Cyberscope

Audit Report

StoicDAO

November 2023

Network GOERLI

Address 0x3a90a864fc9d6d4f837a19770e36c1af0e3e5aa1

Audited by © cyberscope

Table of Contents

Review

Contract Name	stoicDAO
Compiler Version	v0.8.23+commit.f704f362
Optimization	200 runs
Explorer	https://goerli.etherscan.io/address/0x3a90a864fc9d6d4f837a19770e36c1af0e3e5aa1
Address	0x3a90a864fc9d6d4f837a19770e36c1af0e3e5aa1
Network	GOERLI
Symbol	stoicDAO

Audit Updates

Initial Audit	27 Nov 2023

Source Files

Filename	SHA256
stoicDAO.sol	be86ebd8d646b8297da99b9cf950aac74741c7ff73d1c0ea7e5a7d7b5855fc6d

Overview

Our audit team has thoroughly examined the StoicDAO smart contract, which implements the ERC721A interface for a collection named "Senseless Stoics." The contract facilitates a whitelist presale, a public presale round, and the distribution of funds from NFT sales to predefined addresses using OpenZeppelin's PaymentSplitter contract.

Presale Mechanisms

- The contract incorporates a whitelist presale and a public presale round.
- Functions `whitelistMint()` and `publicSaleMint()` handle the minting process during these presale phases.
- Minting constraints, such as maximum supply, per-wallet limits, and total supply checks, are appropriately implemented.

Payment Distribution

- Utilizes OpenZeppelin's PaymentSplitter for the distribution of funds to predefined addresses.
- The `releaseAll()` method efficiently releases proportional payments to the predefined addresses based on the PaymentSplitter configuration.

Configurability

- Various parameters such as maximum supply, per-wallet limits, sale prices, and base URI are configurable by the contract owner.
- Dynamic adjustments of total supply and sale parameters enhance contract flexibility.

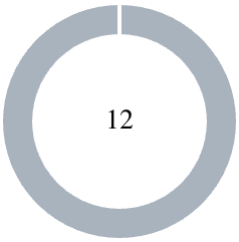
Security Measures

- Includes a modifier `callerIsUser()` to ensure that the caller is an externally-owned account, enhancing security.
- Implements checks to ensure that minting is performed with the participant's own wallet.

Whitelist Functionality

- Incorporates a whitelist mechanism with a merkle tree structure for efficient verification.
- The contract owner can set the merkle root for the whitelist.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	12

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	12	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	MEE	Missing Events Emission	Unresolved
●	RAP	Redundant Account Parameter	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	MPC	Merkle Proof Centralization	Unresolved
●	RTLS	Redundant Team Length Storage	Unresolved
●	PRER	Potential Release Execution Revert	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

MEE - Missing Events Emission

Criticality	Minor / Informative
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

- `whitelistMint()`
- `publicSaleMint()`
- `gift()`
- `lowerSupply()`
- `setMaxTotalPUBLIC()`
- `setMaxTotalWL()`
- `setMaxPerWalletWL()`
- `setMaxPerWalletPUBLIC()`
- `setWLSalePrice()`
- `setPublicSalePrice()`
- `setBaseUri()`
- `setStep()`
- `setMerkleRootWL()`
- `setRoyaltyInfo()`

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

RAP - Redundant Account Parameter

Criticality	Minor / Informative
Location	stoicDAO.sol#L2895,2908
Status	Unresolved

Description

Both functions `whitelistMint()` and `publicSaleMint()` include an argument named `_account`, which is intended to represent the address of the participant attempting to perform the minting operation. However, upon closer inspection, it appears that these functions are designed to be exclusively invoked by the transaction sender `msg.sender`. Consequently, the inclusion of the `_account` parameter seems redundant, as it can be inferred from the `msg.sender` value.

```
function whitelistMint(address _account, uint _quantity, bytes32[]
calldata _proof) external payable callerIsUser {
    require(sellingStep == Step.WhitelistSale, "Whitelist sale is
not activated");
    require(msg.sender == _account, "Mint with your own wallet.");
    ...
}
```

Recommendation

The team is adviced to remove the redundant `_account` parameter from the `whitelistMint()` and `publicSaleMint()` functions. Instead, directly reference the msg.sender within the functions to determine the participant's address.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Status	Unresolved

Description

The contract's functionality and behavior are dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion. The following methods depends on the owner's configuration:

- `gift()`
- `lowerSupply()`
- `setMaxTotalPUBLIC()`
- `setMaxTotalWL()`
- `setMaxPerWalletWL()`
- `setMaxPerWalletPUBLIC()`
- `setWLSalePrice()`
- `setPublicSalePrice()`
- `setBaseUri()`
- `setStep()`
- `setMerkleRootWL()`
- `releaseAll()`

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

The team is adviced to add meaningfull checks in the setters of those methods. For instance, the `MAX_TOTAL_WL` and `MAX_TOTAL_PUBLIC` variables should not be greater than the `MAX_SUPPLY`. The `MAX_SUPPLY` should not be less than the current `totalSupply()`.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	stoicDAO.sol#L2968
Status	Unresolved

