Otoro

Smart Contract Audit Report
Prepared for 0xStudio

Date Issued:	Jun 10, 2022
Project ID:	AUDIT2022037
Version:	v2.0
Confidentiality Level:	Public

inspex
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022037
Version	v2.0
Client	0xStudio
Project	Otoro
Auditor(s)	Natsasit Jirathammanuwat Darunphop Pengkumta Sorawish Laovakul
Author(s)	Darunphop Pengkumta
Reviewer	Patipon Suwanbol
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
2.0	Jun 10, 2022	Update reassessment scope	Natsasit Jirathammanuwat
1.0	Jun 6, 2022	Full report	Darunphop Pengkumta

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by 0xStudio, Inspex team conducted an audit to verify the security posture of the Otoro smart contracts between May 25, 2022 and May 26, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Otoro smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 low, 2 very low, and 2 info-severity issues. With the project team's prompt response, 1 low, 2 very low and 2 info-severity issues were resolved in the reassessment, while 1 low-severity issue was acknowledged by the team. Therefore, Inspex trusts that Otoro smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

The 0xStudio is the platform that provides services for ideating, developing, supporting, and deploying a fully-functioned Web3 application and providing smart contracts for a variety of applications that specifically serve users' goals.

The Otoro contract is an NFT contract that facilitates the platform owners to distribute their NFTs in multiple ways including private sale, public sale, Dutch auction sale, and airdrop.

Scope Information:

Project Name	Otoro
Website	https://www.0x.studio/
Smart Contract Type	Ethereum Smart Contract
Chain	Ethereum Mainnet
Programming Language	Solidity
Category	NFT

Audit Information:

Audit Method	Whitebox
Audit Date	May 25, 2022 - May 26, 2022
Reassessment Date	Jun 2, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 441f0995a86a421d7062f30abcdf654d86d0013c)

Contract	Location (URL)
Otoro	https://github.com/0xstudio/Otoro-Audit/blob/441f0995a8/contracts/Otoro.sol
BlockbasedSale	https://github.com/0xstudio/Otoro-Audit/blob/441f0995a8/contracts/lib/BlockbasedSale.sol
RequestSigning	https://github.com/0xstudio/Otoro-Audit/blob/441f0995a8/contracts/lib/RequestSigning.sol
Revealable	https://github.com/0xstudio/Otoro-Audit/blob/441f0995a8/contracts/lib/Revealable.sol
Roles	https://github.com/0xstudio/Otoro-Audit/blob/441f0995a8/contracts/lib/Roles.sol

Reassessment: (Commit: 913f41ec4e39710606072740c35077894df74902)

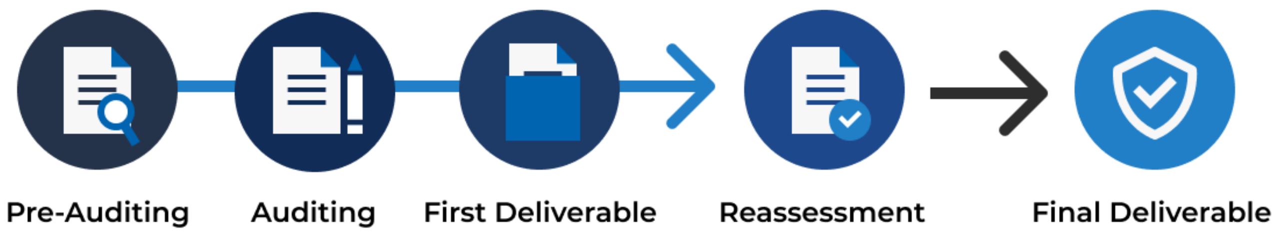
Contract	Location (URL)
Otoro	https://github.com/0xstudio/Otoro-Audit/blob/913f41ec4e/contracts/Otoro.sol
BlockbasedSale	https://github.com/0xstudio/Otoro-Audit/blob/913f41ec4e/contracts/lib/BlockbasedSale.sol
RequestSigning	https://github.com/0xstudio/Otoro-Audit/blob/913f41ec4e/contracts/lib/RequestSigning.sol
Revealable	https://github.com/0xstudio/Otoro-Audit/blob/913f41ec4e/contracts/lib/Revealable.sol
Roles	https://github.com/0xstudio/Otoro-Audit/blob/913f41ec4e/contracts/lib/Roles.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

