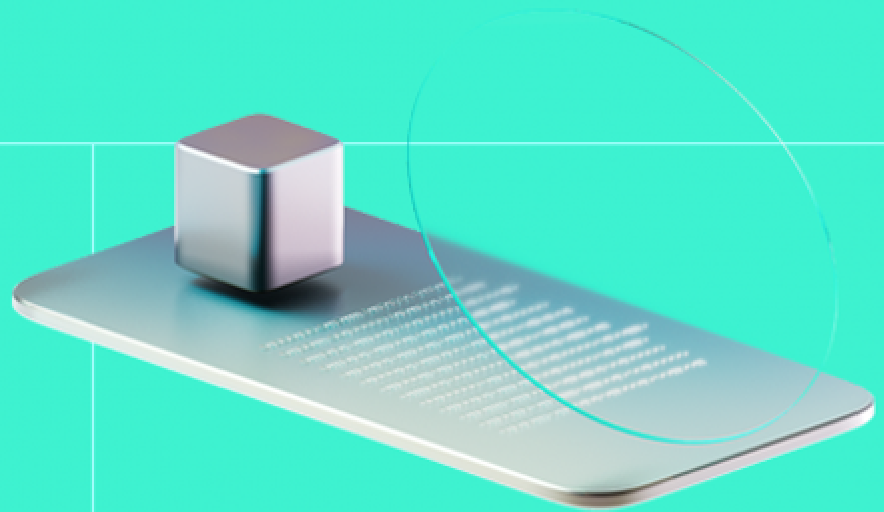
Smart Contract Code Review And Security Analysis Report



Customer: Covalent

Date: 04/07/2024

We express our gratitude to the Covalent team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Document

Name	Smart Contract Code Review and Security Analysis Report for Covalent
Audited By	Carlo Parisi, Viktor Raboshchuk
Approved By	Przemyslaw Swiatowiec
Website	https://www.covalenthq.com/
Changelog	27/06/2024 - Preliminary Report, 04/07/2024 - Final Report
Platform	EVM
Language	Solidity
Tags	ERC20
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/covalenthq/covalent-x-token
Commit	3c3db2841fcd77415c42413feabe483f9b8a8881

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	0
Low	0

Documentation quality

- Functional requirements are mostly missed.
- The technical description is not provided.
- NatSpec is sufficient.

Code quality

- No code quality issues were found.

Test coverage

Code coverage of the project is **93.10%** (branch coverage).

- Deployment and basic user interactions are covered with tests.

Table of Contents

System Overview **5**
Privileged Roles 5
Risks **6**
Findings **7**
Vulnerability Details 7
Observation Details 7
Disclaimers 12
Appendix 1. Severity Definitions **13**
Appendix 2. Scope **14**

System Overview

CovalentMigration - is a smart contract that is designed for the initial migration of Covalent Network Tokens. It has a setToken function to set a new token, and batchDistribute function to distribute tokens to multiple recipients in one transaction.

CovalentXToken - is a ERC20 token with additional functionalities. It inherits from ERC20Permit and AccessControlEnumerable, and implements ICovalentXToken. The mint function allows token minting within a capped rate, updateMintCap changes the minting cap, updatePermit2Allowance manages permit2's allowance. The token has the following attributes:

- Name: Covalent X Token
- Symbol: CXT
- Decimals: 18

DefaultEmissionManager - is a contract designed for managing the emissions of the Covalent ERC20 token on Ethereum L1. It uses Ownable2StepUpgradeable for ownership management and SafeERC20 for safe token operations. The mint function calculates the new supply based on the elapsed time and mints the required amount to maintain the target inflation rate, transferring minted tokens to the treasury. The inflatedSupplyAfter function calculates the expected supply after a given time period using the PowUtil library.