Description

The contract uses a Merkle Proof mechanism in order to define many applicable addresses. The verification process is based on an off-chain configuration. The contract owner is responsible for updating the in-chain “Merkle Root” in order to validate correctly the provided message.

```
//Whitelist
function setMerkleRootWL(bytes32 _merkleRootWL) external onlyOwner
{
    merkleRootWL = _merkleRootWL;
}
...
require(isWhiteListed(_account, _proof), "Not whitelisted");
```

Recommendation

We state that the Merkle Proof algorithm is required for proper protocol operations and gas consumption decrease. Thus, we emphasize that the Merkle proof algorithm is based on an off-chain mechanism. Any off-chain mechanism could potentially be compromised and affect the on-chain state unexpectedly.

The team should carefully manage the private keys of the owner’s account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.

- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

RTLS - Redundant Team Length Storage

Criticality	Minor / Informative
Location	stoicDAO.sol#L3004
Status	Unresolved

Description

The contract uses the OpenZeppelin PaymentSplitter contract to manage the distribution of funds to predefined addresses. However, in the constructor, the total number of payees is stored in the state variable teamLength to facilitate the execution of the `releaseAll()` method. The redundancy arises from the fact that the PaymentSplitter contract provides a dynamic mechanism to query payees using the `payee(index)` function until the zero address is returned, indicating the end of the payees' list. Storing the total number of payees separately in the constructor is not necessary.

```
function releaseAll() external onlyOwner {
    for(uint i = 0 ; i < teamLength ; i++) {
        release(payable(payee(i)));
    }
}
```

Recommendation

Consider removing the redundant storage of the total number of payees in the constructor. Instead, rely on the dynamic querying mechanism provided by the PaymentSplitter contract to determine the number of payees during runtime. Utilize a loop that calls `payee(index)` incrementally until the zero address is returned, and use this information for the execution of the `releaseAll()` method.

PRER - Potential Release Execution Revert

Criticality	Minor / Informative
Location	stoicDAO.sol#L3004
Status	Unresolved

Description

The contract contains a method called "releaseAll()" where the predefined addresses receive the proportional payment. The users can also execute the "release()" method separately. If there are no releasable tokens, the "release()" method reverts. If one of these addresses releases the corresponding amount by calling the "release()" method prior to the "releaseAll()", then the "releaseAll()" will revert since the releasable amount for that user will be zero. Describe the following finding.

```
function releaseAll() external onlyOwner {
    for(uint i = 0 ; i < teamLength ; i++) {
        release(payable(payee(i)));
    }
}
...
require(payment != 0, "PaymentSplitter: account is not due
payment");
```

Recommendation

The team is adviced to check the "releaseAll()" function to exclude users with zero releasable amounts, ensuring that the function can execute successfully even if some users have individually released their funds before the "releaseAll()" call.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	stoicDAO.sol#L2881
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
teamLength
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	stoicDAO.sol#L413,433,444,473,480,487,495,538,548,555,565,617,825,973,987,1088,1191,1247,1254,1262,1271,1361,1368,1376,1387,1403,1487,1501,1537,1548,1590,1639,1652,1682,1692,1746,1753,1762,1781,1788,1889,1898,1929,2174,2182,2191,2206,2240,2290,2311,2325,2339,2358,2414
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function _nextTokenId() internal view returns (uint256) {
        return _currentIndex;
    }

function _totalMinted() internal view returns (uint256) {
        // Counter underflow is impossible as _currentIndex does
not decrement,
...
            return _currentIndex - _startTokenId();
        }
    }

