

Adding ERC20CappedUpgradeable to Concrete Token

Blueprint Finance

HALBORN

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ABOUT THIS REPORT

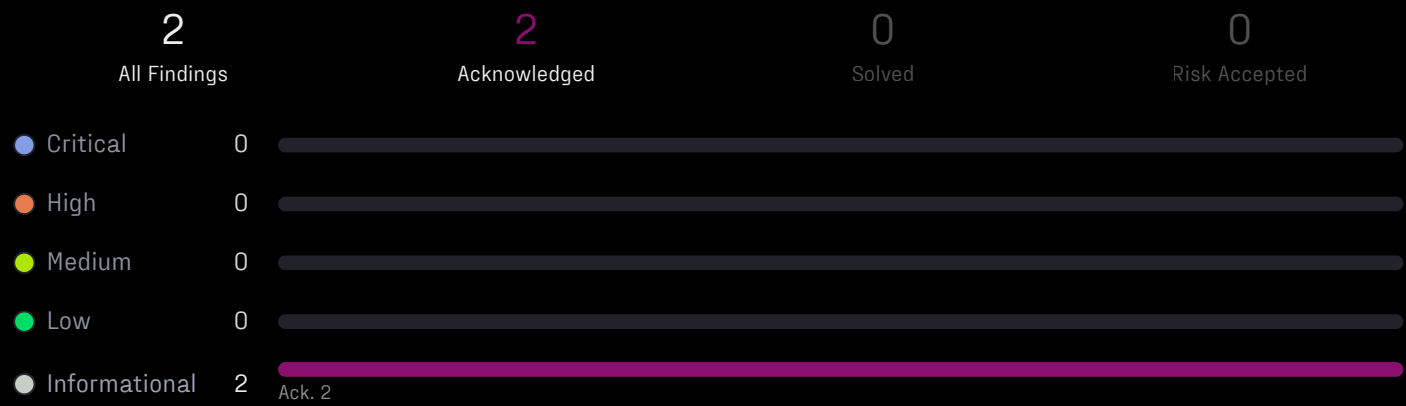
ASSESSMENT TYPE
Assurance Assessment

DATE OF ENGAGEMENT
Sep 11, 2026 - Sep 12, 2026

SOURCE CODE
Concrete-Token

ASSESSED COMMIT ID
774a120

SEVERITY BREAKDOWN



INTRODUCTION

Blueprint Finance engaged Halborn to perform a security assessment of their smart contracts on September 11th, 2026. The assessment scope was limited to the smart contracts provided to Halborn. Commit hashes and additional details are available in the Scope section of this report.

The Blueprint Finance codebase in scope consists of changes made to the upgradeable ConcreteToken ERC20, which now inherits ERC20CappedUpgradeable to enforce a maximum total supply, overrides the _update hook to apply the cap on every mint path, and replaces the original initializer with an initializeV2 reinitializer that sets the immutable cap.

ASSESSMENT SUMMARY

Halborn was allocated 1 day for this engagement and assigned 1 full-time security engineer to conduct a comprehensive review of the smart contracts within scope. The engineer is an expert in blockchain and smart contract security, with advanced skills in penetration testing and smart contract exploitation, as well as extensive knowledge of multiple blockchain protocols.

The objectives of this assessment are to:

- Identify potential security vulnerabilities within the smart contracts.
- Verify that the smart contract functionality operates as intended.

In summary, **Halborn** did not identify any significant security risks. However, some enhancements were recommended to improve code clarity, consistency, or maintainability. These were acknowledged by the **Blueprint Finance** team:

- Validate `cap_` in `ConcreteToken.initializeV2()` before calling `__ERC20Capped_init`, requiring `cap_ >= totalSupply()` with a documented upper bound, and emit a `CapInitialized` event so the chosen `cap` is observable from logs.
- Guarantee that the upgrade transaction installing the capped implementation carries the `initializeV2(CAP)` calldata in `ConcreteToken.upgradeToAndCall()`, enforced by the deployment tooling and multisig runbook, and add a post-upgrade check that `cap()` returns a non-zero value before minting resumes.