Privileged roles

- The owner of the CovalentMigration contract can set new tokens, and distribute the tokens to the users.
- The DEFAULT_ADMIN_ROLE of the CovalentXToken contract can grant and revoke any roles.
- The EMISSION_ROLE of the CovalentXToken contract can mint new tokens using the `mint` function, within the constraints of the minting cap.
- The CAP_MANAGER_ROLE of the CovalentXToken contract can update the minting cap using the `updateMintCap` function.
- The PERMIT2_REVOKER_ROLE of the CovalentXToken contract can update the allowance for the `PERMIT2` address using the `updatePermit2Allowance` function.
- The deployer (set during the contract deployment) of the DefaultEmissionManager.sol contract is the only entity allowed to initialize the contract

Risks

- The project utilizes Solidity version 0.8.20 or higher, which includes the introduction of the PUSH0 (0x5f) opcode. This opcode is currently supported on the Ethereum mainnet but may not be universally supported across other blockchain networks. Consequently, deploying the contract on chains other than the Ethereum mainnet, such as certain Layer 2 (L2) chains or alternative networks, might lead to compatibility issues or execution errors due to the lack of support for the PUSH0 opcode. In scenarios where deployment on various chains is anticipated, selecting an appropriate Ethereum Virtual Machine (EVM) version that is widely supported across these networks is crucial to avoid potential operational disruptions or deployment failures.
- The project has minimal documentation. This lack of comprehensive comments and explanations can lead to misunderstandings about the contract's functionality and intended behavior. It also makes the contract harder to maintain and audit, as future developers or auditors may not fully understand the contract's logic or the implications of its functions. This could potentially lead to overlooked bugs or security vulnerabilities. Furthermore, it may not meet the functional requirements if they are not clearly documented and understood.
- In the `DefaultEmissionManager.sol` contract, the exp2 function from the `PowUtil` library is called with a potentially large argument. The exp2 function expects an unsigned 192.64-bit fixed-point number, and returns a 60.18 unsigned fixed-point number. If the argument passed to exp2 exceeds the precision of a 192.64-bit fixed-point number, it could result in precision loss and inaccurate calculations.
- The `DefaultEmissionManager.sol` contract uses the `safeApprove` function from the OpenZeppelin library, which has been deprecated due to potential security risks. The `safeApprove` function can potentially lead to a race condition where someone may use both the old and the new allowance due to unfortunate transaction ordering. This could result in unauthorized or unexpected token transfers. OpenZeppelin recommends first reducing the spender's allowance to 0 and then setting the desired value to mitigate this risk.
- The `CovalentXToken.sol` contract forces users to grant infinite approval to the PERMIT2 address from Uniswap when the `permit2Enabled` variable is set to true. This could potentially expose users to unnecessary risks, as it allows the PERMIT2 address to spend an unlimited amount of tokens from the user's account at any time. While this might be intended to facilitate transactions with PERMIT2, it could be seen as an overreach, as users could simply approve to PERMIT2 when they want to use it, rather than being forced to grant infinite approval. This practice could potentially lead to misuse or unintended token transfers if the PERMIT2 address is compromised.
- The `DefaultEmissionManager.sol` contract's constructor is initializing immutable variables. This practice is generally discouraged because immutable variables are set at contract creation and cannot be changed afterwards. In the context of upgradeable contracts, this means that all proxies that share the same implementation contract will also share the same values for these immutable variables. If this is not the intended behavior, it could lead to unexpected results and potential security risks.

Findings

Vulnerability Details

Observation Details

F-2024-4072 - Missing Zero Address Check in setToken Function - Info

Description: In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the "zero address". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address. The "
Missing zero address control" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.
For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

The following methods should introduce zero address checks:

- CovalentMigration: setToken()

Assets:

- CovalentMigration.sol [https://github.com/covalenthq/covalent-x-token]

Status: Fixed

Recommendations

Remediation: It is strongly recommended to implement checks to prevent the zero address from being set during the initialization of contracts. This can be achieved by adding require statements that ensure address parameters are not the zero address.

Resolution: Fixed in the commit **89025dca19300daadcb50dc279685deece3d5d5e**: The check against zero address was introduced in the `setToken` function.

