0xStandard, BlockBasedSale, OGBlockBasedSale, Whitelisting & Wagyu

Smart Contract Audit Report
Prepared for 0xStudio

Date Issued:	Apr 15, 2022
Project ID:	AUDIT2022025
Version:	v1.0
Confidentiality Level:	Public

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022025
Version	v1.0
Client	0xStudio
Project	0xStandard, BlockBasedSale, OGBlockBasedSale, Whitelisting & Wagyu
Auditor(s)	Peeraphut Punsuwan Ronnachai chaipha Fungkiat Phadejtaku
Author(s)	Peeraphut Punsuwan
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Apr 15, 2022	Full report	Peeraphut Punsuwan

Contact Information

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

Table of Contents

1. Executive Summary **1**
1.1. Audit Result 1
1.2. Disclaimer 1
2. Project Overview **2**
2.1. Project Introduction 2
2.2. Scope 3
3. Methodology **4**
3.1. Test Categories 4
3.2. Audit Items 5
3.3. Risk Rating 7
4. Summary of Findings **8**
5. Detailed Findings Information **10**
5.1. Incorrect Token Minting Amount 10
5.2. Centralized Control of State Variable 14
5.3. Improper Setting of Hashed Secret 18
5.4. Insufficient Logging for Privileged Functions 21
5.5. Unnecessary Condition Checking 23
5.6. Improper Account Type Checking 25
6. Appendix **29**
6.1. About Inspex 29

1. Executive Summary

As requested by 0xStudio, Inspex team conducted an audit to verify the security posture of the 0xStandard, BlockBasedSale, OGBlockBasedSale, Whitelisting & Wagyu smart contracts between Apr 12, 2022 and Apr 13, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of 0xStandard, BlockBasedSale, OGBlockBasedSale, Whitelisting & Wagyu 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 1 high, 1 low, 2 very low, and 2 info-severity issues. With the project team's prompt response 1 high, 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 0xStandard, BlockBasedSale, OGBlockBasedSale, Whitelisting & Wagyu 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 helps users to design, develop, support, and deploy fully-functioned Web3 applications and smart contracts in order to achieve their objectives.

The 0xStandard is an NFT distributor that facilitates the platform owners to distribute their NFTs in multiple ways including private sale, public sale, or airdrop.

Scope Information:

Project Name	0xStandard, BlockBasedSale, OGBlockBasedSale, Whitelisting & Wagyu
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	Apr 12, 2022 - Apr 13, 2022
Reassessment Date	Apr 15, 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: 92b2888a7c9ff4a7e00d992fd4e9a46c6881ef32)

Contract	Location (URL)
0xStandardV2	https://github.com/0xstudio/0xContract-audit/blob/92b2888a7c/contracts/libs/0xStandardV2.sol
BlockBasedSale	https://github.com/0xstudio/0xContract-audit/blob/92b2888a7c/contracts/libs/BlockBasedSale.sol
EIP712Whitelisting	https://github.com/0xstudio/0xContract-audit/blob/92b2888a7c/contracts/libs/EIP712Whitelisting.sol
WagyuV2	https://github.com/0xstudio/0xContract-audit/blob/92b2888a7c/contracts/WagyuV2.sol
OGBlockBasedSale	https://github.com/0xstudio/0xContract-audit/blob/92b2888a7c/contracts/libs/OGBlockBasedSale.sol

Reassessment: (Commit: 9676218af4c9a245147e012b167bbf73b51da5f7)