I TEST APPROACH AND METHODOLOGY

Halborn conducted a combination of manual code review and automated security testing to balance efficiency, timeliness, practicality, and accuracy within the scope of this assessment. While manual testing is crucial for identifying flaws in logic, processes, and implementation, automated testing enhances coverage of smart contracts and quickly detects deviations from established security best practices.

The following phases and associated tools were employed throughout the term of the assessment:

- Research into the platform's architecture, purpose and use.
- Manual code review and walkthrough of smart contracts to identify any logical issues.
- Comprehensive assessment of the safety and usage of critical Solidity variables and functions within scope that could lead to arithmetic-related vulnerabilities.
- Local testing using custom scripts (**Foundry**).
- Fork testing against main networks (**Foundry**).
- Static security analysis of scoped contracts, and imported functions (**Slither**).

I 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)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	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 (<i>m_I</i>)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N)	0
	Low (C:L)	0.25
	Medium (C:M)	0.5
	High (C:H)	0.75
	Critical (C:C)	1
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

IMPACT METRIC (m_I)	METRIC VALUE	NUMERICAL VALUE
Yield (Y)	Critical (D:C)	1
	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.

Critical	High	Medium	Low	Informational

9 - 10	7 - 8.9	4.5 - 6.9	2 - 4.4	0 - 1.9
--------	---------	-----------	---------	---------

Scope

REPOSITORY ^

(a) Repository:  Concrete-Token

(b) Assessed Commit ID:  774a120188107c981fe92637918322ce16eb1e56

(c) Items in scope:

src 1 file

<> ConcreteToken.sol Solidity

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

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

Assessment Summary & Findings Overview

#	TITLE	SEVERITY	SCORE	STATUS
HAL-001	Mint freeze due to missing cap validation in initializeV2	● Informational	0.5	ACKNOWLEDGED 09/11/2026
HAL-002	Temporary total mint freeze due to non-atomic upgrade leaving cap() at zero	● Informational	0.3	ACKNOWLEDGED 09/11/2026

Findings & Tech Details

HAL-001

ACKNOWLEDGED

Mint freeze due to missing cap validation in initializeV2

• Informational · 0.5 A0:S/AC:L/AX:L/R:F/S:U/C:N/A:C/I:N/D:N/Y:N

DESCRIPTION

The `ConcreteToken.initializeV2()` function is the one-shot setter that installs the token's immutable supply cap when the capped implementation goes live. It is gated by `onlyRole(DEFAULT_ADMIN_ROLE)` and marked `reinitializer(2)`, and it forwards its `cap_` argument directly to OpenZeppelin's `__ERC20Capped_init(cap_)`.

However, `__ERC20Capped_init` validates only that `cap_ != 0`. The `ConcreteToken.initializeV2()` function adds no further checks: it never verifies that `cap_` is at least the live `totalSupply()`, sets no upper bound, and emits no event recording the value chosen. Because the cap is written once (the `reinitializer(2)` version is consumed in the same call) and the contract exposes no cap setter and no burn function, whatever value is stored persists until the admin intervenes with a further full UUPS implementation upgrade.

A `DEFAULT_ADMIN_ROLE` holder that submits a `cap_` below the current `totalSupply()` freezes all token issuance: every subsequent mint reverts with `ERC20ExceededCap`. This is a self-inflicted condition recoverable by the same 2-of-3 admin Safe, which also controls `__authorizeUpgrade` and can deploy a follow-up implementation with a `reinitializer(3)` to reset or override the cap and restore minting. A `cap_` set far above the intended ceiling has the opposite effect, leaving the advertised supply cap cosmetic while still satisfying every on-chain check. The same absence of validation accepts a units mismatched value, such as a cap scaled to `1e9` instead of `1e18`, just as silently. The only floor check in the system, `require(supplyBefore <= CAP)`, lives in the off-chain deploy script and does not execute inside the transaction the admin submits, and since no event is emitted, a wrong cap is invisible to log based monitoring and observable only by reading the raw ERC-7201 storage slot.

