Youniq Labs

smart contracts
preliminary audit report
for internal use only

April 2023

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 Youniq Labs team to perform an audit of their smart contract. The audit was conducted between 19/04/2023 and 24/04/2023.

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 is available at https://github.com/techbandorg/youniq-labs-contracts GitHub repository and was audited after the commit 8529ab2.

Update. A recheck was done after the commit 52d9c51.

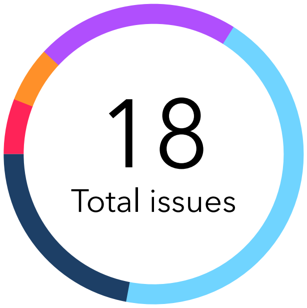
2.1 Summary

Project name	Youniq Labs
URL	https://youniqlabs.io/
Platform	Ethereum
Language	Solidity

2.2 Contracts

Name	Address
DutchAuction	
PrivateDistribution	
UnkwnBonesNftToken	
UnkwnBonesStaking	
Whitelist	

3. Found issues



Critical	1 (6%)
High	1 (6%)
Medium	4 (22%)
Low	8 (44%)
Info	4 (22%)

C8f. DutchAuction

ID	Severity	Title	Status
C8f17d	Critical	DoS attack	Resolved
C8f17e	Medium	Native currency sent to the contract will be locked	Resolved
C8f180	Low	Lack of reentrancy protection	Resolved
C8f181	Low	Not indexed parameters in the events	Resolved
C8f17f	Low	Gas optimizations	Resolved
C8f18c	Info	Typos	Acknowledged

C90. PrivateDistribution

ID	Severity	Title	Status
C90182	High	DoS attack	Resolved

C90I86	Medium	Native currency sent to the contract will be locked	Resolved
C90I84	Low	Lack of reentrancy protection	Resolved
C90I85	Low	Gas optimizations	Partially fixed
C90I8d	Info	Typos	Acknowledged

C91. UnkwnBonesNftToken

ID	Severity	Title	Status
C91I87	Medium	Native currency sent to the contract will be locked	Resolved
C91I88	Low	Excessive inheritance of the Ownable contract	Resolved
C91I8e	Info	Typos	Acknowledged

C92. UnkwnBonesStaking

ID	Severity	Title	Status
C92I89	Medium	Native currency sent to the contract will be locked	Resolved
C92I8a	Low	Lack of reentrancy protection	Resolved
C92I8b	Low	Gas optimizations	Resolved
C92I8f	Info	Typos	Acknowledged

4. Contracts

C8f. DutchAuction

Overview

The DutchAuction smart contract facilitates the sale of YouniqLabs NFT tokens. Users deposit tokens into the contract. After the deadline, the contract owner sets a price for the tokens and distributes them to users by minting them to the users via the DutchAuction smart contract. The owner also can send refunds for the assets that were not used for token purchases (leftovers). It should be noted that the contract does not follow the traditional Dutch auction format; rather, it is primarily a token sale mechanism.

Centralization risks

- The owner can withdraw all funds deposited to the contract and not mint any tokens to users.

- The owner can set an arbitrarily big price for NFT tokens after the token sale.

- The owner may not call the `refund()` function to send to users leftovers after token purchase.

Issues

C8fl7d DoS attack ● Critical ✓ Resolved

After the sale is ended the owner distributes purchased tokens to the recipients. The function `mint()` iterates over all purchases.

```
function mint(uint256 totalMints) external ended onlyOwner {
    if (totalMints == 0) {
        revert ZeroValue("Auction: Quantity should be > zero");
    }
    if (finalPrice == 0) {
        revert ZeroValue("Auction: Final price not setted");
```

```
        }
        uint256 startIndex = mintIndex;
        (uint256 endIndex, uint256 totalTrx) = getMintInfo(totalMints);
        uint256 mintCounter = totalMints;

        if (endIndex == totalBidders() - 1 && totalTrx == 0) {
            revert MintFinished("Auction: mint already finished");
        }
        for (uint256 i = startIndex; i <= endIndex; i++) {
            (uint256 totalNfts, ) = getUserInfo(bidders[i]);
            User storage userData = userInfo[bidders[i]];
            uint256 mintEndIndex = totalNfts - userData.totalMintedNfts <
                mintCounter
                ? totalNfts - userData.totalMintedNfts
                : mintCounter;
            for (uint256 j = 0; j < mintEndIndex; j++) {
                _mint(bidders[i]);
                mintCounter--;
            }
        }
        mintIndex = endIndex;
    }
```

In each iteration, it calls YouniqLabsNFT's `safeMint()` function to mint the token to a recipient. This function checks if the recipient is a contract or not and if so checks if the contract implements an interface to receive tokens. If not the function reverts.

```
    function _mint(address user) internal {
        (uint256 totalNfts, ) = getUserInfo(user);
        User storage userData = userInfo[user];
        if (userData.totalMintedNfts < totalNfts) {
            IYouniqLabsNft(nftToken).safeMint(user);
            ++userData.totalMintedNfts;
            ++totalMintedNfts;
        }
    }
```

An attacker can purchase tokens by a contract that will intentionally revert the `safeMint()` function. The owner won't be able to mint tokens to users whose addresses are after the malicious contract in the bidders array.