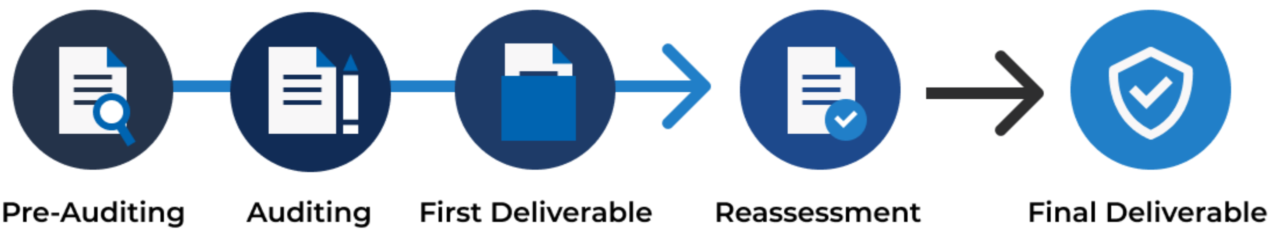
Contract	Location (URL)
0xStandardV2	https://github.com/0xstudio/0xContract-audit/blob/9676218af4/contracts/libs/0xStandardV2.sol
BlockBasedSale	https://github.com/0xstudio/0xContract-audit/blob/9676218af4/contracts/libs/BlockBasedSale.sol
EIP712Whitelisting	https://github.com/0xstudio/0xContract-audit/blob/9676218af4/contracts/libs/EIP712Whitelisting.sol
WagyuV2	https://github.com/0xstudio/0xContract-audit/blob/9676218af4/contracts/WagyuV2.sol
OGBlockBasedSale	https://github.com/0xstudio/0xContract-audit/blob/9676218af4/contracts/libs/OGBlockBasedSale.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 following audit items were checked during the auditing activity.

General
Smart Contract with Unpublished Source Code
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Use of Upgradable Contract Design
Centralized Control of State Variable
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control

Broken Authentication
Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

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**.

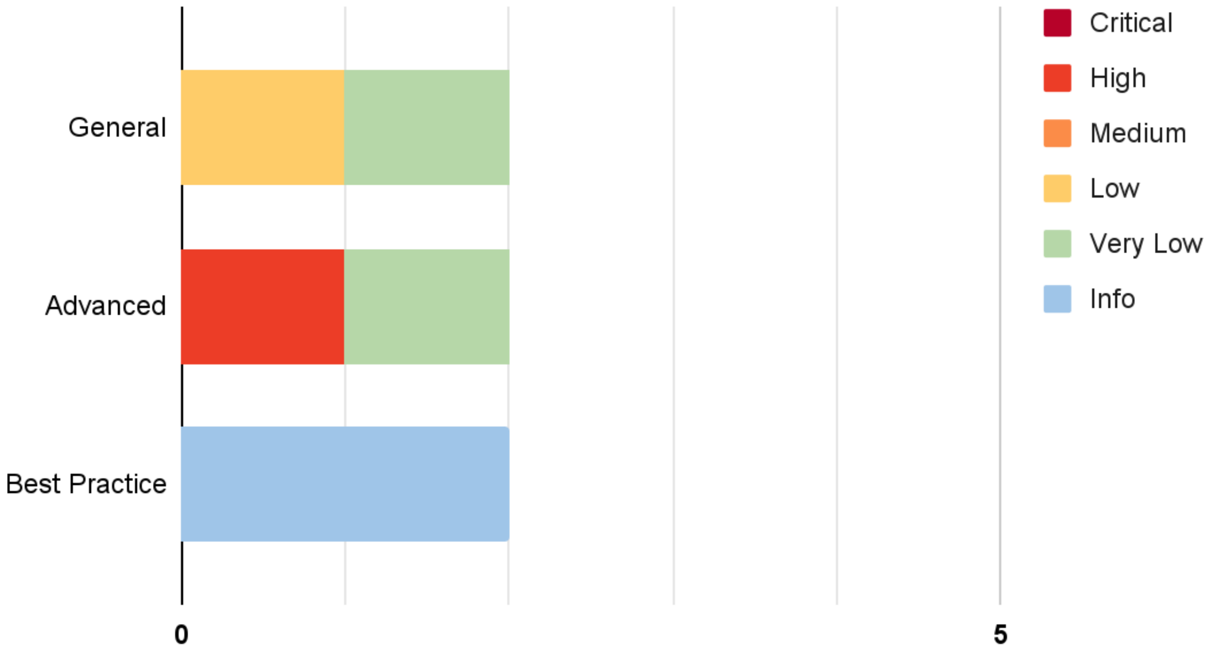
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

