Cyberscope

Audit Report

Rosy token

March 2024

Files	Deployer.sol, Orchestrator.sol, Steak.sol, Burnt.sol, Carbon.sol
Audited by	© cyberscope

Table of Contents

Review

Testing Deploy	https://testnet.bscscan.com/address/0x3c84998f99f7483cf5ee8d98793169ca2c8540e5

Audit Updates

Initial Audit	06 Mar 2024

Source Files

Filename	SHA256
contracts/Steak.sol	8f7db53e2b32a9949b81ee490c453b8d6d8fc11e1e581df486ac0a70fe626df8
contracts/Orchestrator.sol	a98236028af2598f06141209196a24cc4a8b59880be8597b8b9f80c50370a27a
contracts/IBurnableERC20.sol	88840fb03c11c18367476a286dcbd719081e5cf2a69bb4c523c2ae188558520a
contracts/Deployer.sol	54eb161131316dd656fdb6e780631a6237b3625ee15b87939367fd22586ff02c
contracts/Carbon.sol	055f4d08eff2985c49446f45ffd3d7cd289ad33574c4c4f216f4c9fab5d87952
contracts/Burnt.sol	b5248a13f5f3e1ef778553f52ee8f6add1ba2bdb07efd9b86b4ea85f013a5296
@openzeppelin/contracts/utils/Context.sol	847fda5460fee70f56f4200f59b82ae622bb03c79c77e67af010e31b7e2cc5b6
@openzeppelin/contracts/utils/math/Math.sol	a6ee779fc42e6bf01b5e6a963065706e882b016affbedfd8be19a71ea48e6e15

@openzeppelin/contracts/token/ERC20/IERC20.sol	6f2faae462e286e24e091d7718575179644dc60e79936ef0c92e2d1ab3ca3cee
@openzeppelin/contracts/interfaces/IERC20.sol	cb42f0b4d269ba8ef2629c176a7f99bf4fb50837c92f45596b54822b26e3df4b
@openzeppelin/contracts/access/Ownable2Step.sol	90f1f1cdd07ce4b90e987065e82899fdaa6ef967d1996915143c6e39818e160c
@openzeppelin/contracts/access/Ownable.sol	38578bd71c0a909840e67202db527cc6b4e6b437e0f39f0c909da32c1e30cb81
@oasisprotocol/sapphire-contracts/contracts/Sapphire.sol	7b04d3f2de70838e615786cb7fd49e08cbe117c3f42b3c81e024b950385bf484

Overview

Deployer.sol

The `BurntSteakDeployer` contract serves as the entry point for initializing the staking, burning, and rewards ecosystem of the project. It is responsible for deploying and setting up key components of the system, including the `Orchestrator`, `RandomMultiRewardEmitter`, and indirectly, the associated `Burnt`, `Steak`, and `Carbon` contracts through the `Orchestrator`. This contract sets the parameters for the ecosystem, such as the token to be used (rosyToken), burn thresholds, burn rates, and the rewards point rate. Upon deployment, it transfers ownership of the `Orchestrator` to the deployer.

Orchestrator.sol

The `Orchestrator` contract acts as the centre for managing the staking, burning, and rewards components of the project. It inherits from `Ownable2Step`, adding an extra layer of security for ownership transfers. This contract directly initializes and integrates the `Burnt`, `Steak`, and `Carbon` contracts, setting key parameters for each component based on the initial configuration passed during its own construction.

Access Control

Implements custom modifiers like onlySteak and publicBurnAllowed to enforce access control, ensuring that only authorized interactions occur.

Administration and Configuration

Provides functions for the contract owner to adjust key operational parameters such as burn thresholds, burn rates, rewards rates, and even the ability to enable or disable public token burning. It also allows for the management of component contract ownership.

Steak.sol

The `Steak` contract is dedicated to the staking functionality within the project, enabling users to stake and unstake tokens as part of their participation in the ecosystem. It is designed to work closely with the Orchestrator contract, signaling stake changes and interacting with other components of the system, particularly for the purpose of adjusting rewards and managing token burns. When stake changes are made, they are accompanied by the emission of `Staked` and `Unstaked` events for transparency and tracking.

Burnt.sol

The `Burnt` contract is designed to manage the burning of tokens within the ecosystem. It introduces a mechanism to burn tokens based on a calculated rate that can adjust over time, influenced by various factors within the system. This contract allows for a responsive approach to token burning. Parameters such as the `burnThreshold`, `baseBurnPerSecond`, `maxBurnPerSecond`, and `scaleFactor` can be adjusted by the contract owner. Emits events to provide transparency over the contract's actions.

Carbon.sol

The `Carbon` contract is integral to the rewards system of the project, focusing on the accumulation and redemption of points based on users' staking and unstakin. It provides a flexible framework for calculating user points over time and converting these points into rewards, facilitating an engaging user experience. Emits events like `PointsRedeemed` to offer transparency and traceability

Point Accumulation

Implements a mechanism for users to accumulate points over time, based on factors such as the duration of their stake. This is achieved through a combination of the user's points factor and the system's annual rate, allowing for dynamic rewards calculation.