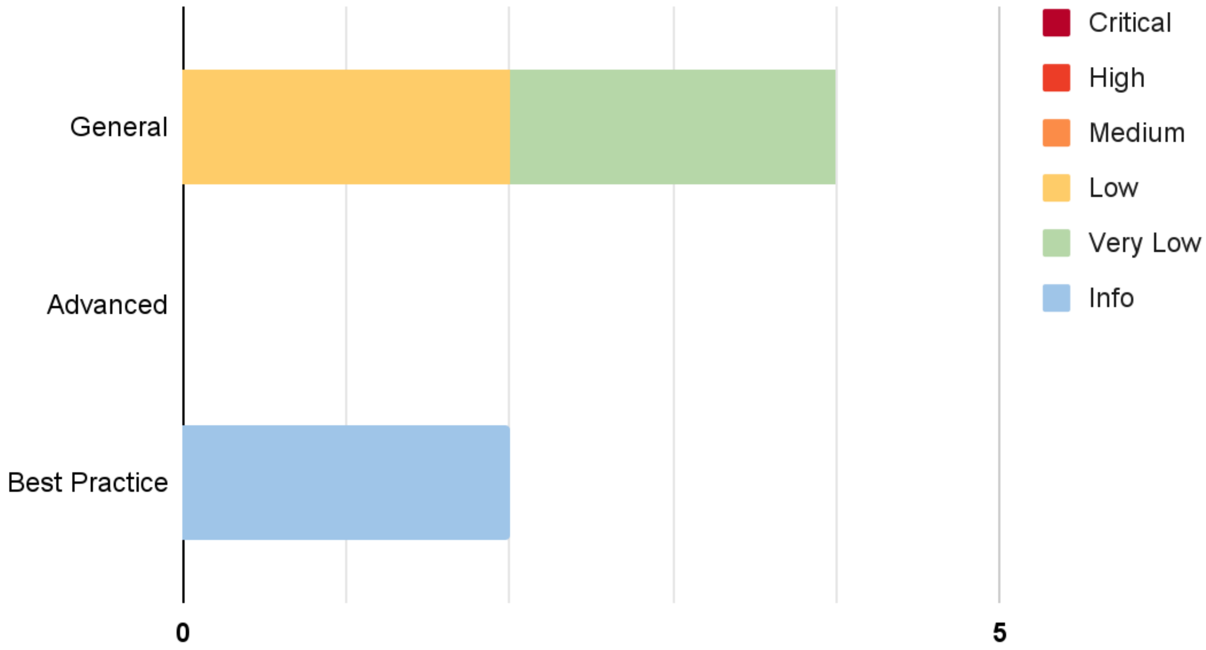
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