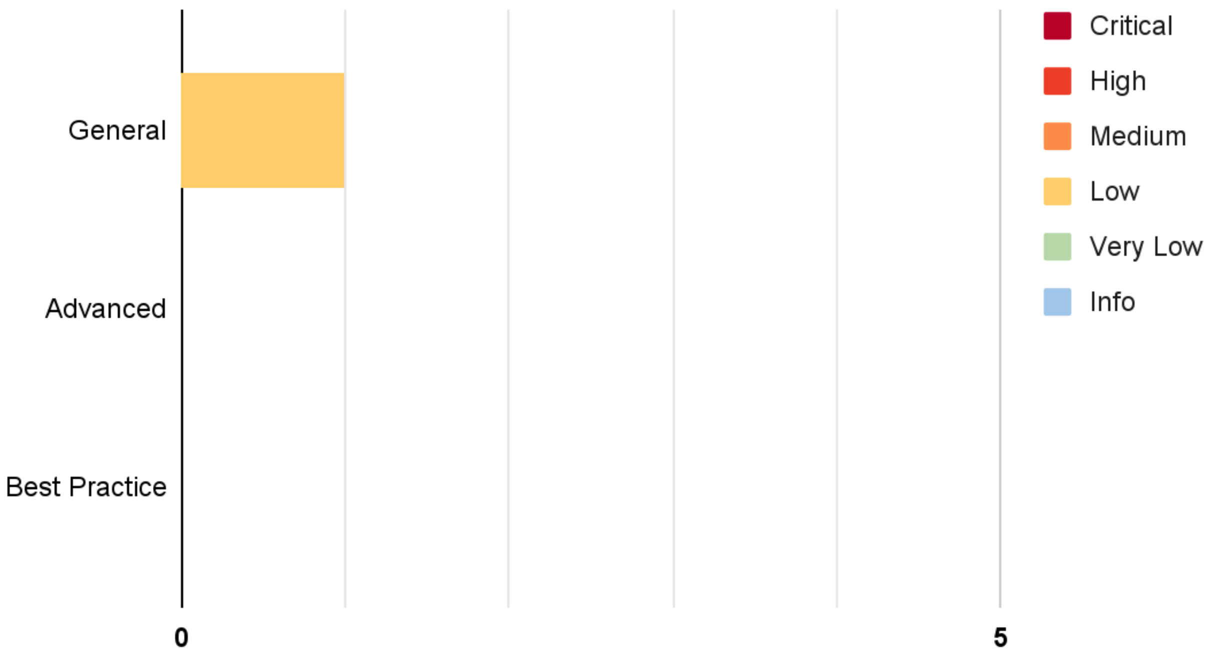
4. Summary of Findings

From the assessments, Inspex has found 6 issues in three categories. 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	Incorrect Token Minting Amount	Advanced	High	Resolved
IDX-002	Centralized Control of State Variable	General	Low	Acknowledged
IDX-003	Improper Setting of Hashed Secret	Advanced	Very Low	Resolved
IDX-004	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-005	Unnecessary Condition Checking	Best Practice	Info	Resolved
IDX-006	Improper Account Type Checking	Best Practice	Info	Resolved

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

5. Detailed Findings Information

5.1. Incorrect Token Minting Amount

ID	IDX-001
Target	WagyuV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High The user will receive the token amount less than it should be. **Likelihood:** Medium This issue will have an impact only when the `maxPublicSalePerTx` state is set to be greater than 1. However, the `maxPublicSalePerTx` state is set to 1 by default.
Status	Resolved The 0xStudio team has resolved this issue as suggested in the commit `9676218af4c9a245147e012b167bbf73b51da5f7`.

5.1.1. Description

The maximum token purchased amount depends on the `maxSaleCapped` state at line 140.

However, when the operator sets `maxPublicSalePerTx` state to be greater than 1. users can input the token amount more than the `maxSaleCapped` in the first time of purchasing. Then, users will receive the token amount less than it should be. As the `mintToken()` function always forwards `amount` parameter as 1 to `_mintToken()` function in line 147 as shown in the following source code:

WagyuV2.sol

```
112 function mintToken(uint256 amount, bytes calldata signature)
113     external
114     payable
115     nonReentrant
116     returns (bool)
117 {
118     require(!msg.sender.isContract(), "Contract is not allowed.");
119     require(
120         getState() == SaleState.PublicSaleDuring,
121         "Sale not available."
122     );
123 
124     if (getState() == SaleState.PublicSaleDuring) {
```

```
            require(
                amount <= maxPublicSalePerTx,
                "Mint exceed transaction limits."
            );
            require(
                msg.value >= amount.mul(getPriceByMode()),
                "Insufficient funds."
            );
            require(
                totalSupply().add(amount).add(availableReserve()) <= maxSupply,
                "Purchase exceed max supply."
            );
        }

        require(
            purchaseCount[msg.sender] < maxSaleCapped,
            "Max purchase reached"
        );

        emit MintAttempt(msg.sender, signature);

        if (getState() == SaleState.PublicSaleDuring) {
            _mintToken(msg.sender, 1);
            totalPublicMinted = totalPublicMinted + amount;
            if (isSubsequenceSale()) {
                nextSubsequentSale = block.number + subsequentSaleBlockSize;
            }
            payable(_splitter).transfer(msg.value);
        }

        return true;
    }
```

5.1.2. Remediation

Inspex suggests adding the following code to the `mintToken()` function:

- Add the purchased amount to the `purchseCount[msg.sender]` before validating with `maxSaleCapped` at line 140 in order to fix incorrect cap validation issue.
- Forward `amount` parameter to `_mintToken()` function at line 147 in order to resolve incorrect minting amount issue.

For example:

WagyuV2.sol

```
112 function mintToken(uint256 amount, bytes calldata signature)
113     external
114     payable
115     nonReentrant
116     returns (bool)
117 {
118     require(!msg.sender.isContract(), "Contract is not allowed.");
119     require(
120         getState() == SaleState.PublicSaleDuring,
121         "Sale not available."
122     );
123 
124     if (getState() == SaleState.PublicSaleDuring) {
125         require(
126             amount <= maxPublicSalePerTx,
127             "Mint exceed transaction limits."
128         );
129         require(
130             msg.value >= amount.mul(getPriceByMode()),
131             "Insufficient funds."
132         );
133         require(
134             totalSupply().add(amount).add(availableReserve()) <= maxSupply,
135             "Purchase exceed max supply."
136         );
137     }
138 
139     require(
140         purchaseCount[msg.sender] + amount <= maxSaleCapped,
141         "Max purchase reached"
142     );
143 
144     emit MintAttempt(msg.sender, signature);
145 
146     if (getState() == SaleState.PublicSaleDuring) {
```

```
        _mintToken(msg.sender, amount);
        totalPublicMinted = totalPublicMinted + amount;
        if (isSubsequenceSale()) {
            nextSubsequentSale = block.number + subsequentSaleBlockSize;
        }
        payable(_splitter).transfer(msg.value);
    }

    return true;
}
```

5.2. Centralized Control of State Variable

ID	IDX-002
Target	0xStandard BlockBasedSale WagyuV2 OGBlockBasedSale EIP712Whitelisting
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Low **Impact:** Medium The controlling authorities can change the state variables to gain unfair advantages, but does not directly impact the other users. **Likelihood:** Low There is nothing to restrict the changes from being done; however, this action can only be done by the privileged roles and the state control does not provide direct benefit for the privileged parties.
Status	**Acknowledged** The 0xStudio team has acknowledged this issue and clarified that the contract aims to operate sale at most sufficient capability. The 0xstudio team mitigates the operation by splitting roles and responsibilities between the operator and governor. - The operator role will be operated by the operation team with multisig wallet to ensure state changes are aligned across stakeholders yet maintain a pace of sale operation. These sale operations cover necessary sale & marketing features e.g. start sale/suspend sale/resume sale/change sale blocks/change dutch auction params which need to compete with time. - The governor role will operate by a team to secure metadata and art and ensure the preservation of NFT values (art itself). The timelock will be applied after the sale is completed, and the reveal stage is finished to ensure that the change of NFT metadata will be maintained properly and inform to the community.

