hashlock.

Security Audit

Carbonmark (Token)

November 2024

info@hashlock.com.au

Hashlock Pty Ltd

Table of Contents

Executive Summary 4
Project Context 4
Audit scope 7
Security Rating 8
Intended Smart Contract Functions 9
Code Quality 11
Audit Resources 11
Dependencies 11
Severity Definitions 12
Audit Findings 13
Centralisation 23
Conclusion 24
Our Methodology 25
Disclaimers 27
About Hashlock 28

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

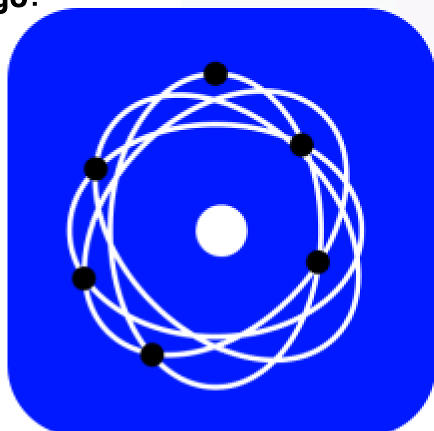
Executive Summary

The Carbonmark team partnered with Hashlock to conduct a security audit of their CreditToken.sol, CreditTokenFactory.sol Deploy.sol, Errors.sol and Validate.sol smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