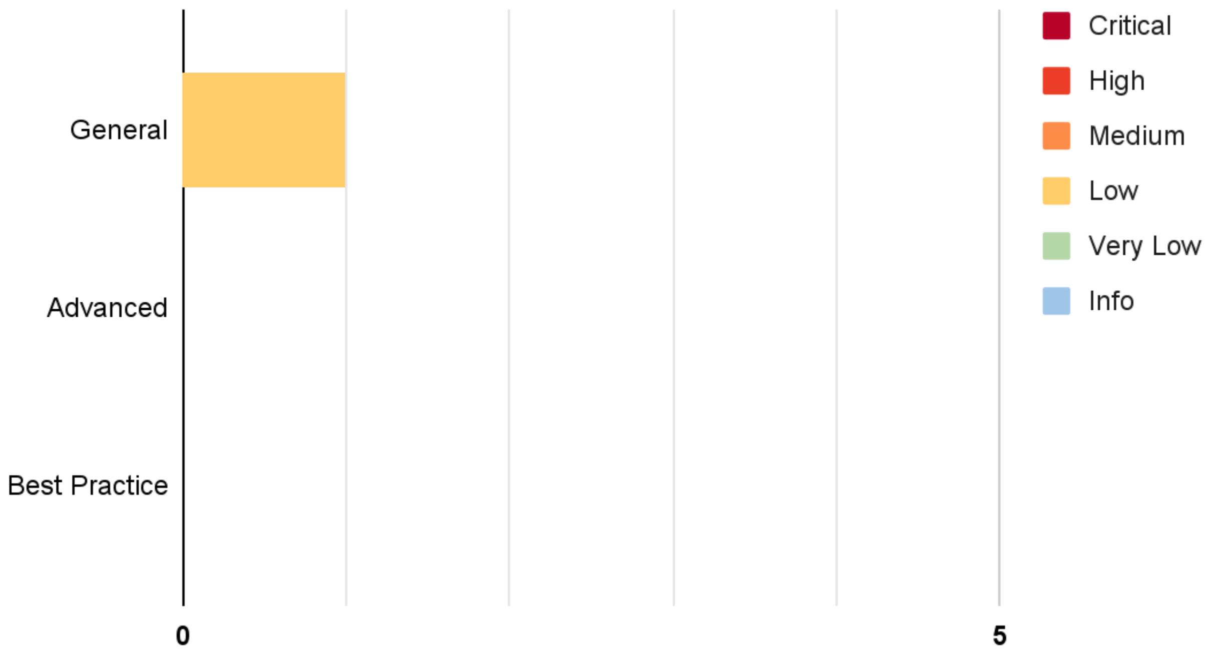
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Centralized Control of State Variable	General	Low	Acknowledged
IDX-002	Loop Over Unbounded Data Structure	General	Low	Resolved
IDX-003	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-004	Outdated Compiler Version	General	Very Low	Resolved
IDX-005	Improper Function Visibility	Best Practice	Info	Resolved
IDX-006	Unnecessary Condition Checking	Best Practice	Info	Resolved

* The mitigations or clarifications by 0xStudio can be found in Chapter 5.

5. Detailed Findings Information

5.1. Centralized Control of State Variable

ID	IDX-001
Target	Otoro BlockbasedSale RequestSigning Revealable Roles
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Low **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. In this case, the owner changes the metadata of the entire NFT collection. **Likelihood:** Low Only the contract owner has permission to perform this action. However, if the owner uses this issue to take advantage, it will result in the platform's reputation loss.
Status	**Acknowledged** The 0xStudio team has acknowledged this issue.

5.1.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users. For example, the operator role can call the `setRevealedBaseURI()` function to change the metadata of the entire NFT collection at any time.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
BlockbasedSale.sol (L:145)	BlockbasedSale	setOverrideFinalDAPrice ()	onlyOperator
BlockbasedSale.sol (L:150)	BlockbasedSale	setDutchAuctionParam()	onlyOperator

BlockbasedSale.sol (L:161)	BlockbasedSale	setTransactionLimit()	onlyOperator
BlockbasedSale.sol (L:170)	BlockbasedSale	setPublicSalePrice()	onlyOperator
BlockbasedSale.sol (L:175)	BlockbasedSale	setPrivateSaleCapPrice()	onlyOperator
BlockbasedSale.sol (L:195)	BlockbasedSale	setReserve()	onlyOperator
BlockbasedSale.sol (L:200)	BlockbasedSale	setDutchAuctionCap()	onlyOperator
RequestSigning.sol (L:50)	RequestSigning	setWhitelistSigningKey()	onlyOperator
RequestSigning.sol (L:58)	RequestSigning	setOgSigningKey()	onlyOperator
Revealable.sol (L:48)	Revealable	setRevealedBaseURI()	onlyOperator
Roles.sol (L:33)	Roles	setOperatorAddress()	onlyOwner
Roles.sol (L:39)	Roles	setGovernorAddress()	onlyOwner

5.1.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests implementing a community-run smart contract governance to control the use of these functions.