function _totalBurned() internal view returns (uint256) {
        return _burnCounter;
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	stoicDAO.sol#L1449,1452,1464,1468,1469,1470,1471,1472,1473,1479,2991
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
function calculateRoyalty(uint256 _salePrice) view public returns (uint256){
    return(_salePrice / 10000) * royaltyFeesInBips;
}
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	stoicDAO.sol#L2995
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
ltyReceiver = _receiver;
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	stoicDAO.sol#L7,288,1238,1284,1702,1798,1960,2053,2135,2255,2490,2517,2733,2833
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.4;
pragma solidity ^0.8.20;
pragma solidity ^0.8.0;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	stoicDAO.sol#L7,288,1238,1284,1702,1798,1960,2053,2135,2255,2490,2517,2733,2833
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.4;
pragma solidity ^0.8.20;
pragma solidity ^0.8.0;
```

Recommendation

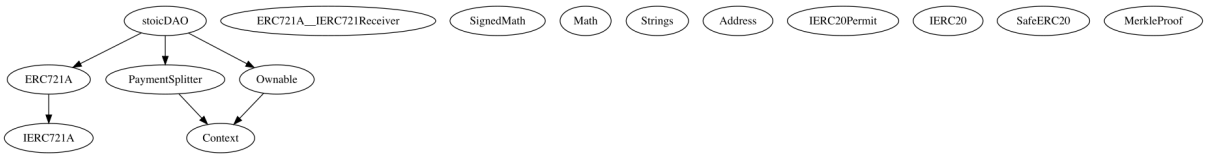
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

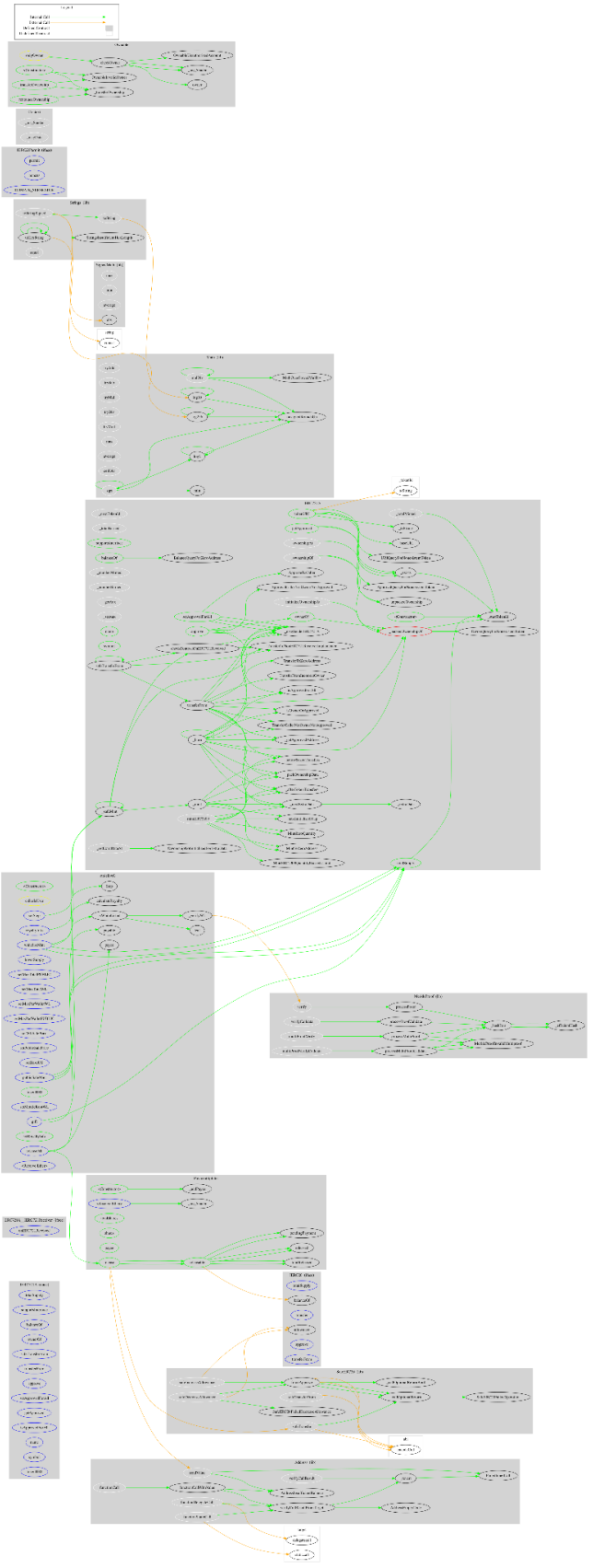
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
stoicDAO	Implementation	Ownable, ERC721A, PaymentSplitter		
		Public	✓	ERC721A PaymentSplitter Ownable
	whitelistMint	External	Payable	callerIsUser
	publicSaleMint	External	Payable	callerIsUser
	gift	External	✓	onlyOwner
	lowerSupply	External	✓	onlyOwner
	setMaxTotalPUBLIC	External	✓	onlyOwner
	setMaxTotalWL	External	✓	onlyOwner
	setMaxPerWalletWL	External	✓	onlyOwner
	setMaxPerWalletPUBLIC	External	✓	onlyOwner
	setWLSalePrice	External	✓	onlyOwner
	setPublicSalePrice	External	✓	onlyOwner
	setBaseUri	External	✓	onlyOwner
	setStep	External	✓	onlyOwner
	tokenURI	Public		-
	setMerkleRootWL	External	✓	onlyOwner
	isWhiteListed	Internal		
	leaf	Internal		

	_verifyWL	Internal		
	royaltyInfo	External		-
	calculateRoyalty	Public		-
	setRoyaltyInfo	Public	✓	onlyOwner
	releaseAll	External	✓	onlyOwner
		External	Payable	-

Inheritance Graph



Flow Graph



Summary

The StoicDAO smart contract demonstrates well-implemented features for a dynamic and configurable NFT collection. The contract's use of OpenZeppelin libraries enhances security and functionality. The identified findings are minor and can be addressed to further optimize the contract. Overall, the contract exhibits a thoughtful design and effective utilization of industry-standard practices.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io