The same goes to the `refund()` function, which strictly requires successful ETH transfers to all the recipients. An attacker can sabotage the process by implementing a revert in the `receive()` function.

```
function refund(uint256 quantity) external ended onlyOwner {
    if (quantity == 0) {
        revert ZeroValue("Auction: Quantity should be > zero");
    }
    if (finalPrice == 0) {
        revert ZeroValue("Auction: Final price not setted");
    }
    uint256 startIndex = refundIndex;
    uint256 endIndex = refundIndex + quantity;

    if (endIndex > bidders.length) {
        revert InvalidIndex("Auction: Invalid index");
    }
    for (uint256 i = startIndex; i < endIndex; i++) {
        (, uint256 refundAmount) = getUserInfo(bidders[i]);
        User storage userData = userInfo[bidders[i]];
        // userData.refund = refundAmount;
        if (!userData.isRefunded && refundAmount > 0) {
            userData.isRefunded = true;
            _sendETH(bidders[i], refundAmount);
        }
        refundIndex++;
    }
}

function _sendETH(address to_, uint256 amount_) internal {
    (bool success, ) = to_.call{value: amount_}("");
    require(success, "Transfer failed");
}
```

Recommendation

Use a pull payment pattern to distribute the tokens allowing users to withdraw purchased tokens by themselves. This will also increase decentralization and users will be less dependent on the owner's account. Refunding should then be performed in the same transaction.

C8fl7e Native currency sent to the contract will be locked

Medium | Resolved

The contract contains the `receive()` function. This function allows the contract to accept native currency sent directly to it. But this functionality is not used anywhere in the contract.

```
receive() external payable {}
```

Recommendation

Remove the `receive()` function.

C8fl80 Lack of reentrancy protection

Low | Resolved

The functions `IYouniqLabsNft(nftToken).safeMint()` and `_sendEth()` can lead to reentrancy attacks.

Recommendation

Add reentrancy guard from OpenZeppelin library to functions that call `safeMint()` and `_sendEth()`.

C8fl81 Not indexed parameters in the events

Low | Resolved

It is essential to optimize event logging to ensure efficient data retrieval and filtering. Indexing event parameters allows users to filter logs more effectively.We recommend adding an indexed `user` parameter to the Bid event and a `recipient` parameter in the `WithdrawEth` event.

C8fl7f Gas optimizations

Low | Resolved

- the `nftToken` variable can be declared as immutable.

- multiple reads from storage in the `mint()` function: `mintIndex`, `bidders[i]`, `userData.totalMintedNfts`.

- multiple reads from storage in the `refund()` function: `bidders[i]`, `userData.totalMintedNfts`.

- multiple reads from storage in the `getMintInfo()` function: `bidders[i]`, `bidders.length`.

- multiple reads from storage in the `getUserInfo()` function: `finalPrice`.

- unnecessary reads from storage in the `updateMinBid()` function: `minBid`.

- unnecessary reads from storage in the `getUserInfo()` function: full `User` structure.

- no need to update a storage variable `refundIndex` in every iteration of the loop in the `refund()` function.

C8fl8c Typos

Info | Acknowledged

Typos reduce the code's readability. Typos in 'setted', 'addresss'.

C90. PrivateDistribution

Overview

An NFT token sale contract where whitelisted users can buy the YouniqLabs NFT token at a given price. After the sale is ended the owner of the contract distributes purchased tokens.

Centralization risk

- The owner may not distribute purchased tokens after the sale.

Issues

C90I82 DoS attack

High | Resolved

After the sale is ended the owner of the contract distributes NFT tokens with the `mint()` function which iterates over accounts that bought the tokens.

```
function mint(uint256 quantity) external onlyOwner {
    if (block.timestamp < distributionInfo.end) {
        revert DistributionNotEnded("PrivateDistribution: Not ended");
    }
    uint256 startIndex = mintIndex;
    uint256 endIndex = mintIndex + quantity;

    if (endIndex > mintList.length) {
        revert InvalidIndex("PrivateDistribution: Invalid index");
    }
    if (quantity == 0) {
        revert ZeroValue("PrivateDistribution: Quantity should be > zero");
    }
    for (uint256 i = startIndex; i < endIndex; i++) {
        _mint(mintList[i]);
        mintIndex++;
    }
}
```

If mint to a user fails, the owner won't be able to distribute tokens to subsequent users. An attacker can make a purchase for a contract which will revert when `safeMint()` to it is called.