If removing the functions or implementing the smart contract governance is not possible, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the changes for a reasonable amount of time, e.g., 24 hours.

If the timelock mechanism is used to mitigate this issue, the mentioned functions as in the table with the `onlyOperator` modifier should be changed to another modifier such as the `onlyConfigurator` modifier, then apply the timelock mechanism to the `onlyConfigurator` role instead to prevent the other functions from being affected by the timelock mechanism.

5.2. Loop Over Unbounded Data Structure

ID	IDX-002
Target	Otoro Revealable
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `tokenURI()` function will be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the owner might set the `maxSupply` parameter of the NFT collection higher than the maximum gas limit of the RPC node could handle.
Status	Resolved The 0xStudio team has resolved this issue by setting the value of the `maxSupply` state to be `10,000` to ensure maximum gas usage to be within the executable range in commit `0aebe44c1e709d5ee24cf03ed09bcb5b5bbb36a4`.

5.2.1. Description

While calling the `tokenURI()` function in the revealed state, the `maxSupply` state will be passed through the `getShuffledId()` function at line 279.

Otoro.sol

```
266 function tokenURI(uint256 tokenId)
267     public
268     view
269     override(ERC721)
270     returns (string memory)
271 {
272     require(tokenId <= totalSupply(), "Token not exist.");
273
274     return
275         isRevealed()
276             ? string(
277                 abi.encodePacked(
278                     revealedBaseURI,
279                     getShuffledId(totalSupply(), maxSupply, tokenId, 1),
280                     ".json"
281                 )
282             )
```

```
            : defaultURI;
}
```

The `maxSupply` value is used as the iteration number of the shuffle loop in the `getShuffledId()` function, as shown in line 98.

Revealable.sol

```
80  function getShuffledId(
81      uint256 totalSupply,
82      uint256 maxSupply,
83      uint256 tokenId,
84      uint256 startIndex
85  ) public view returns (string memory) {
86      if (_msgSender() != owner()) {
87          require(tokenId <= totalSupply, "Token not exists");
88      }
89  
90      if (!isRevealed()) return "default";
91  
92      uint256[] memory metadata = new uint256[](maxSupply + 1);
93  
94      for (uint256 i = 1; i <= maxSupply; i += 1) {
95          metadata[i] = i;
96      }
97  
98      for (uint256 i = startIndex; i <= maxSupply; i += 1) {
99          uint256 j = (uint256(keccak256(abi.encode(seed, i))) %
100             (maxSupply)) + 1;
101 
102         if (j >= startIndex && j <= maxSupply) {
103             (metadata[i], metadata[j]) = (metadata[j], metadata[i]);
104         }
105     }
106 
107     return Strings.toString(metadata[tokenId]);
108 }
```

Even though the `getShuffledId()` function is a view function, which does not cost any gas when it is called, there exists a limitation to this type of calling which depends on the RPC node setting. For example, Infura's RPC node has set the maximum gas limit of the `eth_call` command at around 300,000,000 units (10x of the current Ethereum Mainnet block gas limit), this means the highest `maxSupply` value that can be set is approximately around ≈100,000. If the `maxSupply` value is set higher than this value, the `getShuffledId()` function will be unusable.

5.2.2. Remediation

Inspex suggests implementing the pull over push strategy; for example, shuffle the `metadata` array every time that the NFT is minting instead of shuffle all at once every time that the `tokenURI()` function is called. However, preventing the setting of the `maxSupply` state higher than the limitation by hard coding the `maxSupply` value within the limitation can also resolve this issue.

