IMMUNEBYTES

Security Assessment

Smart Contract Audit

ethernity

09th Jan 2025

Ethernity Chain
epictoken
v1.0

Introduction

Name	Ethernity
Website	https://www.ethernity.io/
Repository/Source	https://github.com/ethernitychain/epictoken/tree/governor
Platform	L1
Network	Ethereum and EVM compatible chains
Language	Solidity
Initial Commit	bc207b68178504e7a80822c9ac4e384722595761
Final Commit	-
Timeline	7th Jan 2025 - 9th Jan 2025

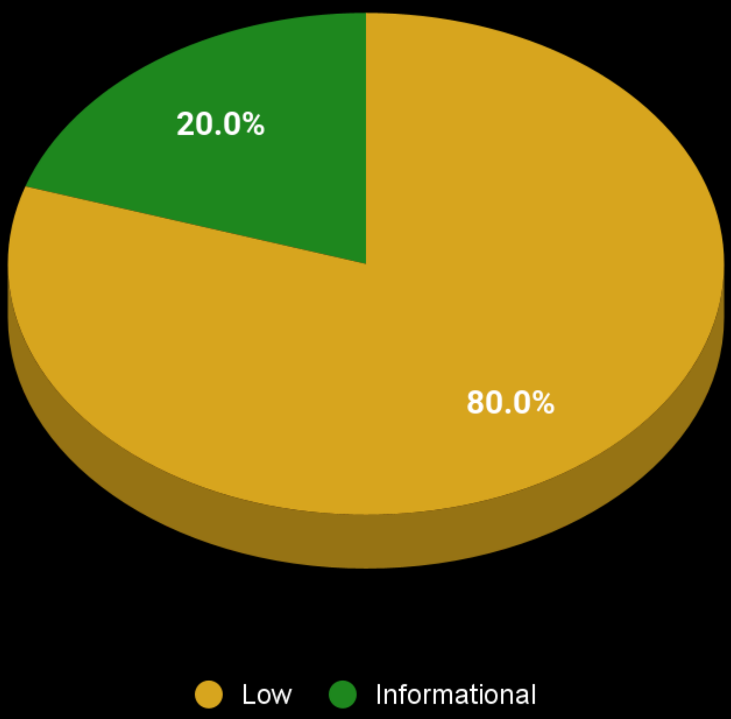
Table of Content

Executive Summary

This initial audit report provides a detailed overview of the findings from our completed security audit of the Ethernity project. The audit commenced on January 7, 2025, and concluded on January 9, 2025. This report captures the findings and observations from the final day of our assessment.

The Ethernity project is a sophisticated smart contract system that enables token minting, voting-based governance, and token swapping on the Ethereum blockchain. Our audit focused on ensuring the security, correctness, and robustness of the system's functionalities, including governance logic, minting constraints, and swap mechanisms.

Issues Overview



Project Overview

Key Components

1. **Initialization**
    - Secure initialization ensures that only authorized entities can set up critical contract parameters such as the governor, mintable contract, and token swap configurations.
    - Proper access control checks are implemented to prevent unauthorized initialization of sensitive contract states.

2. **Voting and Governance**
    - Ensure the voting mechanism is robust and accurately tracks the votes required for executing key actions like minting.
    - Prevent scenarios where vote manipulation could disrupt the governance logic.
    - Validate that vote resetting after successful operations functions as intended.

3. **Minting Functions**
    - Enforce the 12-month minting wait period and the 12% supply cap for new tokens.
    - Validate governor restrictions, ensuring only the designated governor contract can mint new tokens.
    - Verify that unauthorized entities cannot modify mintable contract parameters or exploit minting logic.

4. **Token Swap**
    - Ensure the swap process securely handles incoming and outgoing tokens and adheres to burn mechanics.
    - Validate the burn address and token swap configurations for correctness and security.
    - Prevent misuse of the token swap contract by ensuring all configuration changes are properly authorized.

5. **Admin Functions**
    - Verify that only authorized entities, such as the multisig owner, can perform sensitive admin actions like setting the governor, token swap addresses, or outgoing tokens.
    - Confirm the integrity of admin-configurable parameters to prevent unintended state changes or inconsistencies.
    - Validate the correctness of modifiers like onlyOwner and onlyGovernor.

6. **Reentrancy and Security**
    - Ensure all state-changing functions are protected against reentrancy attacks.