5.2.1. Description

The state variables can be updated at any time by the controlling authorities. Changes in these variables can cause an impact 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.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
0xStandardV2.sol (L:142)	0xStandardV2	setAirdropRole()	onlyOwner
0xStandardV2.sol (L:618)	0xStandardV2	release()	shareHolderOnly
0xStandardV2.sol (L:627)	0xStandardV2	enableDutchAuction()	operatorOnly
0xStandardV2.sol (L:638)	0xStandardV2	disableDutchAuction()	operatorOnly
BlockBasedSale.sol (L:94)	0xStandardV2	setOperatorAddress()	onlyOwner
BlockBasedSale.sol (L:100)	0xStandardV2	setGovernorAddress()	onlyOwner
BlockBasedSale.sol (L:106)	0xStandardV2	setDiscountBlockSize()	operatorOnly
BlockBasedSale.sol (L:111)	0xStandardV2	setPriceDecayParams()	operatorOnly
BlockBasedSale.sol (L:122)	0xStandardV2	setTransactionLimit()	operatorOnly
BlockBasedSale.sol (L:140)	0xStandardV2	setPrivateSaleConfig()	operatorOnly
BlockBasedSale.sol (L:162)	0xStandardV2	setPublicSalePrice()	operatorOnly
BlockBasedSale.sol (L:167)	0xStandardV2	setPrivateSalePrice()	operatorOnly
BlockBasedSale.sol (L:187)	0xStandardV2	setReserve()	operatorOnly
BlockBasedSale.sol (L:192)	0xStandardV2	setPrivateSaleCap()	operatorOnly
WagyuV2.sol (L:77)	WagyuV2	airdrop()	airdropRoleOnly
WagyuV2.sol (L:102)	WagyuV2	setAirdropRole()	airdropRoleOnly
WagyuV2.sol (L:158)	WagyuV2	setBaseURI()	onlyOwner
WagyuV2.sol (L:204)	WagyuV2	release()	shareHolderOnly
setOperatorAddress.sol (L:107)	WagyuV2	setOperatorAddress()	onlyOwner
setOperatorAddress.sol (L:113)	WagyuV2	setGovernorAddress()	onlyOwner
setOperatorAddress.sol (L:119)	WagyuV2	setDiscountBlockSize()	operatorOnly
setOperatorAddress.sol (L:124)	WagyuV2	setPriceDecayParams()	operatorOnly
setOperatorAddress.sol (L:135)	WagyuV2	setTransactionLimit()	operatorOnly
setOperatorAddress.sol (L:155)	WagyuV2	setPublicSalePrice()	operatorOnly

setOperatorAddress.sol (L:175)	WagyuV2	setReserve()	operatorOnly
setOperatorAddress.sol (L:274)	WagyuV2	setPublicSaleCap()	operatorOnly
setOperatorAddress.sol (L:333)	WagyuV2	enableDutchAuction()	operatorOnly
setOperatorAddress.sol (L:338)	WagyuV2	disableDutchAuction()	operatorOnly
EIP712Whitelisting.sol (L:47)	EIP712Whitelisting	setWhitelistSigningAddress()	operatorOnly
EIP712Whitelisting.sol (L:54)	EIP712Whitelisting	setOgSigningAddress()	operatorOnly

Please note that, for the governer role, if the owner uses the DAO mechanism for `governerOnly` modifier, the following function will not affect the issue:

File	Contract	Function	Modifier
0xStandardV2.sol (L:236)	0xStandardV2	setBaseURI()	governerOnly
0xStandardV2.sol (L:570)	0xStandardV2	setBlockNumbertoGenSeed()	governerOnly
0xStandardV2.sol (L:591)	0xStandardV2	setRandomResultToSeed()	governerOnly
0xStandardV2.sol (L:627)	0xStandardV2	withdraw()	governerOnly
WagyuV2.sol (L:213)	WagyuV2	withdraw()	governerOnly

Finally, there are functions with operator role that has no direct impact to the users and can help the owner to easily follow the business plan. Therefore, Inspex will not include the following functions to this issue.