5.3. Insufficient Logging for Privileged Functions

ID	IDX-003
Target	Otoro Revealable
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	**Resolved** The 0xStudio team has resolved this issue by adding the event emitters in the suggested functions in commit `0aebe44c1e709d5ee24cf03ed09bcb5b5bbb36a4`.

5.3.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the operator role can call the `setRevealedBaseURI()` function to change the metadata of the entire NFT collection, and no events are emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function
Revealable.sol (L:38)	Revealable	setDefaultURI()
Revealable.sol (L:44)	Revealable	setRevealBlock()
Revealable.sol (L:48)	Revealable	setRevealedBaseURI()
Otoro.sol (L:286)	Otoro	release()

5.3.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

Revealable.sol

```
event SetRevealedBaseURI(string _baseURI);
function setRevealedBaseURI(string memory _baseURI) external onlyOperator {
    revealedBaseURI = _baseURI;
    emit SetRevealedBaseURI(_baseURI);
}
```

5.4. Outdated Compiler Version

ID	IDX-004
Target	Otoro BlockbasedSale RequestSigning Revealable Roles
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Very Low **Impact:** Low From the list of known Solidity bugs, direct impact cannot be caused from those bugs themselves **Likelihood:** Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	Resolved The 0xStudio team has resolved this issue by changing the Solidity compiler version of the contracts to be the latest version in commit `0aebe44c1e709d5ee24cf03ed09bcb5b5bbb36a4`.

5.4.1. Description

The Solidity compiler versions specified in the smart contracts were outdated.These versions have publicly known inherent bugs (https://docs.soliditylang.org/en/v0.8.14/bugs.html) that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

Otoro.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.13;
```

The following table contains all targets which the outdated compiler version is declared.

Contract	Version
Otoro	0.8.13
BlockbasedSale	0.8.13

RequestSigning	0.8.13
Revealable	0.8.13
Roles	0.8.13

5.4.2. Remediation

Inspex suggests upgrading the Solidity compiler to the latest stable version (https://github.com/ethereum/solidity/releases).

At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.14.

5.5. Improper Function Visibility

ID	IDX-005
Target	Otoro BlockbasedSale
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The 0xStudio team has resolved this issue by changing the visibility of the suggested functions to external in commit `0aebe44c1e709d5ee24cf03ed09bcb5b5bbb36a4`.

5.5.1. Description

Public functions that are never called internally by the contract itself should have external visibility. This improves the readability of the contract, allowing clear distinction between functions that are externally used and functions that are also called internally.

The following source code shows that the `dutchAuctionInfo()` function in the `Otoro` contact is set to public and it is never called from any internal functions.

Otoro.sol

```
247 function dutchAuctionInfo(address user)
248     public
249     view
250     returns (MintInfo[] memory)
251 {
252     return fairDAInfo[user];
253 }
```

The following table contains all functions that have public visibility and are never called from any internal functions.

Target	Contract	Function
Otoro.sol (L:247)	Otoro	dutchAuctionInfo()
BlockbasedSale.sol (L:349)	BlockbasedSale	getStateName()

5.5.2. Remediation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function, as shown in the following example:

Otoro.sol

```
function dutchAuctionInfo(address user)
    external
    view
    returns (MintInfo[] memory)
{
    return fairDAInfo[user];
}
```

5.6. Unnecessary Condition Checking

ID	IDX-006
Target	Otoro Revealable
Category	Smart Contract Best Practice
CWE	CWE-571: Expression is Always True
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The 0xStudio team has resolved this issue by removing the unnecessary conditions from the contracts as suggested in commit `0aebe44c1e709d5ee24cf03ed09bcb5b5bbb36a4`.

5.6.1. Description

There are some condition checking statements that always result in `true` regardless of the function or the contract states. There are two cases where this has happened in the contracts.

This is a list of all the lines that contain the unnecessary condition.

File	Contract	Function
Otoro.sol (L:127)	Otoro	mintOg()
Otoro.sol (L:209-213)	Otoro	mintToken()
Revealable.sol (L:102)	Revealable	getShuffledId()

The first one is the case that is caused by the fact that the possible value always satisfies the checking operation. The condition in line 102, `j <= maxSupply`, of the `Otoro` contract checks that the value of the `j` variable does not exceed the `maxSupply` value.

The `j` variable is declared at line 99. The variable's value is determined by performing a modulo operation on a `uint256` number `(uint256(keccak256(abi.encode(seed, i))` with the `maxSupply` value. The range of the modulo operation is a value between `0` and `maxSupply-1`. Then, this value is added by `1`. So, the `j` variable's possible values are between `1` and `maxSupply` (we exclude the overflow case because the contract is compiled on versions newer than 0.8.0), which means that the `j <= maxSupply` condition will always be true regardless of the function inputs.

Revealable.sol