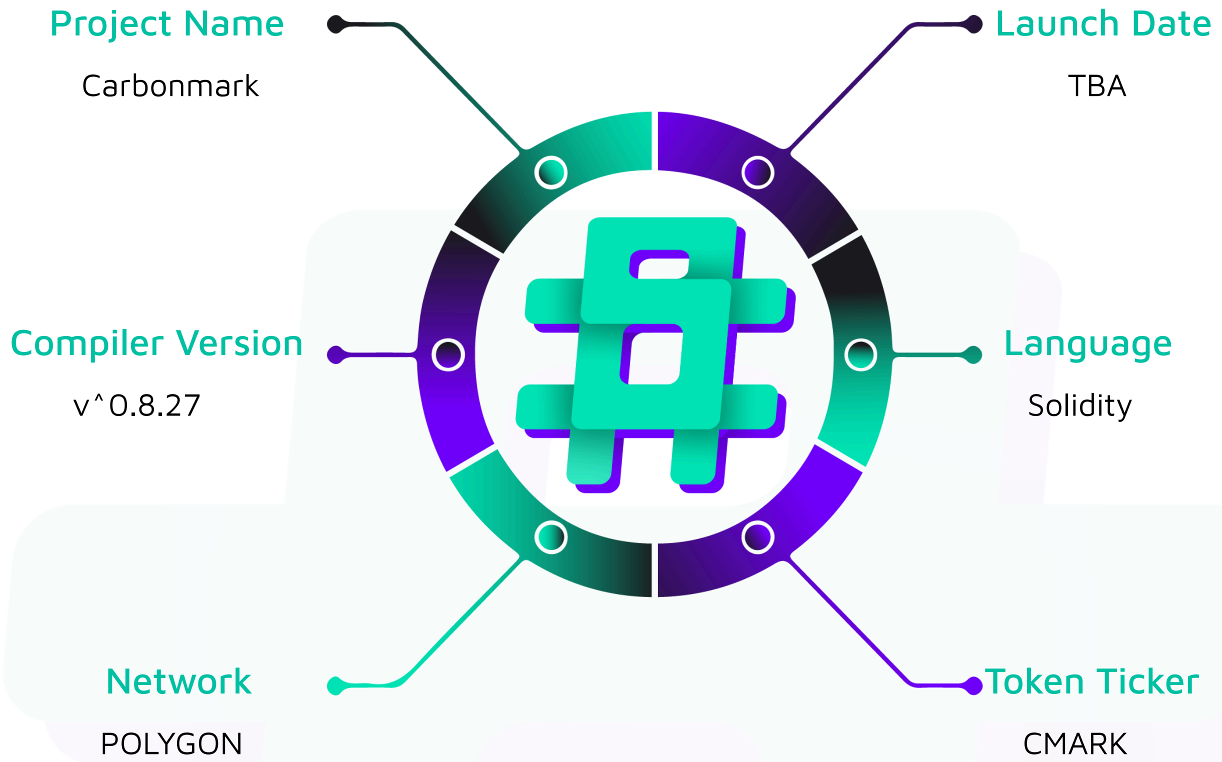
Project Context

Carbonmark is leveraging Web3 technology to revolutionize carbon offsetting by creating a decentralized, transparent, and verifiable platform for trading carbon credits. Through blockchain, Carbonmark ensures that every transaction is securely recorded, providing users with full traceability of their carbon offset contributions. This innovation not only empowers individuals and businesses to make verifiable, impact-driven choices but also fosters a new economy where sustainability and environmental responsibility are seamlessly integrated into digital assets and tokens. With Web3, Carbonmark is positioning itself at the intersection of climate action and cutting-edge blockchain solutions, offering a transformative way to support the fight against climate change while participating in the emerging digital economy.

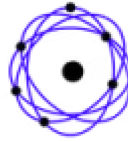
Project Name: Carbonmark
Compiler Version: ^0.8.27
Website: www.carbonmark.com
Logo:



Visualised Context:



Project Visuals:



CARBONMARK

Connecting nature to our global economy

Carbonmark's platform is the access point to environmental markets, making asset procurement simple, accessible and secure.

Audit scope

We at Hashlock audited the solidity code within the Carbonmark project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Carbonmark Protocol Smart Contracts**
Platform	**Polygon / Solidity**
Audit Date	**November, 2024**
Contract 1	CreditToken.sol
Contract 1 MD5 Hash	f1a5a8994ef8f494ac771507d6b0555e
Contract 2	CreditTokenFactory.sol
Contract 2 MD5 Hash	43d827d32273ad94b799a560cd09f362
Contract 3	Deploy.sol
Contract 3 MD5 Hash	593dafeaafb92684c25838569e51f601
Contract 4	Errors.sol
Contract 4 MD5 Hash	c60cba698b04327252e80ef375573f22
Contract 5	Validate.sol
Contract 5 MD5 Hash	d24d00e4991c3a352005a39bad28facc
GitHub Commit Hash	dc80e31c38a83ea0fbf45eefe3e03971ca1a3f42

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts. We initially identified some significant vulnerabilities that have since been addressed.

Not Secure	Vulnerable	Secure	Hashlocked

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The general security overview is presented in the Standardised Checks section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

1 Medium severity vulnerabilities

5 Low severity vulnerabilities

4 Gas Optimisations

2 QAs

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour

CreditToken.sol
- Allows users to:
    - Transfer tokens
    - Retire tokens from their balance
    - Retire tokens from another account (requires allowance)
- Allows the parent factory to:
    - Mint tokens to an address
    - Cancel tokens from circulation
    - Pause and unpause all token operations
	Contract achieves this functionality.

CreditTokenFactory.sol
- Allows owner to:
    - Issue new credits to a new or existing token contract
    - Cancels a specified amount of credits for a given credit ID and owner
    - Pause or unpause specific CreditToken contracts
    - Pause or unpause the entire factory
    - Upgrade the CreditToken implementation for all tokens via the beacon
    - Deploy a new beacon for backward-incompatible CreditToken upgrades
	Contract achieves this functionality.

Deploy.sol - Library used to: - Deploy and initialize CreditToken contracts	**Contract achieves this functionality.**

- Deploy and initialize CreditTokenFactory contracts.	
Errors.sol - Library used to declare errors.	**Contract achieves this functionality.**

Validate.sol
- Library used to:
  - Implement zero address check
  - Implement zero amount check
  - Implement max length check
	Contract achieves this functionality.

Code Quality

This audit scope involves the smart contracts of the Carbonmark project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was required.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Carbonmark project smart contract code in the form of Github access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies
QA	Quality Assurance (QA) findings are purely informational and don't impact functionality. These notes help clients improve the clarity, maintainability, or overall structure of the code, ensuring a cleaner and more efficient project. They should be addressed for optimization but are not critical to the system’s performance or security.

Audit Findings

Medium

[M-01] CreditTokenFactory#cancelCredits - Centralization risk of privileged function can cause unfair control over user tokens

Description

The `cancelCredit` allows for the privileged owner role to burn user tokens. This function forces the owner to enter a cancellation reason, however, this does not introduce sufficient decentralization of the function on its own. The owner account must be controlled by a multi-sig decentralized entity.