Security Checklist

During our comprehensive audit of the Ethernity project, we have assessed the following potential vulnerabilities inspired by common issues encountered in Ethereum-based systems:

Access Control Issue	✅
Initialization Vulnerabilities	✅
Reentrancy Attacks	✅
Logic and Calculation Errors	✅
Unintended State Manipulation	✅
ERC20 Compliance	✅
Token Swapping Risks	✅
Denial of Service (DoS)	✅
Ownership Validation	✅
Integer Overflow and Underflow	✅
Error Handling	✅
Frontrunning Risks	✅
Audit-Specific Issues	✅

Methodology

ImmuneBytes team has performed thorough testing of the project, starting with analyzing the code design patterns where the smart contracts and provided technical documentation were reviewed to ensure the architecture of the codebase aligns with the business specifications from a bird's eye view. During this phase, the invariants were identified along with execution of the pre-existing test suite to ensure smooth working of the implemented functionality and set up for the auditor's edge case testing in upcoming phases

Our team then performed a formal line-by-line inspection of the Smart Contract to find potential issues like Signature Replay Attacks, Unchecked External Calls, Merkle Tree Exploits, Reentrancy, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the next phase, the team ran curated test cases in order to verify the findings and issues identified during previous phases so as to provide a proof of concept. The team also runs automated tests over the provided code base using a combination of external and in-house-developed tools to identify vulnerabilities and security flaws of static nature.

The code was audited by a team of independent auditors, which included -

- Testing the functionality of the Smart Contracts to determine proper logic has been followed throughout.
- Thorough, line-by-line review of the code, done by a team of security researchers with experience across Blockchain security, cybersecurity, software engineering, game theory, economics and finance.
- Deploying the code on localnet using open source testing frameworks and clients to run live tests.
- Analyzing failing transactions to check whether the Smart Contracts implementation handles bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest secure version.
- Analyzing the security of the on-chain data and components.

Scope

In-Scope Files

The following files and their respective locations were included in the scope of the audit:

1. *epictoken/tree/governor/contracts/DAOGovSample.sol*
    - Governance contract implementing voting and minting logic based on community decisions. Key areas include vote counting, governor assignment, and mint authorization.
2. *epictoken/tree/governor/contracts/EpicToken.sol*
    - ERC20-compliant token contract with minting and burning capabilities, including constraints such as supply caps, time locks, and governor enforcement.
3. *epictoken/tree/governor/contracts/TokenSwap.sol*
    - Token swapping contract facilitating exchanges between tokens, with mechanisms for burning the incoming token and ensuring liquidity for outgoing tokens.

Critical Area of Focus

1. **Minting Constraints**
    - Validation of the minting logic to ensure adherence to the 12-month wait period and 12% cap on the total supply of 30 million.
    - Verification that only the designated governor contract can initiate mint operations.

2. **Governance and Voting Logic**
    - Ensuring accurate vote counting and enforcement of the required vote threshold before minting operations.
    - Validation of vote resetting and proper access control for governor changes.

3. **Token Swap Mechanism**
    - Examination of token burning, swap logic, and the proper configuration of incoming and outgoing token addresses.
    - Ensuring the integrity of burn address handling and prevention of double-spending scenarios.

4. **Access Control and Authorization**
    - Verification of onlyOwner and onlyGovernor modifiers to restrict access to sensitive functions.
    - Ensuring proper access control for admin actions such as setting the token swap address and outgoing tokens.

5. **Reentrancy and Security**
    - Auditing all state-changing functions, such as mint and swap, for potential reentrancy vulnerabilities.
    - Validation of proper error handling and prevention of unauthorized state manipulations.

6. **ERC20 Compliance**
    - Ensuring compatibility with ERC20 standards, including handling of tokens with non-standard implementations or edge cases.

Out of Scope

The following areas were excluded from the scope of this audit:

- **Frontend Integrations**: The review did not include the user interface or frontend applications that interact with the Ethernity contracts.
- **External Contracts**: Integration with external smart contracts or decentralized applications (dApps) beyond the provided codebase.
- **Deployment Process**: Gas optimization strategies and deployment processes were not examined.

Security Review Report

Severity	Open	Acknowledged	Partially Resolved	Resolved	**TOTAL**
Critical	-	-	-	-	-
High	-	-	-	-	-
Medium	-	-	-	-	1
Low	4	-	-	-	3
Informational	1	-	-	-	1
TOTAL	5	-	-	-	5

Severity Reference

1. **Critical**
   Issues causing protocol insolvency, unauthorized theft, or irreversible loss or alteration of assets, or governance outcomes.

2. **High**
   Issues involving theft or freezing of unclaimed yield, royalties, or temporary locking of assets.

3. **Medium**
   Operational failures, inefficiencies, or exploitations causing delays, excessive gas usage, or disruption without direct loss.

4. **Low**
   Minor deviations in promised functionality without impact on underlying value.

5. **Informational**
   Recommendations for code readability, maintainability, or best practices.

