

Concrete Token

Blueprint Finance

HALBORN

Concrete Token - Blueprint Finance

Prepared by:  HALBORN

Last Updated 01/09/2026

Date of Engagement: December 15th, 2025 - December 15th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	0	0	1

TABLE OF CONTENTS

1. Introduction

2. Assessment summary

3. Test approach and methodology

4. Risk methodology

5. Scope

6. Assessment summary & findings overview

7. Findings & Tech Details

7.1 Inconsistent token name between erc20 and erc20permit implementations

8. Automated Testing

1. Introduction

Blueprint Finance engaged Halborn to conduct a security assessment on their **Concrete Token** program beginning and ending on December 15th, 2025. The security assessment was scoped to the smart contracts provided in the GitHub repository [Concrete-Token](#), commit hashes, and further details can be found in the Scope section of this report.

The **Blueprint Finance team** is releasing a new version of their **Concrete Token** Ethereum smart contracts. The **Concrete Token** ecosystem implements standard ERC20 functionality with ERC20Permit for gasless approvals, role-based access control for minting and upgrading, and LayerZero integration for cross-chain token transfers through both native OFT and adapter patterns.

2. Assessment Summary

Halborn was provided 1 business days for the engagement and assigned one full-time security engineer to review the security of the Ethereum smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the Ethereum smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified an improvement to reduce the likelihood and impact of risks, which was acknowledged by the **Blueprint Finance team**:

- Use consistent naming between ERC20 token name and ERC20Permit domain name to stick to OpenZeppelin's best practices

3. Test Approach And Methodology

Halborn performed a combination of manual review and security testing based on scripts to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Manual code review of the ConcreteToken smart contract implementation.
- Proof of Concept (POC) development to verify critical functionality including initialization, UUPS upgrade mechanism, ERC20Permit functionality, and implementation contract protection.
- Analysis of role-based access control implementation and upgrade authorization mechanisms

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets of Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker’s control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

Impact Metric (M_I)	Metric Value	Numerical Value
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical (A:C)	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium (Y:M)	0.5
	High (Y:H)	0.75
	Critical (Y:C)	1

Impact I is calculated using the following formula:

$$I = \max(m_I) + \frac{\sum m_I - \max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

Severity Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient *C* is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score *S* is obtained by:

$$S = \min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: Concrete-Token

(b) Assessed Commit ID: b8319fc

(c) Items in scope:

- src/ConcreteToken.sol
- src/ConcreteTokenOFT.sol
- src/ConcreteTokenOFTAdapter.sol

Out-of-Scope: Third party dependencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCONSISTENT TOKEN NAME BETWEEN ERC20 AND ERC20PERMIT IMPLEMENTATIONS	INFORMATIONAL	ACKNOWLEDGED - 12/22/2025

7. FINDINGS & TECH DETAILS

7.1 INCONSISTENT TOKEN NAME BETWEEN ERC20 AND ERC20PERMIT IMPLEMENTATIONS

// INFORMATIONAL

Description

Both `ConcreteToken` and `ConcreteToken0FT` contracts are upgradeable ERC20 tokens that implement ERC20Permit functionality. During the token initialization process, there is a naming inconsistency between the ERC20 token name and the ERC20Permit domain name implementation in both contracts. Both contracts allow separate parameters for the ERC20 token name and the ERC20Permit domain name during initialization. This design permits potential inconsistencies where the ERC20 token name (accessible via `token.name()`) differs from the EIP-712 domain separator used for permit signatures. According to OpenZeppelin's documentation, it is recommended to use the same name for both the ERC20 token and the ERC20Permit domain to maintain consistency and avoid potential confusion.

`src/ConcreteToken.sol`

```
91 | __ERC20_init(name, symbol);
92 | __AccessControlDefaultAdminRules_init(initialDelay, defaultAdmin);
93 | __ERC20Permit_init(eip712Name);
```

This design creates potential confusion for developers integrating with the tokens, as they need to understand which name to use when constructing permit signatures. While the permit functionality works correctly when the domain separator is used consistently, developers might incorrectly assume they should use the ERC20 token name when generating EIP-712 signatures, leading to invalid signatures and failed transactions.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to use the same name for both the ERC20 token initialization and the ERC20Permit initialization to maintain consistency and follow OpenZeppelin's best practices.

Remediation Comment

ACKNOWLEDGED: The `Blueprint Finance team` acknowledged this finding. The `Blueprint Finance team` is aware of the OpenZeppelin recommendation and does not plan to use different strings for the permit and ERC20 name. While maintaining the flexibility to do so, they intend to keep the permit string identical to the token name across all chain deployments, even if token names vary.

References

8. AUTOMATED TESTING

Description

Halborn used automated security scanners to assist with the detection of well-known security issues and vulnerabilities. Among the tools used was **Slither**, a static analysis framework for Solidity smart contracts that identifies common vulnerabilities and code quality issues. Slither performs comprehensive analysis including detection of reentrancy vulnerabilities, incorrect implementations of standards, and optimization opportunities.

Results

No security vulnerabilities were detected by automated scanning tools. **Slither** analysis revealed only informational findings related to OpenZeppelin library implementations and naming conventions, none of which affect the security of the ConcreteToken contract itself.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.