Vulnerability Details

The `cancelCredit` function allows the privileged owner role to burn any user's tokens.

```
    function cancelCredits(string calldata creditId, uint256 amount, string calldata reason, address owner)
        external
        onlyOwner
        whenNotPaused
    {
        Validate.notZeroAmount(amount);
        Validate.maxLength(reason, CANCEL_REASON_MAX);
        if (bytes(reason).length == 0) revert Errors.EmptyReason();
        CreditToken(_getCreditAddress(creditId)).cancel(amount, owner);
        emit Canceled(amount, reason, creditId, owner);
    }
```

As observed in this function, it forces the owner to enter a cancel reason when calling the function. However, this does not eliminate the centralization issue in the function. The owner can still use the function to burn user tokens maliciously or accidentally.

Impact

User tokens can be burnt maliciously or accidentally.

Recommendation

To mitigate centralization risks associated with the privileged function, consider implementing a multi-signature or decentralized governance mechanism in control of the privileged role.

Note

The Carbonmark team informed Hashlock that the ownership will be transferred to a multisig (2/3 initially, 3/5 later via safe.global).

Status

Resolved

Low

[L-01] CreditTokenFactory#transferOwnership - Function lacks an address(0) check

Description

It is good practice to check for `address(0)` when updating the state variable. This check reverts when there are accidental empty inputs to the function.

Recommendation

Implement an `address(0)` check to the function. An example is shown below.

```
function transferOwnership(address newOwner) public override onlyOwner {
    if (newOwner == address(0)) revert;
    pendingOwner = newOwner;
}
```

Status

Resolved

[L-02] CreditToken, CreditTokenFactory - Upgradeable contracts are missing `__gap[..]` storage variable

Description

Storage gaps are needed to not break storage layouts when adding new variables to base contracts. It is good practice to add it now rather than forgetting it later.

Recommendation

When designing upgradeable contracts, it's important to include `__gap[..]` storage variables as storage gaps to prevent issues with storage layout changes when adding new variables to base contracts in the future.

Status

Resolved

[L-03] CreditToken#transfer, transferFrom - Safe version of transfer and transferFrom should be used

Description

As it is implemented, `transfer` and `transferFrom` functions use `transfer` and `transferFrom` from the `ERC20Upgradeable.sol` of OpenZeppelin. However, these functions should use the safe version of these functions.

```
    function transfer(address to, uint256 amount) public virtual override whenNotPaused
returns (bool) {
        if (amount == 0) revert;
        return super.transfer(to, amount);
    }

    function transferFrom(address from, address to, uint256 amount)
        public
```

```
        virtual
        override
        whenNotPaused
        returns (bool)
    {
        return super.transferFrom(from, to, amount);
    }
```

Recommendation

Change the functions to use `safeTransfer` and `safeTransferFrom` from `openzeppelin-contracts-upgradeable/SafeERC20.sol`.

Note

The Carbonmark team informed Hashlock that the contracts do not interact with external tokens or invoke `transfer/transferFrom`. Refer to the OpenZeppelin discussion for details.

Status

Acknowledged

[L-04] Contracts - Avoid using floating pragma when not necessary

Description

A "floating pragma" in Solidity refers to the practice of using a pragma statement that does not specify a fixed compiler version but instead allows the contract to be compiled with any compatible compiler version. This issue arises when pragma statements like `pragma solidity ^0.8.27;` are used without a specific version number, allowing the contract to be compiled with the latest available compiler version. This can lead to various compatibility and stability issues.

Recommendation

Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs for the compiler version that is chosen.

Status

Resolved

[L-05] CreditTokenFactory - Centralization risk of privileged functions can have unintended results

Description

Functions with privileges need the owner to be trusted not to perform malicious updates. This also introduces a single point of failure. Functions with possible serious impacts are listed below.

```
function issueCredits()
function cancelCredits()
function pauseCredit()
function unpauseCredit()
function pause()
function unpause()
function upgradeBeacon()
function deployNewBeacon()
```

Recommendation

To mitigate centralization risks associated with the privileged function, consider implementing a multi-signature or decentralized governance mechanism.

