instead of using the require statement we can simply use the modifier **onlyOwner** on line 184.

```
183     // only current owner can delegate new one
184     function giveOwnership(address _newOwner) external {
185         require(msg.sender == owner, "Only for Owner");
186         newOwner = _newOwner;
187     }
```

2. In the UniqVesting contract on line 214 in the require message there is a typo. Instead of **match** there's **math.** Please rectify that.

```
210     function addInvestors(address[] calldata addr, uint256[] calldata amount)
211         external
212         onlyOwner
213     {
214         require(addr.length == amount.length, "Data length not math");
215         for (uint256 i = 0; i < addr.length; i++) {
216             _addInvestor(addr[i], amount[i]);
217         }
218     }
```

3. In the **balanceOf** function of the UniqVesting contract instead of dividing first please multiply first and then divide because in all cases where **pctWithdrawn** is greater than 0 **balanceOf** will return **0.**

   Instead we can use: **(_tokensTotal[user] * (100 - pctWithdrawn[user])) / 100;**

```
108     function balanceOf(address user) external view returns (uint256) {
109         return _tokensTotal[user] * ((100 - _pctWithdrawn[user]) / 100);
110     }
```

4. In the UniqVesting contract on line 191 in the require message there is a typo. Instead of **You're** there's **Ure.** Please rectify that.

```
189     // new owner need to accept ownership
190     function acceptOwnership() external {
191         require(msg.sender == newOwner, "Ure not New Owner");
192         newOwner = address(0);
193         owner = msg.sender;
194     }
```

Unit Test

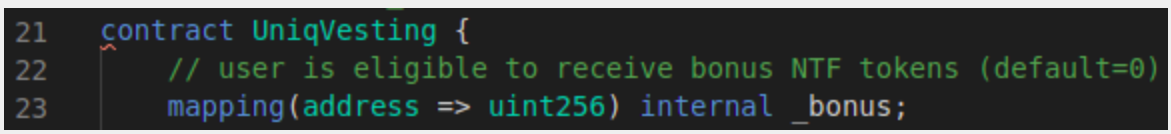
Test cases using mocha gave the following results

```
> uniqly@1.0.0 test /home/luv/Desktop/uniqly-contracts
> mocha --exit --recursive



  UniqToken
    ✓ has a name (39ms)
    ✓ has a symbol
    ✓ has 18 decimals (41ms)
    ✓ has 10000 initial balance
    ✓ a publisher is owner (40ms)
    Transfer
      ✓ an owner transfers to recipient (59ms)
      ✓ a sender has a proper balance
      ✓ a recipient has a proper balance after transfer
    Burn
      ✓ an owner burns tokens (63ms)
      ✓ total supply changed
    Approve/allowance
      ✓ set allowance (55ms)
      ✓ use TransferFrom (76ms)
      ✓ use burnFrom (88ms)
    rescueERC20: Rouge token withdrawal
      ✓ deploy rouge token contract (87ms)
      ✓ transfer rouge tokens (114ms)
      ✓ Rescue tokens (59ms)
    Change ownership
      ✓ Owner delegates Minter (48ms)
      ✓ Minter accepts (52ms)
      ✓ Minter is new Owner

  UniqVesting
    initialize()
      ✓ Should initialize properly (266ms)
    calc()
      ✓ Should be able to calculate and store total tokens without withdrawing for investor1 (149ms)
      ✓ Should be able to calculate and store total tokens without withdrawing for investor2 (159ms)
      ✓ Should be able to calculate and store total tokens without withdrawing for investor3 (127ms)
      ✓ Should not be able to call calc more than once for a single investor (96ms)
    claim()
      ✓ Should not be able to claim if we haven't reached startDate (39ms)
      ✓ Should be able to claim once we reach startDate with no bonus (232ms)
      ✓ Should be able to claim with a bonus after 10 vesting periods pass only if investor has claimed no more than
20% before (266ms)
      ✓ Should be able to claim without a bonus after 10 vesting periods pass if investor has claimed more than
20% before (327ms)
      ✓ Should be able to claim with a bonus after 20 vesting periods pass if investor has claimed no more than
20% before (366ms)
      ✓ Should be able to claim without a bonus after 20 vesting periods pass if investor has claimed more than
```

```
20% before (510ms)
  giveOwnership()/acceptOwnership()
    ✓ Should be able to propose new owner (153ms)
    ✓ Should be able to accept ownership (303ms)
    ✓ Should not be able to propose new owner if not called by current owner (83ms)
    ✓ Should not be able to accept ownership if not called by proposed owner (272ms)
  addInvestor()/addInvestors()
    ✓ Should be able to add investor (158ms)
    ✓ Should be able to add multiple investors (259ms)
    ✓ Should not be able to add investor if not called by owner (143ms)
    ✓ Should not be able to add multiple investors if not called by owner (150ms)
    ✓ Should not be able to add multiple investors if array lengths don't match (208ms)
  rescueERC20()
    ✓ Should be able to rescue token (588ms)
    ✓ Should not be able to rescue token if not called by owner (348ms)
    ✓ Should not be able to rescue token if there is nothing to rescue (364ms)


  42 passing (24s)
```