RECOMMENDATION

In `ConcreteToken.initializeV2()`, validate the argument before calling `__ERC20Capped_init`: require `cap_ >= totalSupply()` and enforce a documented sane upper bound tied to the intended tokenomics ceiling. Additionally, emit a `CapInitialized(uint256 cap)` event so the chosen cap is observable from logs immediately rather than only through raw storage reads.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Blueprint Finance** team made a business decision to acknowledge this finding and not alter the contracts, stating:

The behaviour is intentional. `initializeV2()` is admin-only, executed atomically with the upgrade, and the cap value is set at deployment. A cap below `totalSupply()` halts issuance, that is the safe direction of failure and matches our intent: we do not want to mint beyond the cap under any circumstance. Existing balances and transfers are unaffected, and the condition is recoverable by the same admin Safe. We accept the risk.

Temporary total mint freeze due to non-atomic upgrade leaving cap() at zero

• Informational · 0.3 A0:S/AC:L/AX:L/R:F/S:U/C:N/A:M/I:N/D:N/Y:N

DESCRIPTION

The `ConcreteToken._authorizeUpgrade()` function is the UUPS authorization hook that guards implementation upgrades, gated by `onlyRole(DEFAULT_ADMIN_ROLE)`. It has an empty body and imposes no requirement on the `data` calldata that accompanies the upgrade. OpenZeppelin's `UUPSUpgradeable.upgradeToAndCall(newImplementation, data)` accepts `data.length == 0`, so the pointer swap and the cap setting call are not forced to happen together.

However, the capped implementation depends on `initializeV2` running to set the cap. If the `DEFAULT_ADMIN_ROLE` holder executes `upgradeToAndCall(newImpl, "")`, splitting the upgrade and the cap setting into two separate transactions instead of the atomic form carrying `abi.encodeCall(initializeV2, (CAP))`, the new implementation goes live with `cap()` reading zero.

While the cap is zero, `ERC20CappedUpgradeable._update` reverts on every mint of a non-zero amount with `ERC20ExceededCap`, because any positive supply already exceeds a zero cap. The result is a total minting freeze that lasts until a follow-up `initializeV2(CAP)` transaction succeeds. The window is reachable and closable only by the `DEFAULT_ADMIN_ROLE` holder, since `initializeV2` is admin gated, and the contract natspec already states that the upgrade must be executed atomically. The freeze is therefore temporary and recoverable by the same admin, but nothing on-chain enforces the atomicity the natspec assumes.

RECOMMENDATION

Guarantee that the single upgrade transaction installing the capped implementation carries the `initializeV2(CAP)` calldata in the `data` argument of `ConcreteToken.upgradeToAndCall()`, and enforce this through the deployment tooling and the multisig approval runbook so the implementation pointer swap and the cap initialization are committed together.

Retain the `ConcreteToken.initializeV2()` natspec note stating that `cap()` reads zero and every mint reverts until `initializeV2` runs. Add a post-upgrade verification step that confirms `cap()` returns the intended non-zero value before minting resumes.

REMEDIATION COMMENT

ACKNOWLEDGED: The **Blueprint Finance** team made a business decision to acknowledge this finding and not alter the contracts, stating:

`initializeV2` carries the `reinitializer(2)` modifier and is additionally restricted to `<ROLE>`, so no other account can call it. Per the NatSpec on OpenZeppelin's `Initializable`, a reinitializer must be called atomically with the upgrade, in our deployment it is executed via the data payload of `upgradeToAndCall`, so the implementation swap and the re-initialization happen in the same transaction. There is thus no block in which the new implementation is live but uninitialized, and the version guard prevents any repeat call.

Disclaimer

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.