Zellic

**Prepared for**
**Eric Nordelo**
OpenZeppelin

**Prepared by**
**Jinseo Kim**
**Jisub Kim**
Zellic

**June 3, 2025**

# OpenZeppelin Cairo Contracts
Cairo Application Security Assessment

## Contents

## About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

# 1. Overview

## 1.1. Executive Summary

Zellic conducted a security assessment for OpenZeppelin from May 19th to May 20th, 2025. During this engagement, Zellic reviewed OpenZeppelin Cairo contracts' code for security vulnerabilities, design issues, and general weaknesses in security posture.

## 1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does the library adhere to robust and good coding practices?
- Are the macros implemented appropriately?
- Are the test suites implemented appropriately?

## 1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Optimizing runtime performance
- Infrastructure relating to the project

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

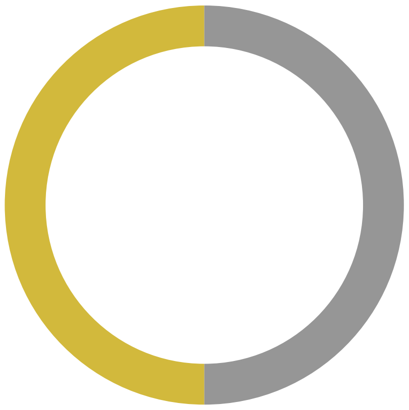
## 1.4. Results

During our assessment on the scoped OpenZeppelin Cairo contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of OpenZeppelin in the Discussion section (4. ↗).

## Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| 🟥 Critical | 0 |
| 🟧 High | 0 |
| 🟨 Medium | 1 |
| 🟩 Low | 0 |
| ⬜ Informational | 1 |



# 2. Introduction

## 2.1. About OpenZeppelin Cairo Contracts

OpenZeppelin contributed the following description of OpenZeppelin Cairo contracts:

> A library for secure smart contract development written in Cairo for Starknet, a decentralized ZK Rollup. The library contains a set of components, contract presets, and utilities intended as backbone for different Starknet protocols implemented in Cairo.

## 2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

**Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the targets.

**Architecture risks.** This encompasses potential hazards originating from the blueprint of a system, which involves its core validation mechanism and other architecturally significant constituents influencing the system's fundamental security attributes, presumptions, trust mode, and design.

**Arithmetic issues.** This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

**Implementation risks.** This encompasses risks linked to translating a system's specification into practical code. Constructing a custom system involves developing intricate on-chain and off-chain elements while accommodating the idiosyncrasies and challenges presented by distinct programming languages, frameworks, and execution environments.

**Availability.** Denial-of-service attacks are another leading issue in custom systems. Issues including but not limited to unhandled panics, unbounded computations, and incorrect error handling can potentially lead to consensus failures.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and

Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped targets itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

## 2.3. Scope

The engagement involved a review of the following targets:

### OpenZeppelin Cairo Contracts Targets

| | |
|---|---|
| **Type** | Cairo |
| **Platform** | Starknet |

| | |
|---|---|
| **Target** | Only changes to Cairo contracts between 680ff88e...1f792f52 |
| **Repository** | https://github.com/OpenZeppelin/cairo-contracts ↗ |
| **Version** | `1f792f52e7842eed345373ee8554af32d2aa9056` |
| **Programs** | `*.cairo` |

| | |
|---|---|
| **Target** | cairo-contracts |
| **Repository** | https://github.com/OpenZeppelin/cairo-contracts ↗ |
| **Version** | `1f792f52e7842eed345373ee8554af32d2aa9056` |
| **Programs** | `packages/macros/src/with_components.rs` |

## 2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of four person-days. The assessment was conducted by two consultants over the course of two calendar days.

### Contact Information

The following project managers were associated with the engagement:

**Jacob Goreski**
Engagement Manager
jacob@zellic.io ↗

**Chad McDonald**
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

**Jinseo Kim**
Engineer
jinseo@zellic.io ↗

**Jisub Kim**
Engineer
jisub@zellic.io ↗

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **May 19, 2025** | Start of primary review period |
| **May 20, 2025** | End of primary review period |

## 3. Detailed Findings

### 3.1. Withdraw/redeem limits in ERC-4626 may be bypassed if the before_withdraw hook reenters

| | | | |
|---|---|---|---|
| **Target** | ERC4626Component | | |
| **Category** | Business Logic | **Severity** | Medium |
| **Likelihood** | Medium | **Impact** | Medium |

#### Description

The ERC-4626 component allows a deployer to define custom limits on withdrawing (redeeming) or depositing (minting) tokens (shares) by overriding the functions `withdraw_limit`, `redeem_limit`, `deposit_limit`, and `mint_limit`. The user-facing entry points of this component (`withdraw`, `redeem`, `deposit`, and `mint`) enforce these limits if defined. Note that pairs of the entry points — `withdraw` and `redeem` as well as `deposit` and `mint` — are equivalent features with different interfaces.