File	Contract	Function	Modifier
0xStandardV2.sol (L:246)	0xStandardV2	setDefaultURI()	operatorOnly
0xStandardV2.sol (L:162)	0xStandardV2	setRevealBlock()	operatorOnly
0xStandardV2.sol (L:291)	0xStandardV2	requestChainlinkVRF()	operatorOnly
BlockBasedSale.sol (L:172)	0xStandardV2	setCloseSale()	operatorOnly
BlockBasedSale.sol (L:177)	0xStandardV2	setPauseSale()	operatorOnly
BlockBasedSale.sol (L:197)	0xStandardV2	enablePublicSale()	operatorOnly
BlockBasedSale.sol (L:202)	0xStandardV2	enablePrivateSale()	operatorOnly
BlockBasedSale.sol (L:182)	0xStandardV2	resetOverridedSaleState()	operatorOnly
WagyuV2.sol (L:107)	WagyuV2	setRevealBlock()	operatorOnly

OGBlockBasedSale.sol (L:144)	WagyuV2	setPublicSaleConfig()	operatorOnly
OGBlockBasedSale.sol (L:160)	WagyuV2	setCloseSale()	operatorOnly
OGBlockBasedSale.sol (L:165)	WagyuV2	setPauseSale()	operatorOnly
OGBlockBasedSale.sol (L:184)	WagyuV2	enablePublicSale()	operatorOnly
OGBlockBasedSale.sol (L:170)	WagyuV2	resetOverridedSaleState()	operatorOnly
OGBlockBasedSale.sol (L:189)	WagyuV2	setSubsequentSaleBlock()	operatorOnly

5.2.2. Remediation

In the ideal case, the state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions.
- Using a timelock mechanism to delay the changes for a reasonable amount of time, e.g., 24 hours.

5.3. Improper Setting of Hashed Secret

ID	IDX-003
Target	OxStandardV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low The attacker will know the correct secret that is used in the `setRandomResultToSeed()` function, allowing the random result to be calculated and let the attacker withhold the block if it is not desirable. **Likelihood:** Low Only the miner can use this flaw to perform the attack, and the loss of the block reward may not worth the value gained from the withholding of the block.
Status	**Resolved** The 0xStudio team has resolved this issue as suggested in the commit `9676218af4c9a245147e012b167bbf73b51da5f7`.

5.3.1. Description

The `setBlockNumbertoGenSeed()` function is used for setting the settlement block number and the hash of the secret to random the `seed`. The hash of the settlement block together with the secret will be used to generate the random result for the value of `seed` in the `setRandomResultToSeed()` function.

0xStandardV2.sol

```
function setBlockNumbertoGenSeed(bytes32 _hashedSecret)
    external
    governerOnly
{
    require(
        bytes(_tokenBaseURI).length != 0,
        "The token base URI is not set yet"
    );
    require(!randomseedRequested, "The random already requested");
    require(
        settlementBlockNumber == 0 ||
            block.number - settlementBlockNumber >= 256,
        "settlementBlockNumber block is already set"
    );

    //set settlementBlockNumber to the future block
```

```
586         settlementBlockNumber = block.number + 10;
587         hashedSecret = keccak256(abi.encodePacked(_hashedSecret));
588         emit AssignSettlementBlockNumber(settlementBlockNumber);
589     }
590 
591     function setRandomResultToSeed(bytes32 _secret) external governerOnly {
592         require(
593             settlementBlockNumber != 0,
594             "Settlement block number not exists"
595         );
596         require(
597             block.number > settlementBlockNumber,
598             "Settlement block number not reached"
599         );
600         require(
601             block.number - settlementBlockNumber < 256,
602             "Settlement block number expired."
603         );
604         require(
605             keccak256(abi.encodePacked(_secret)) == hashedSecret,
606             "Incorrect secret"
607         );
608 
609         seed = uint256(
610             keccak256(
611                 abi.encodePacked(blockhash(settlementBlockNumber), _secret)
612             )
613         );
614         randomseedRequested = true;
615         emit AssignRandomNess(seed);
616     }
```

The `_hashedSecret` will be used to check the validity of the secret used; however, the `_hashedSecret` parameter is hashed again before being stored in the `hashedSecret` state as seen in line 587. Therefore, to execute the `setRandomResultToSeed()` function successfully and pass the validation condition in line 605, the `_secret` must be the same as `_hashedSecret`. This means that the miners will know the `_secret` to be used and can pre-calculate the random result with the knowledge of the secret and the block hash.

This knowledge of the secret allows the miner to withhold the block if the random result is not desirable to the miner.