Note

The Carbonmark team has claimed the same situation where the ownership will be transferred to a multisig (2/3 initially, 3/5 later via safe.global).

Status

Resolved

Gas

[G-01] CreditTokenFactory#transferOwnership - Prevent setting a state variable with the same value

Description

Not only is it wasteful in terms of gas, but this is especially problematic when an event is emitted and the old and new values set are the same, as listeners might not expect this kind of scenario.

Recommendation

Implement a new check in the function to prevent this scenario. An example is shown below.

```
function transferOwnership(address newOwner) public override onlyOwner {
     if (owner() == newOwner) revert;
     pendingOwner = newOwner;
 }
```

Status

Resolved

[G-02] CreditTokenFactory#acceptOwnership - State variables that are used multiple times in a function should be cached

Description

When performing multiple operations on a state variable in a function, it is recommended to cache it first. Either multiple reads or multiple writes to a state variable can save gas by caching it on the stack.

Recommendation

Change the function to cache the state variable that is used multiple times. An example is shown below.

```
function acceptOwnership() external {
        address pendingOwnerCached = pendingOwner;
        if (msg.sender != pendingOwnerCached) revert NotPendingOwner();
        _transferOwnership(pendingOwnerCached);
        pendingOwner = address(0);
    }
```

Note

The Carbonmark team emphasized to Hashlock that readability should take precedence over caching since gas efficiency is not a primary concern for this method.

Status

Acknowledged

[G-03] CreditTokenFactory#getAllCreditAddresses - Redundant getter function

Description

The getter function is used to return a state variable that is already set to public, meaning that this state variable is already accessible without the need of the getter function. This makes the function redundant and increases deployment gas costs.

Recommendation

Remove the function.

Note

The Carbonmark team explained that this was added for convenience, as default getters for public arrays are limited to single-element access.

Status

Acknowledged

[G-04] CreditToken#transfer, transferFrom - Avoid zero transfer calls

Description

In Solidity, unnecessary operations can waste gas. For example, a transfer function without a zero amount check uses gas even if called with a zero amount, since the contract state remains unchanged. Implementing a zero amount check avoids these unnecessary function calls, saving gas and improving efficiency.

Recommendation

Implement zero-amount checks in the functions. An example is shown below.

```
    function transfer(address to, uint256 amount) public virtual override whenNotPaused
returns (bool) {

        if (amount == 0) revert;

        return super.transfer(to, amount);

    }
```

Note

The Carbonmark team highlighted that the implementation remains aligned with the standard ERC20Upgradeable, with the addition of a pause modifier for enhanced functionality.

Status

Acknowledged

QA

[Q-01] CreditTokenFactory#acceptOwnership - Consider moving `msg.sender` checks to modifiers

Description

Consider using a modifier instead of checking with a required statement to improve code readability.

```
function acceptOwnership() external {
    if (msg.sender != pendingOwner) revert Errors.NotPendingOwner();
    _transferOwnership(pendingOwner);
    pendingOwner = address(0);
}
```

Recommendation

Implement a new `pendingOwnerOnly` modifier to this function. It will improve code readability at the cost of some extra gas.

Note

The Carbonmark team informed Hashlock that the `msg.sender` check is utilized only once, prioritizing readability and avoiding redundant abstractions.

Status

Acknowledged

[Q-02] CreditToken#transfer, transferFrom, CreditTokenFactory#getAllCreditAddresses - Consider using named returns

Description

Using named returns makes the code more self-documenting, makes it easier to fill out NatSpec, and in some cases can save gas.

Recommendation

Adopt named returns in your Solidity functions, especially in cases where functions contain a single return statement. This approach enhances code readability and documentation by making the return values clear and explicit. When defining your function, specify the return types with names, and manipulate these named variables directly within your function logic.

Status

Resolved

Centralisation

The Carbonmark project values security and utility over decentralisation.

The Carbonmark project values security and efficiency over decentralisation. Carbonmark emphasizes a security-focused approach while ensuring operational functionality. By centralizing critical functions, the project enhances reliability and rapid decision-making while maintaining accountability, ensuring the system remains robust and efficient.

Conclusion

After Hashlock's analysis, the Carbonmark project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:
In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:
Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au

hashlock.