```
function getShuffledId(
    uint256 totalSupply,
    uint256 maxSupply,
    uint256 tokenId,
    uint256 startIndex
) public view returns (string memory) {
    if (_msgSender() != owner()) {
        require(tokenId <= totalSupply, "Token not exists");
    }

    if (!isRevealed()) return "default";

    uint256[] memory metadata = new uint256[](maxSupply + 1);

    for (uint256 i = 1; i <= maxSupply; i += 1) {
        metadata[i] = i;
    }

    for (uint256 i = startIndex; i <= maxSupply; i += 1) {
        uint256 j = (uint256(keccak256(abi.encode(seed, i))) %
            (maxSupply)) + 1;

        if (j >= startIndex && j <= maxSupply) {
            (metadata[i], metadata[j]) = (metadata[j], metadata[i]);
        }
    }

    return Strings.toString(metadata[tokenId]);
}
```

The other cases are caused by the states having already been checked earlier. Therefore, they are not needed to be checked again, if the states have not been changed intermediately.

Secondly, in the `mintOg()` function, the value from the `getState()` function is checked with `SaleState.PrivateSaleDuring` at line 123, then, the value from the `getState()` function, which is a view function that does not have any function argument, is checked again with the same value, `SaleState.PrivateSaleDuring` at line 127.

Otoro.sol

```
115 function mintOg(bytes calldata signature)
116     external
117     payable
118     nonReentrant
119     returns (bool)
120 {
```

```
    require(msg.sender == tx.origin, "Contract is not allowed.");
    require(
        getState() == SaleState.PrivateSaleDuring,
        "Sale not available."
    );

    if (getState() == SaleState.PrivateSaleDuring) {
        require(isOG(signature), "Not OG whitelisted.");
        require(_ogClaimed[msg.sender] == 0, "Already Claimed OG.");
        require(
            totalPrivateSaleMinted().add(1) <= privateSaleCapped,
            "Exceed Private Sale Limit"
        );

        require(msg.value >= getPriceByMode(), "Insufficient funds.");

        _ogClaimed[msg.sender] = _ogClaimed[msg.sender] + 1;
        saleStats.totalOGMinted = saleStats.totalOGMinted.add(1);

        _mintToken(msg.sender, 1);

        payable(_splitter).transfer(msg.value);

        return true;
    }

    return false;
}
```

Similarly, in the `mintToken()` function, the `state` state is checked from the lines between 159 and 161, and it will be checked again from the lines between 210 and 212. Between these two checks, there are no statements that modify the `state` state.

Otoro.sol

```
150 function mintToken(uint256 amount, bytes calldata signature)
151     external
152     payable
153     nonReentrant
154     returns (bool)
155 {
156     SaleState state = getState();
157     require(msg.sender == tx.origin, "Contract is not allowed.");
158     require(
159         state == SaleState.PrivateSaleDuring ||
160             state == SaleState.PublicSaleDuring ||
161             state == SaleState.DutchAuctionDuring,
```