Recommendation

Use a pull payment pattern for the distribution of the purchased tokens.

C90I86 Native currency sent to the contract will be locked

Medium | Resolved

The contract has a `receive()` function that allows the contract to receive native currency that is sent directly to it. However, the contract does not have any mechanism to withdraw such sent funds.

```
/**
@dev fallback function to obtain ETH per contract.
*/
receive() external payable {}
```

Recommendation

Remove the `receive()` function.

C90I84 Lack of reentrancy protection

Low | Resolved

The functions `IYouniqLabsNft(nftToken).safeMint()` and `_sendEth()` can lead to reentrancy attacks.

Recommendation

Add a reentrancy guard from the OpenZeppelin library to the functions that call `safeMint()` and `_sendEth()`.

C90I85 Gas optimizations

Low | Partially fixed

- `nftToken`, `revenueRecipient` variables can be declared as immutable.

- storage variable `mintIndex` is updated in every loop iteration in the `mint()` function.

- multiple storage reads in the `mint()` function: `mintIndex`.

- multiple storage reads in the `_buy()` function: `userBalance[user]`.

- multiple storage reads in the `_mint()` function: `userBalance[user]`, `nftToken`.

- unnecessary storage reads in the `updateDistributionStartTime()` function: `distributionInfo.end`.

- unnecessary storage reads in the `updateDistributionDuration()` function: `distributionInfo.end`, `distributionInfo.distributionDuration`.

- no need to send native currency to the revenue recipient on every purchase. Use a pull payment pattern.

C90I8d Typos

Info | Acknowledged

Typos reduce the code's readability. Typos in 'availavble', 'onwer'.

C91. UnkwnBonesNftToken

Overview

The UnkwnBonesNftToken smart contract is an ERC721 token implementation that includes the ERC2981 royalty standard, ERC721Enumerable, and access control features. The token has a maximum supply which is set in the constructor.

Centralization risks

- The owner can set a URI of the token to a wrong value.

- Accounts with MINTER roles can mint new tokens until the number of the tokens reaches the maximum total supply.

- The owner can change the recipient and the amount of ERC2981 royalties.

Issues

C91I87 Native currency sent to the contract will be locked

Medium | Resolved

The contract has a `receive()` function that allows the contract to receive native currency that is sent directly to it. However, the contract does not have any mechanism to withdraw such sent funds.

```
/**
@dev fallback function to obtain ETH per contract.
*/
receive() external payable {}
```

Recommendation

Remove the `receive()` function.

C91I88 Excessive inheritance of the Ownable contract

Low | Resolved

The contract inherits both the AccessControl and Ownable contracts from OpenZeppelin. AccessControl contract includes all functionality of the Ownable contract and hence the inheritance from the Ownable can be removed and its functionality released via AccessControl.

C91I8e Typos

Info | Acknowledged

Typos reduce the code's readability. Typos in 'procceds'.

C92. UnkwnBonesStaking

Overview

Users send their NFT tokens to the contract during the staking time. After the staking time expires, users can claim their tokens back by paying a fee in the native currency. The amount of the fee is set by the contract owner and can be changed anytime.

No rewards of any kind are accounted to the stakers.

Centralization risks

- The owner may set an arbitrary big value for a fee to claim back deposited tokens.

- Owner may set an arbitrary value for unlocking time.

Issues

C92I89 Native currency sent to the contract will be locked — Medium — Resolved

The contract has a `receive()` function that allows the contract to receive native currency that is sent directly to it. However, the contract does not have any mechanism to withdraw such sent funds.

```
/**
@dev fallback function to obtain ETH per contract.
*/
receive() external payable {}
```

Recommendation

Remove the receive() function.

C92I8a Lack of reentrancy protection

Low | Resolved

The functions `_sendETH()` and `IERC721(nftToken).safeTransferFrom()` can lead to reentrancy attacks.

Recommendation

Add a reentrancy guard from the OpenZeppelin library to the functions that call `_sendETH()` and `IERC721(nftToken).safeTransferFrom()`.

C92I8b Gas optimizations

Low | Resolved

- `nftToken` variable can be declared immutable.

- multiple storage reads in the `userStakes()` function: `_userStakes[user].length()`.

C92I8f Typos

Info | Acknowledged

Typos reduce the code's readability. Typo in 'timestmamp'.

C93. Whitelist

Overview

A simple owner-governed contract implementing a whitelist of privileged accounts. No issues were found.

5. Conclusion

1 critical, 1 high, 4 medium, 8 low severity issues were found during the audit. 1 critical, 1 high, 4 medium, 7 low issues were resolved in the update.

The reviewed contracts are highly dependent on the owner’s account. See the centralization risks chapters for each contract. Users using the project have to trust the owner and that the owner's account is properly secured.

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

contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx

BLOCKCHAIN SECURITY