The component also provides the way to hook into the withdrawal and deposit logic by overriding the `before_withdraw` function and the `after_deposit` function. These functions are invoked in the withdraw and deposit functions of the contract.

The following code shows how these features jointly work in the `withdraw` function:

```
fn max_withdraw(self: @ComponentState<TContractState>, owner: ContractAddress)
    -> u256 {
    // (...)
    match Limit::withdraw_limit(self, owner) {
        // (returns limit if defined)
    }
}

fn withdraw(
    ref self: ComponentState<TContractState>,
    assets: u256,
    receiver: ContractAddress,
    owner: ContractAddress,
) -> u256 {
    let max_assets = self.max_withdraw(owner);
    assert(assets <= max_assets, Errors::EXCEEDED_MAX_WITHDRAW);
    // (...)
    self._withdraw(caller, receiver, owner, assets, shares);
    // (...)
}
```

```
fn _withdraw(
    ref self: ComponentState<TContractState>,
    caller: ContractAddress,
    receiver: ContractAddress,
    owner: ContractAddress,
    assets: u256,
    shares: u256,
) {
    // Before withdraw hook
    Hooks::before_withdraw(ref self, assets, shares);
    // (burns shares and transfers tokens)
}
```

Note that the `withdraw_limit` is checked before the `before_withdraw` hook is invoked. If this hook reenters to invoke the `withdraw` function again, the subsequent check on `withdraw_limit` will be performed before the first withdrawal logic (burning shares and transferring tokens) is executed.

Suppose the vesting logic is implemented by overriding the `withdraw_limit`. One may implement this function to return `current shares - committed shares * remaining time / total vesting period`. Let us assume `current shares` = 100, `committed shares` = 100, `remaining time` = one year, and `total vesting period` = two years. The owner is allowed to redeem their shares (withdraw their tokens) up to 50 as of now.

However, if the `before_withdraw` hook reenters the `withdraw` function again (maybe through an untrusted contract), the withdrawal limit could be bypassed in the following way:

1. A user invokes the `withdraw` function to burn 50 shares, and the following occurs.
    - The first check on `withdraw_limit` will return 50.
    - The `before_withdraw` hook reenters to the `withdraw` function to burn 50 shares.
    - The second check on `withdraw_limit` will return 50.
    - Fifty shares of the user are burned, and the user receives the corresponding tokens.
2. Fifty shares of the user are burned, and the user receives the corresponding tokens — again.
3. As a result, the user successfully burns all their shares, bypassing the defined limit.

## Impact

The `withdraw_limit`, which depends on the current shares of a user, could be bypassed if the `before_withdraw` hook reenters.

### Recommendations

Consider performing the reentrancy check at the `withdraw` and `redeem` functions or documenting the security implications of calling an untrusted contract in the hook.

### Remediation

This issue has been acknowledged by OpenZeppelin, and a fix was implemented in commit `800ea465` ↗. OpenZeppelin has updated their documentation to reflect our recommendation in the associated fix.

## 3.2. Weak `initializer_missing` check

| | | | |
|---|---|---|---|
| **Target** | Macros | | |
| **Category** | Code Maturity | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

The current implementation only looks for the literal substring `self.<component>.initializer(` inside the constructor code. As a result, commenting out the line or embedding the same phrase inside a string literal makes the warning disappear even though the initializer call is actually missing.

### Impact

Since a comment or string literal can satisfy the check, the actual initializer may never run, leaving the component in an un-initialized state and causing unintended behavior at runtime.

### Recommendations

Consider adding an AST-based check that ignores comments and string literals and verifies the initializer is truly invoked.

### Remediation

This issue has been acknowledged by OpenZeppelin. OpenZeppelin stated that they plan to improve the macro in the future including this suggestion for warnings.