5.3.2. Remediation

Inspex suggests saving the value of `_hashedSecret` parameter to the `hashedSecret` state as is, without hashing that parameter again, as shown in the code snippet below at line 587. This will prevent anyone other than the governor role from having the knowledge of the correct `_secret`.

0xStandardV2.sol

```
570 function setBlockNumbertoGenSeed(bytes32 _hashedSecret)
571     external
572     governerOnly
573 {
574     require(
575         bytes(_tokenBaseURI).length != 0,
576         "The token base URI is not set yet"
577     );
578     require(!randomseedRequested, "The random already requested");
579     require(
580         settlementBlockNumber == 0 ||
581             block.number - settlementBlockNumber >= 256,
582         "settlementBlockNumber block is already set"
583     );
584 
585     //set settlementBlockNumber to the future block
586     settlementBlockNumber = block.number + 10;
587     hashedSecret = _hashedSecret;
588     emit AssignSettlementBlockNumber(settlementBlockNumber);
589 }
```

The value of the `_hashedSecret` should be the keccak256 hash of the `_secret` that will be used in the `setRandomResultToSeed()` function.

For example:

`_hashedSecret`: 0x1fc437c6d3cce86063a049107645c87a6c11534dffc711b9cdfd8beccb5c86c5
`_secret`: 0xfa26db7ca85ead399216e7c6316bc50ed24393c3122b582735e7f3b0f91b93f0

5.4. Insufficient Logging for Privileged Functions

ID	IDX-004
Target	0xStandardV2 BlockBasedSale WagyuV2 OGBlockBasedSale EIP712Whitelisting
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 as suggested in the commit `9676218af4c9a245147e012b167bbf73b51da5f7`.

5.4.1. 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 could release the airdrop by executing the `setOperatorAddress()` function in the `0xStandardV2` contract, and no events are emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function
BlockBasedSale.sol (L:94)	0xStandardV2	setOperatorAddress()
BlockBasedSale.sol (L:100)	0xStandardV2	setGovernorAddress()
EIP712Whitelisting.sol (L:47)	0xStandardV2	setWhitelistSigningAddress()
EIP712Whitelisting.sol (L:54)	0xStandardV2	setOgSigningAddress()
OGBlockBasedSale.sol (L:107)	WagyuV2	setOperatorAddress()

OGBlockBasedSale.sol (L:113)	WagyuV2	setGovernorAddress()

5.4.2. Remediation

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

BlockBasedSale.sol

```
event OperatorAddress(address _operator);
function setOperatorAddress(address _operator) external onlyOwner {
    require(_operator != address(0));
    operatorAddress = _operator;
    operatorAssigned = true;
    emit OperatorAddress(_operator);
}
```

5.5. Unnecessary Condition Checking

ID	IDX-005
Target	OGBlockBasedSale
Category	Advanced Smart Contract Vulnerability
CWE	CWE-1164: Irrelevant Code
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The 0xStudio team has resolved this issue as suggested in the commit `9676218af4c9a245147e012b167bbf73b51da5f7`.

5.5.1. Description

Some conditions are not unnecessary to implement and these could lead to the consumption of more gas. The `setPriceDecayParams()` function has unnecessary conditions that was implemented on the `OGBlockBasedSale` contract. For example, in line 128, the `_lowerBoundPrice` variable type is `uint` so its value is impossible to be lower than 0. Thus, this condition is unnecessary.

OGBlockBasedSale.sol

```
124 function setPriceDecayParams(uint256 _lowerBoundPrice, uint256 _priceFactor)
125     external
126     operatorOnly
127 {
128     require(_lowerBoundPrice >= 0);
129     require(_priceFactor <= publicSalePrice);
130     lowerBoundPrice = _lowerBoundPrice;
131     priceFactor = _priceFactor;
132     emit AssignPriceDecayParameter(_lowerBoundPrice, _priceFactor);
133 }
```

5.5.2. Remediation

Inspex suggests removing all unneccessary conditions, as shown in the following code snippet:

OGBlockBasedSale.sol

```
function setPriceDecayParams(uint256 _lowerBoundPrice, uint256 _priceFactor)
    external
    operatorOnly
{
    require(_priceFactor <= publicSalePrice);
    lowerBoundPrice = _lowerBoundPrice;
    priceFactor = _priceFactor;
    emit AssignPriceDecayParameter(_lowerBoundPrice, _priceFactor);
}
```

5.6. Improper Account Type Checking

ID	IDX-006
Target	0xStandardV2 WagyuV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The 0xStudio team has resolved this issue as suggested in the commit `9676218af4c9a245147e012b167bbf73b51da5f7`.