```
            "Sale not available."
        );
        require(
            msg.value >= amount.mul(getPriceByMode()),
            "Insufficient funds."
        );

        if (state == SaleState.DutchAuctionDuring) {
            require(
                amount <= saleConfig.maxDAMintPerWallet,
                "Mint exceed transaction limits."
            );
            require(
                _dutchAuctionMinted[msg.sender] + amount <=
                    saleConfig.maxDAMintPerWallet,
                "Mint limit per wallet exceeded."
            );
            require(
                saleStats.totalDAMinted.add(amount) <= dutchAuctionCapped,
                "Purchase exceed limit."
            );
        }

        if (state == SaleState.PublicSaleDuring) {
            require(
                amount <= saleConfig.maxFMMintPerTx,
                "Mint exceed transaction limits."
            );
            require(
                totalSupply().add(amount).add(availableReserve()) <= maxSupply,
                "Purchase exceed max supply."
            );
        }

        if (state == SaleState.PrivateSaleDuring) {
            require(isWhiteListed(signature), "Not whitelisted.");
            require(amount <= 2, "Mint exceed transaction limits");
            require(
                _privateSaleClaimed[msg.sender] + amount <= 2,
                "Mint limit per wallet exceeded."
            );
            require(
                totalPrivateSaleMinted().add(amount) <= privateSaleCapped,
                "Purchase exceed sale capped."
            );
        }

```

```
    if (
        state == SaleState.PrivateSaleDuring ||
        state == SaleState.PublicSaleDuring ||
        state == SaleState.DutchAuctionDuring
    ) {
        _mintToken(msg.sender, amount);
        if (state == SaleState.DutchAuctionDuring) {
            saleStats.totalDAMinted = saleStats.totalDAMinted.add(amount);

            uint256 mintPrice =  msg.value.div(amount);

            fairDAInfo[msg.sender].push(
                MintInfo(uint128(mintPrice), uint8(amount))
            );

            if (mintPrice < finalDAPrice) {
                finalDAPrice = mintPrice;
            }

            _dutchAuctionMinted[msg.sender] =
                _dutchAuctionMinted[msg.sender] +
                amount;
        }
        if (state == SaleState.PublicSaleDuring) {
            saleStats.totalFMMinted = saleStats.totalFMMinted.add(amount);
        }
        if (state == SaleState.PrivateSaleDuring) {
            _privateSaleClaimed[msg.sender] =
                _privateSaleClaimed[msg.sender] +
                amount;
            saleStats.totalWLMinted = saleStats.totalWLMinted.add(amount);
        }
        payable(_splitter).transfer(msg.value);
    }

    return true;
}
```

5.6.2. Remediation

Inspex suggests removing the conditions that are considered redundant.

For example, the `j <= maxSupply` condition in line 102 can be removed.

Revealable.sol

```
80 function getShuffledId(
81     uint256 totalSupply,
```

```
    uint256 maxSupply,
    uint256 tokenId,
    uint256 startIndex
) public view returns (string memory) {
    if (_msgSender() != owner()) {
        require(tokenId <= totalSupply, "Token not exists");
    }

    if (!isRevealed()) return "default";

    uint256[] memory metadata = new uint256[](maxSupply + 1);

    for (uint256 i = 1; i <= maxSupply; i += 1) {
        metadata[i] = i;
    }

    for (uint256 i = startIndex; i <= maxSupply; i += 1) {
        uint256 j = (uint256(keccak256(abi.encode(seed, i))) %
            (maxSupply)) + 1;

        if (j >= startIndex) {
            (metadata[i], metadata[j]) = (metadata[j], metadata[i]);
        }
    }

    return Strings.toString(metadata[tokenId]);
}
```

In the `mintOg()` function, the `if (getState() == SaleState.PrivateSaleDuring)` condition can also be removed.

Otoro.sol

```
function mintOg(bytes calldata signature)
    external
    payable
    nonReentrant
    returns (bool)
{
    require(msg.sender == tx.origin, "Contract is not allowed.");
    require(
        getState() == SaleState.PrivateSaleDuring,
        "Sale not available."
    );

    require(isOG(signature), "Not OG whitelisted.");
    require(_ogClaimed[msg.sender] == 0, "Already Claimed OG.");
    require(
```