F-2024-4073 - Floating Pragma Statement in CovalentXToken.sol - Info

Description: The `CovalentXToken.sol` file uses a floating pragma statement `pragma solidity ^0.8.21;`. This means that the contract can be compiled with any compiler version from 0.8.21 and newer. This could potentially introduce unexpected behavior if the contract is compiled with a newer, backwards-incompatible compiler version.

Assets:

- CovalentXToken.sol [https://github.com/covalenthq/covalent-x-token]

Status: Fixed

Recommendations

Remediation: It is recommended to lock the pragma to a specific compiler version to ensure that the contract behaves as expected. This can be done by removing the caret (^) from the pragma statement.

Resolution: Fixed in the commit **5865ae70525e1c5eea66889c40961ea2d63ef662**: The exact compiler version (Pragma statement) was defined.

F-2024-4075 - Incomplete Initialization of DefaultEmissionManager.sol Contract - Info

Description:

The `DefaultEmissionManager.sol` contract inherits from the `Ownable2Step` contract but does not call the `Ownable2Step` initializer in its `initialize` function. This is a deviation from best practices, which recommend initializing all inherited contracts to ensure proper contract behavior and avoid potential security risks.

The `initialize` function sets the `token` and `startTimestamp` variables, approves the `migration` contract to spend the maximum possible amount of tokens, and transfers ownership to the `owner_` address. However, it bypasses the two-step ownership transfer process defined in the `Ownable2Step` contract by directly calling the `_transferOwnership` function.

```
function initialize(address token_, address owner_) external initializer {
    // prevent front-running since we can't initialize on proxy deployment
    if (DEPLOYER != msg.sender) revert();
    if (token_ == address(0) || owner_ == address(0)) revert InvalidAddress();
    token = ICovalentXToken(token_);
    startTimestamp = block.timestamp;
    assert(START_SUPPLY == token.totalSupply());
    token.safeApprove(address(migration), type(uint256).max); //@todo: uncomment
    // initial ownership setup bypassing 2 step ownership transfer process
    _transferOwnership(owner_);
}
```

Assets:

- DefaultEmissionManager.sol [https://github.com/covalenthq/covalent-x-token]

Status: Fixed

Recommendations

Remediation: Add a call to the `Ownable2Step` initializer in the `initialize` function of `DefaultEmissionManager.sol`. This will ensure that the `Ownable2Step` contract is properly initialized and the two-step ownership transfer process is respected.

Resolution: Fixed in the commit **7fc7cc3b628bc58fa35d2595823571deee85db5c**: The `Ownable2Step` initializer was added to the `initialize` function of `DefaultEmissionManager.sol`.

F-2024-4076 - Inefficient Token Minting Process in DefaultEmissionManager.sol - Info

Description: In the `DefaultEmissionManager.sol` contract, tokens are first minted to the contract's address and then transferred to the `treasury` address. This two-step process is Gas-inefficient as it involves an unnecessary transfer operation after the minting operation. Optimizing this process can save gas, making the contract more efficient and cost-effective.

```
function mint() external {
...
ICovalentXToken _token = token;
_token.mint(address(this), amountToMint);
_token.safeTransfer(treasury, amountToMint);
}
```

Assets:

- DefaultEmissionManager.sol [https://github.com/covalenthq/covalent-x-token]

Status: Fixed

Recommendations

Remediation: Mint the tokens directly to the `treasury` address. This will eliminate the need for a transfer operation, saving gas.

Resolution: Fixed in the commit **760556f219260569c5cc1cd351853c6322d462a8**: tokens are minted directly to the `treasury` address.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in 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).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

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 recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

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 vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/covalenthq/covalent-x-token
Commit	3c3db2841fcd77415c42413feabe483f9b8a8881
Whitepaper	
Requirements	https://covalentnetwork.mintlify.app/introduction
Technical Requirements	https://covalentnetwork.mintlify.app/introduction

Contracts in Scope
CovalentMigration.sol
CovalentXToken.sol
DefaultEmissionManager.sol
lib/PowUtil.sol
interfaces/ICovalentMigration.sol
interfaces/ICovalentXToken.sol
interfaces/IDefaultEmissionManager.sol