Recommendations

1. In both UniqVesting and UniqToken as the abstract contracts are never actually inherited and implemented we can declare **IErc20** and **IContracts** as interfaces.
2. In both the contracts, to improve readability, we can group all state variables at the top, then the mappings, events, modifiers and the rest of the functions. The order of functions followed by the solidity style guide is constructor, receive function (if exists), fallback function (if exists), external, public, internal and then lastly private.
3. We can add the **rescueETH()** function to the UniqVesting contract as well like it's present in the UniqToken contract.
4. We can add events to the following functions:
    - UniqVesting
        - **claim()**
        - **acceptOwnership()**
        - **_addInvestor()**
    - UniqToken
        - **acceptOwnership()**
        - **rescueETH()**
5. In the UniqVesting contract on line 22 there is a typo in the comments. I believe you mean to write **NFT** here :)



6. All top level definitions should be surrounded by 2 blank lines(you can read more here). There are instances in both UniqVesting and UniqToken at lines 5, 12, 21 and lines 5, 12 respectively. For eg.

```
2   pragma solidity 0.8.2;
3
4   // Contracts interaction interface
5   abstract contract IContract {
6       function balanceOf(address owner) external virtual returns (uint256);
7
8       function transfer(address to, uint256 value) external virtual;
9   }
10
11  // https://eips.ethereum.org/EIPS/eip-20
12  contract UniqToken {
```

7. We can add a check in **_addInvestor()** which checks whether the investor is being added before the vesting period ends but this is a mere suggestion and it fully depends on the use case whether we want to add investors even after the vesting period has ended.

Notes

1. ***Note 1:*** *While testing a mock of the UniqVesting was used in which the vesting periods were in minutes instead of weeks and the rest of the code was exactly the same. This was done for ease in testing.*

2. ***Note 2:*** *While testing on Ropsten and calling the claim function it failed with a "Out of gas" error. This was a problem with Metamask and when the transaction was submitted with a higher gas limit it went through. So in conclusion there is no problem in the smart contract logic but it was an issue with Metamask.*

Automated Audit

Solhint Linting Violations

Solhint is an open-source project for linting solidity code, providing both security and style guide validations. It integrates seamlessly into most mainstream IDEs. We used Solhint as a plugin within our VScode for this analysis. Multiple Linting violations were detected by Solhint, it is recommended to use Solhint's npm package to lint the contract.

Contract Library

Contract-library contains the most complete, high-level decompiled representation of all Ethereum smart contracts, with security analysis applied to these in real-time.
We performed analysis using the contract Library on the Ropsten address of the Uniqly contracts used during manual testing:

- UniqToken: 0x76D92BDAAB4FB45C3CB8AA9198C53D181C886EA9
- UniqVesting: 0xC589498F462FF538DDEB4A53C5E67A01FDBA390F

It raised no warnings in any of the contracts.

Concluding Remarks

While conducting the audits of Uniqly smart contracts, it was observed that the contracts contain only Low severity issues, along with several areas of recommendations.

Our auditors suggest that 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.

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.