Status Reference

1. **Open**
   The issue has been identified and is pending resolution.

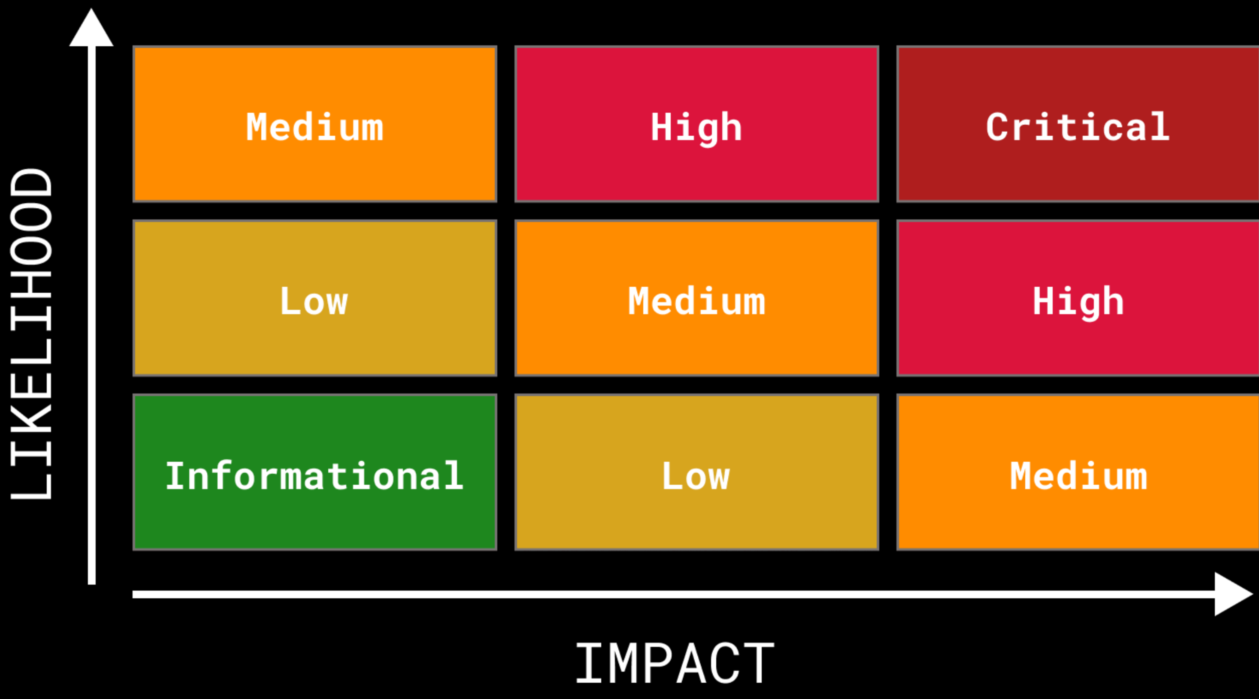
2. **Acknowledged**
   The issue has been reviewed and accepted by the team but remains unresolved.

3. **Partially Resolved**
   The issue has been mitigated to some extent, but residual risks or concerns persist.

4. **Resolved**
   The issue has been fully addressed, with no further action required.

5. **Closed**
   The issue is no longer relevant or actionable, regardless of resolution.

Risk Matrix


LIKELIHOOD
Medium
High
Critical
Low
Medium
High
Informational
Low
Medium
IMPACT

Summary of Findings

CODE	ISSUE	SEVERITY	STATUS
ECSA001	Improper Differentiation of Withdrawn Tokens	Low	Open
ECSA002	Missing Input Validation in EpicToken Constructor	Low	Open
ECSA003	Missing Input Validation in TokenSwap Constructor	Low	Open
ECSA004	Insufficient Supply Check During Token Swap	Low	Open
ECSA005	Lack of Check for Already Set Token Contracts	Info	Open

Findings Overview

Severity: Low	Impact: Low	Likelihood: Low
ECSA001	**Improper Differentiation of Withdrawn Tokens**	
File Name	TokenSwap.sol	
Status	Open	

Description

The adminTokenWithdraw function does not distinguish between tokens intended for swaps and those mistakenly sent to the contract. This oversight could lead to the accidental withdrawal of swap-related tokens by the admin, affecting users who are expecting a swap to succeed.

Code Affected

```
function adminTokenWithdraw(
    address _tokenAddress,
    uint256 _amount
) public onlyOwner nonReentrant {
    IERC20 _token = IERC20(_tokenAddress);
    _token.safeTransfer(owner(), _amount);
}
```

Proposed Solution

Introduce a mechanism to differentiate swapped tokens from mistaken/unclaimed tokens and restrict admin withdrawals accordingly.