```
130         totalPrivateSaleMinted().add(1) <= privateSaleCapped,
131         "Exceed Private Sale Limit"
132     );
133 
134     require(msg.value >= getPriceByMode(), "Insufficient funds.");
135 
136     _ogClaimed[msg.sender] = _ogClaimed[msg.sender] + 1;
137     saleStats.totalOGMinted = saleStats.totalOGMinted.add(1);
138 
139     _mintToken(msg.sender, 1);
140 
141     payable(_splitter).transfer(msg.value);
142 
143     return true;
144 }
```

And in the `mintToken()` function, the `if` condition from line 209 to line 213 can be removed, including the `return false` at the end of the function.

Otoro.sol

```
150 function mintToken(uint256 amount, bytes calldata signature)
151     external
152     payable
153     nonReentrant
154     returns (bool)
155 {
156     SaleState state = getState();
157     require(msg.sender == tx.origin, "Contract is not allowed.");
158     require(
159         state == SaleState.PrivateSaleDuring ||
160             state == SaleState.PublicSaleDuring ||
161             state == SaleState.DutchAuctionDuring,
162         "Sale not available."
163     );
164     require(
165         msg.value >= amount.mul(getPriceByMode()),
166         "Insufficient funds."
167     );
168 
169     if (state == SaleState.DutchAuctionDuring) {
170         require(
171             amount <= saleConfig.maxDAMintPerWallet,
172             "Mint exceed transaction limits."
173         );
174         require(
175             _dutchAuctionMinted[msg.sender] + amount <=
176                 saleConfig.maxDAMintPerWallet,
```

```
            "Mint limit per wallet exceeded."
        );
        require(
            saleStats.totalDAMinted.add(amount) <= dutchAuctionCapped,
            "Purchase exceed limit."
        );
    }

    if (state == SaleState.PublicSaleDuring) {
        require(
            amount <= saleConfig.maxFMMintPerTx,
            "Mint exceed transaction limits."
        );
        require(
            totalSupply().add(amount).add(availableReserve()) <= maxSupply,
            "Purchase exceed max supply."
        );
    }

    if (state == SaleState.PrivateSaleDuring) {
        require(isWhiteListed(signature), "Not whitelisted.");
        require(amount <= 2, "Mint exceed transaction limits");
        require(
            _privateSaleClaimed[msg.sender] + amount <= 2,
            "Mint limit per wallet exceeded."
        );
        require(
            totalPrivateSaleMinted().add(amount) <= privateSaleCapped,
            "Purchase exceed sale capped."
        );
    }

    _mintToken(msg.sender, amount);
    if (state == SaleState.DutchAuctionDuring) {
        saleStats.totalDAMinted = saleStats.totalDAMinted.add(amount);

        uint256 mintPrice =  msg.value.div(amount);

        fairDAInfo[msg.sender].push(
            MintInfo(uint128(mintPrice), uint8(amount))
        );

        if (mintPrice < finalDAPrice) {
            finalDAPrice = mintPrice;
        }

        _dutchAuctionMinted[msg.sender] =
```

```
                _dutchAuctionMinted[msg.sender] +
                amount;
        }
        if (state == SaleState.PublicSaleDuring) {
            saleStats.totalFMMinted = saleStats.totalFMMinted.add(amount);
        }
        if (state == SaleState.PrivateSaleDuring) {
            _privateSaleClaimed[msg.sender] =
                _privateSaleClaimed[msg.sender] +
                amount;
            saleStats.totalWLMinted = saleStats.totalWLMinted.add(amount);
        }
        payable(_splitter).transfer(msg.value);


        return true;
    }
```

6. Appendix

6.1. About Inspex

CYBERSECURITY
PROFESSIONAL
SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspex

CYBERSECURITY PROFESSIONAL SERVICE