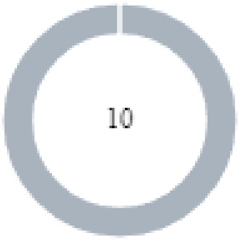
Reward Redemption

Offers users the ability to redeem their accumulated points for rewards. The actual redemption process is handled by `rewardEmitter`, which is responsible for determining the rewards given in exchange for points.

Audit Scope

The current audit report specifically focuses on the following contract files: `Deployer.sol` , `Orchestrator.sol` , `Steak.sol` , `Carbon.sol` , `Burnt.sol` . The `RandomMultiRewardEmitter.sol` is out of audit scope for the current audit phase. This means that while the provided contracts are thoroughly examined for security and functionality, any interactions, dependencies, or integrations with the aforementioned contract are not covered in this audit report. This limitation should be taken into consideration when interpreting the findings and conclusion of this audit.

Findings Breakdown



- Critical 0
- Medium 0
- Minor / Informative 10

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	10	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	CCR	Contract Centralization Risk	Unresolved
●	MC	Missing Check	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	TUU	Time Units Usage	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L19	Stable Compiler Version	Unresolved

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/Orchestrator.sol#L60,64,68,72,76,80,84...
Status	Unresolved

Description

The contract's functionality and behavior are heavily 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.

Specifically, the contract owner has the authoriry to set key variables, that impact the functionality of the contract. This capability grants the contract owner substantial control.

```
function setBurnInfluencingFactor(IBurnInfluencingFactor
_burnInfluencingFactor) external onlyOwner {
    burnt.setBurnInfluencingFactor(_burnInfluencingFactor);
}

function setburnThreshold(uint256 _burnThreshold) external
onlyOwner {
    burnt.setburnThreshold(_burnThreshold);
}

function setAllowPublicBurn(bool _allowPublicBurn) public
onlyOwner {
    allowPublicBurn = _allowPublicBurn;
}

...
```

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.

MC - Missing Check

Criticality	Minor / Informative
Location	contracts/Burnt.sol#L100,104
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues. Specifically, there is no check to ensure that `baseBurnPerSecond` is lower than `maxBurnPerSecond` .

```
function setBaseBurnPerSecond(uint256 _baseBurnPerSecond)
external onlyOwner {
    baseBurnPerSecond = _baseBurnPerSecond;
}

function setMaxBurnPerSecond(uint256 _maxBurnPerSecond)
external onlyOwner {
    maxBurnPerSecond = _maxBurnPerSecond;
}
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/Burnt.sol#L92,96,100,104,108 contracts/Carbon.sol#L108,112,116 contracts/Steak.sol#L62 contracts/Orchestrator.sol#L100
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.

```
function setAllowPublicBurn(bool _allowPublicBurn) public
onlyOwner {
    allowPublicBurn = _allowPublicBurn;
}

function setBurnInfluencingFactor(IBurnInfluencingFactor
_burnInfluencingFactor) external onlyOwner {
    burnInfluencingFactor = _burnInfluencingFactor;
}

function setburnThreshold(uint256 _burnThreshold) external
onlyOwner {
    burnThreshold = _burnThreshold;
}

function setBaseBurnPerSecond(uint256 _baseBurnPerSecond)
external onlyOwner {
    baseBurnPerSecond = _baseBurnPerSecond;
}

function setMaxBurnPerSecond(uint256 _maxBurnPerSecond)
external onlyOwner {
    maxBurnPerSecond = _maxBurnPerSecond;
}

...
```

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.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	contracts/Steak.sol#L47,51
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
stakingToken.transferFrom(msg.sender, address(this), amount)
stakingToken.transfer(msg.sender, amount)
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance
Before Transfer
```

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	contracts/Burnt.sol#L92,96,100,104,108 contracts/Carbon.sol#L108,112,116 contracts/Steak.sol#L62 contracts/Orchestrator.sol#L100
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
function setBurnInfluencingFactor(IBurnInfluencingFactor
_burnInfluencingFactor) external onlyOwner {
    burnInfluencingFactor = _burnInfluencingFactor;
}

function setburnThreshold(uint256 _burnThreshold) external
onlyOwner {
    burnThreshold = _burnThreshold;
}

function setBaseBurnPerSecond(uint256 _baseBurnPerSecond)
external onlyOwner {
    baseBurnPerSecond = _baseBurnPerSecond;
}

function setMaxBurnPerSecond(uint256 _maxBurnPerSecond)
external onlyOwner {
    maxBurnPerSecond = _maxBurnPerSecond;
}

function setAllowPublicBurn(bool _allowPublicBurn) public
onlyOwner {
    allowPublicBurn = _allowPublicBurn;
}