5.6.1. Description

The `mintOg()` function of `0xStandardV2` contract checks whether the `msg.sender` is a smart contract or not by using `Address.isContract()` function as shown below:

0xStandardV2.sol

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

      if (getState() == SaleState.PrivateSaleDuring) {
          require(isOGwhitelisted(signature), "Not OG whitelisted.");
          require(_ogClaimed[msg.sender] == 0, "Already Claimed OG.");
          require(
              totalPrivateSaleMinted.add(1) <= privateSaleCapped,
              "Purchase exceed private sale capped."
          );

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

```
            emit OGClaim(msg.sender);
            _ogClaimed[msg.sender] = _ogClaimed[msg.sender] + 1;
            totalPrivateSaleMinted = totalPrivateSaleMinted + 1;
            totalOGClaimed = totalOGClaimed + 1;

            _mintToken(msg.sender, 1);

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

            return true;
        }

        return false;
    }
```

The `Address.isContract()` function checks `EXTCODESIZE` opcode which returns the size of the contract's bytecode of an address. If the size is larger than zero, the address is a contract.

Address.sol

```
function isContract(address account) internal view returns (bool) {
    // This method relies on extcodesize/address.code.length, which returns 0
    // for contracts in construction, since the code is only stored at the end
    // of the constructor execution.

    return account.code.length > 0;
}
```

However, the bytecode will be stored at the end of the `constructor` function call. Therefore, calling the affected functions from within the constructor will cause the `EXTCODESIZE` to return 0. As a result, the `Address.isContract()` can return false when calling from the constructor function of a newly deployed contract.

The following code is an example of contract that can bypass the condition check in the `mintOg()` function:

BypassMintOg.sol

```
contract BypassMintOg {

    I0xStandardV2 public 0xStandardV2;

    constructor(I0xStandardV2 _0xStandardV2) {
        0xStandardV2 = _0xStandardV2;
        0xStandardV2.mintOg();
    }
}
```

The following table contains all improper Account Type Checking:

File	Contract	Function
0xStandardV2.sol (L:173)	0xStandardV2	mintOg()
0xStandardV2.sol (L:210)	0xStandardV2	mintToken()
WagyuV2.sol (L:112)	WagyuV2	mintToken()

5.6.2. Remediation

Inspex suggests checking that the caller is the smart contract or not by comparing `msg.sender` with `tx.origin`. The `tx.origin` returns the transaction creator address. If the `tx.origin` is not equal to `msg.sender`, the caller will not be an externally-owned account (EOA), for example, as shown in line 179:

0xStandardV2.sol

```
173 function mintOg(bytes calldata signature)
174     external
175     payable
176     nonReentrant
177     returns (bool)
178 {
179     require(msg.sender == tx.origin , "Allow non-contract only");
180     require(
181         getState() == SaleState.PrivateSaleDuring,
182         "Sale not available."
183     );
184
185     if (getState() == SaleState.PrivateSaleDuring) {
186         require(isOGwhitelisted(signature), "Not OG whitelisted.");
187         require(_ogClaimed[msg.sender] == 0, "Already Claimed OG.");
188         require(
189             totalPrivateSaleMinted.add(1) <= privateSaleCapped,
190             "Purchase exceed private sale capped."
191         );
192
193         require(msg.value >= getPriceByMode(), "Insufficient funds.");
194
195         emit OGClaim(msg.sender);
196         _ogClaimed[msg.sender] = _ogClaimed[msg.sender] + 1;
197         totalPrivateSaleMinted = totalPrivateSaleMinted + 1;
198         totalOGClaimed = totalOGClaimed + 1;
199
200         _mintToken(msg.sender, 1);
201
202         payable(_splitter).transfer(msg.value);
```

```
        return true;
    }

    return false;
}
```

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

CYBERSECURITY PROFESSIONAL SERVICE