# 4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

## 4.1. Notable changes

Here are notable changes between

- Git commit 680ff88e0fb5c22bba4cf130cdbca2e4f2cf85fb corresponding to tag `release-v1.0.0` (base)
- and Git commit 1f792f52e7842eed345373ee8554af32d2aa9056 corresponding to tag `v2.0.0-alpha.1`

### Added

- ERC4626Component (#1170 ↗)
  - Special care must be taken when calling external contracts in these hooks. Please see 3.1. ↗ for more details
- `Math::u256_mul_div` (#1170 ↗)
  - u512 Precision math for multiply and divide
- The `with_components` macro (#1282 ↗, #1414 ↗)
  - Found three issues in the macro, please see 3.2. ↗ and 4.2. ↗ for more details
- Support for granting a role with delay in the AccessControl component (#1317 ↗)
- The `type_hash` macro (#1399 ↗)

### Changed

- Add SRC-107 to ERC20Component (#1294 ↗)
  - `decimals` are now configurable using the ImmutableConfig trait
- Update UDC interface and preset for backward compatibility with v1 (#1371 ↗)
  - Since the `from_zero` flag was inverted to `not_from_zero`, it might silently break the security assumption of the contracts relying on the old interface. Please see 4.3. ↗ for more details
  - Update salt hashing algorithm from Poseidon to Pedersen — it may deploy a contract to different addresses (compared to the old Universal Deployer contract) even with the same parameters given, because the hash function for the salt calculation logic has changed.
  - Add `deployContract` function to the preset

- Update ISRC6 interface to match latest changes reflected in the SNIP (#1383 ↗)
    - `__execute__` entry point now doesn't return any value
    - Account and EthAccount components SRC6 implementation updated accordingly

## 4.2. Error-handling standards in `with_component!` macro

We identified two issues in how the macro handles components:

1. First, `#[with_components()]` accepts duplicates. For example, `#[with_components(Account, Account, Account)]` generates three `Account` components.
2. Second, the macro does not guard against storage or event-name collisions. If a contract already contains `erc20: ERC20Component::Storage` and another ERC-20–based component is added, the generated code emits a second `erc20` field, and compilation fails.

Both situations are still caught by the compiler, so they do not pose a security risk at the time of this audit. However, the macro already performs extra validation for other conditions (e.g., a missing `#[starknet::contract]` attribute) that the compiler would also catch. By the same logic, the macro could either

- (a) handle **all** structural errors itself for faster feedback, or
- (b) defer **all** such checks to the compiler and focus solely on higher-level diagnostics.

Establishing a clear standard one way or the other would make error reporting more predictable and improve code maturity.

## 4.3. Silent breaking change on Universal Deployer Contract

Universal Deployer Contract provides the `deploy_contract` function. One of the parameters of this function, `not_from_zero` (in the changed code), determines whether the address of the new contract to be deployed reflects the address of the deployer or not.

We noticed that the function `deploy_contract` had the `from_zero` parameter at the same place before the scoped changes of this audit. The `from_zero` parameter behaves in an inverted way compared to the `not_from_zero` parameter — the address of the new contract to be deployed is influenced by the address of the deployer when it is set to `false`.

This interface change is incompatible with the existing contracts using Universal Deployer Contract. Furthermore, it is *silently* incompatible; if one decides to update their contract to use the latest Universal Deployer Contract from the previous one, the updated contract may functionally work (i.e., not stop working normally), but it may break the security assumptions. For example, a factory contract could set the `from_zero` parameter to `false` in order to ensure that a contract to deploy was not deployed to the same address before unless the factory contract itself deployed it. Vice versa, a factory contract may assume that a given contract is deployed by itself because the `from_zero` parameter is set to `false` and no other can deploy to the same address. These assumptions could be critical to the business logic of the contracts.

OpenZeppelin stated that this change is due to the incompatibility between the zeroth version and the first version (the base commit of the scope of this audit) of Universal Deployer Contract — the zeroth version has the `unique` parameter at the same place, which behaves like the `not_from_zero` parameter. The first version broke this by changing it to the `from_zero` parameter, which was incompatible with the zeroth version, and this change is to make it compatible to the zeroth version.

The scoped changes of this audit involve a major version bump that implies that the changes may break the compatibility. Still, we recommend to clearly mention this change in the relevant documentation (e.g., the migration guide), considering the nature of the smart contract–development ecosystem.

## 5. Assessment Results

During our assessment on the scoped OpenZeppelin Cairo contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature.

### 5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.