...
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	contracts/Carbon.sol#L27
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
uint256 public constant SECONDS_PER_YEAR = 31536000;
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/Deployer.sol#L18,19,20,21,22,23
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
address rosyToken = 0x21543397869098d5aF02E6AE611fE183dD3f2c6C
uint256 burnThreshold = 1_000_000_000 ether
uint256 baseBurnPerSecond = 32 ether
uint256 maxBurnPerSecond = 48 ether
uint256 scaleFactor = 5_700_000_000 wei
uint256 rewardPointRate = 1
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/Steak.sol#L62 contracts/Orchestrator.sol#L60,64,68,72,76,84,88,92,96,100 contracts/Carbon.sol#L108,112,116 contracts/Burnt.sol#L92,96,100,104,108
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
IStakeChangeListener _stakeChangeListener
IBurnInfluencingFactor _burnInfluencingFactor
uint256 _burnThreshold
uint256 _baseBurnPerSecond
uint256 _maxBurnPerSecond
uint256 _scaleFactor
IUserPointsFactor _userPointsFactor
IRewardEmitter _rewardEmitter
uint256 _annualRateBasisPoints
bool _allowPublicBurn
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	contracts/Carbon.sol#L117 contracts/Burnt.sol#L97,101,105,109
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
annualRateBasisPoints = _annualRateBasisPoints
burnThreshold = _burnThreshold
baseBurnPerSecond = _baseBurnPerSecond
maxBurnPerSecond = _maxBurnPerSecond
scaleFactor = _scaleFactor
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/Steak.sol#L2 contracts/Orchestrator.sol#L2 contracts/Deployer.sol#L2 contracts/Carbon.sol#L2 contracts/Burnt.sol#L2
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 ^0.8.20;
```

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**
IStakeChangeListener	Interface			
	onBeforeStakeChange	External	✓	-
Steak	Implementation	Ownable		
		Public	✓	Ownable
	stake	External	✓	-
	unstake	External	✓	-
	_onBeforeStakeChange	Internal	✓	
	setStakeChangeListener	External	✓	onlyOwner
Orchestrator	Implementation	Ownable2Step, IBurnInfluencingFactor, IUserPointsFactor, IStakeChangeListener		
		Public	✓	Ownable
	getBurnInfluencingFactor	External		-
	getUserPointsFactor	External		-
	onBeforeStakeChange	External	✓	onlySteak
	tryBurn	External	✓	publicBurnAllowed

	setBurnInfluencingFactor	External	✓	onlyOwner
	setburnThreshold	External	✓	onlyOwner
	setBaseBurnPerSecond	External	✓	onlyOwner
	setMaxBurnPerSecond	External	✓	onlyOwner
	setScaleFactor	External	✓	onlyOwner
	withdraw	External	✓	onlyOwner
	setStakeChangeListener	External	✓	onlyOwner
	setUserPointsFactor	External	✓	onlyOwner
	setRewardEmitter	External	✓	onlyOwner
	setAnnualRateBasisPoints	External	✓	onlyOwner
	setAllowPublicBurn	Public	✓	onlyOwner
	transferComponentOwnership	Public	✓	onlyOwner
	renounceComponentOwnership	Public	✓	onlyOwner
IBurnableERC20	Interface	IERC20		
	burn	External	✓	-
BurntSteakDeployer	Implementation			
		Public	✓	-
IUserPointsFactor	Interface			
	getUserPointsFactor	External		-

IRewardEmitter	Interface			
	onBeforeUpdatePoints	External	✓	-
	redeemPoints	External	✓	-
Carbon	Implementation	Ownable		
		Public	✓	Ownable
	_getAnnualRatePerSecond	Internal		
	_getUserPointsFactor	Internal		
	getEarnedPointsSinceLastUpdate	Public		-
	currentPoints	External		-
	updatePoints	External	✓	onlyOwner
	_updatePoints	Internal	✓	
	redeemPoints	External	✓	-
	setUserPointsFactor	External	✓	onlyOwner
	setRewardEmitter	External	✓	onlyOwner
	setAnnualRateBasisPoints	External	✓	onlyOwner
IBurnInfluencingFactor	Interface			
	getBurnInfluencingFactor	External		-
Burnt	Implementation	Ownable		
		Public	✓	Ownable
	_getBurnInfluencingFactor	Internal		

	burnRatePerSecond	Public		-
	tryBurn	External	✓	onlyOwner
	setBurnInfluencingFactor	External	✓	onlyOwner
	setburnThreshold	External	✓	onlyOwner
	setBaseBurnPerSecond	External	✓	onlyOwner
	setMaxBurnPerSecond	External	✓	onlyOwner
	setScaleFactor	External	✓	onlyOwner
	withdraw	External	✓	onlyOwner
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
	_contextSuffixLength	Internal		

Summary

Rosy token implements a staking, token burning and rewards mechanism. This audit investigates security issues, business logic concerns and potential improvements.

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