Severity: Low	Impact: Low	Likelihood: Low
ECSA002	**Missing Input Validation in EpicToken Constructor**	
File Name	EpicToken.sol	
Status	Open	

Description

The EpicToken contract constructor does not validate the _swap and _waitPeriod parameters. An unvalidated _swap address can lead to critical misconfigurations if a zero address or an incorrect address is provided. Additionally, the absence of logical boundaries for _waitPeriod could result in an unrealistic or unsafe minting frequency, undermining the tokenomics.

Code Affected

```
constructor(
    string memory _name,
    string memory _symbol,
    address _swap,
    address _initialOwner,
    uint256 _waitPeriod
) ERC20(_name, _symbol) Ownable(_initialOwner) {
    require(_initialOwner.isContract(), "Initial owner can't be an EOA");
    waitPeriod = _waitPeriod;
    _mint(_swap, initialSupply * (10 ** 18));
    lastMint = block.timestamp;
}
```

Proposed Solution

Add require checks to ensure _swap is not a zero address and validate _waitPeriod within a reasonable range.

Severity: Low	Impact: Low	Likelihood: Low
ECSA003	**Missing Input Validation in TokenSwap Constructor**	
File Name	TokenSwap.sol	
Status	Open	

Description

The constructor of TokenSwap does not validate the _incomingToken and _burnAddress parameters. If either of these addresses is a zero address, the contract may fail to execute core functionalities like swapping or burning tokens, potentially locking user funds or breaking the swap mechanism.

Code Affected

```
constructor(
    address _incomingToken,
    address _burnAddress,
    address _initialOwner
) Ownable(_initialOwner) {
    require(_initialOwner.isContract(), "Initial owner can't be an EOA");
    incomingToken = IERC20(_incomingToken);
    burnAddress = _burnAddress;
}
```

Proposed Solution

Add require statements to ensure _incomingToken and _burnAddress are not zero addresses.

Severity: Low	Impact: Low	Likelihood: Low
ECSA004	**Insufficient Supply Check During Token Swap**	
File Name	TokenSwap.sol	
Status	Open	

Description

The swap function in TokenSwap transfers outgoingToken without checking the contract's balance. If a user attempts to swap more tokens than the contract currently holds, the transaction will revert, causing poor user experience and potential disputes.

Code Affected

```
function swap(uint256 _amount) public nonReentrant {
    incomingToken.safeTransferFrom(msg.sender, burnAddress, _amount);
    outgoingToken.safeTransfer(msg.sender, _amount);
}
```

Proposed Solution

Add a balance check to ensure the contract holds sufficient outgoingToken before transferring.

Severity: Info	
ECSA005	**Lack of Check for Already Set Token Contracts**
File Name	DAOGovSample.sol
Status	Open

Description

The setMintableContract function in DAOGovSample does not check if the mintableContract has already been initialized. This could lead to accidental overwrites, allowing the owner to change the mintable token contract after deployment, which could undermine trust and disrupt governance.

Code Affected

```
function setMintableContract(address _mintableContract) external onlyOwner {
    require(_mintableContract != address(0), "Invalid contract address");
    mintableContract = _mintableContract;
    epicToken = EpicToken(_mintableContract);
}
```

Proposed Solution

Add a require condition to ensure mintableContract is not already set before assignment.

Concluding Remarks

During the audit of Ethernity, our auditors identified issues categorized as Low severity within the system. It is strongly recommended that these issues be resolved as a priority by the development team before proceeding to production. Implementing the suggested recommendations will enhance the efficiency and security of the smart contract's operations.

Disclaimer

ImmuneBytes' security audit services are designed to help identify and mitigate potential security risks in smart contracts and related systems. However, it is essential to recognize that no security audit can guarantee absolute protection against all possible security threats. The audit findings and recommendations are based on the information provided during the assessment.

Furthermore, the security landscape is dynamic and ever-evolving, with new vulnerabilities and threats emerging regularly. As a result, it is strongly advised to conduct multiple audits and establish robust bug bounty programs as part of a comprehensive and continuous security strategy. A single audit alone cannot protect against all potential risks, making ongoing vigilance and proactive risk management essential.

To ensure maximum security, the project must implement the recommendations provided in the audit report and regularly monitor its smart contracts for vulnerabilities. It is imperative to adopt appropriate risk management strategies and remain proactive in addressing potential threats as they arise.

Please note that auditors cannot be held liable for any security breaches, losses, or damages that may occur after the audit has been completed despite using audit services. The responsibility for implementing the recommendations and maintaining the ongoing security of the systems rests solely with the project.