Zellic

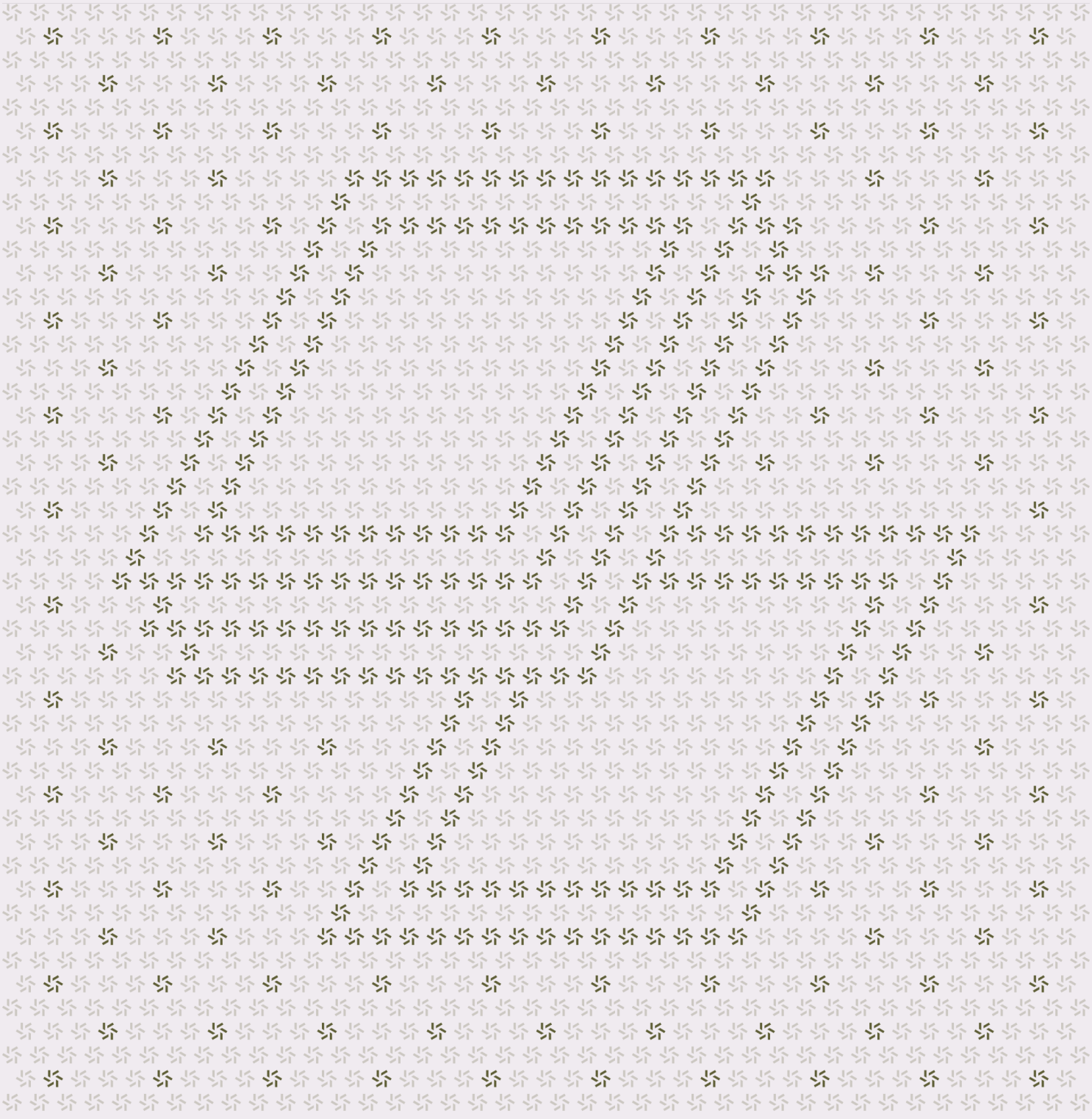
January 17, 2025

Prepared for
Starknet Foundation
OpenZeppelin

Prepared by
Junghoon Cho
Jinseo Kim
Jisub Kim
Daniel Lu
Zellic

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 November 19th, 2024, to January 17th, 2025. During this engagement, Zellic reviewed OpenZeppelin Cairo Contracts's code for security vulnerabilities, design issues, and general weaknesses in security posture. We additionally reviewed changes from select commits 38ce9d07 ↗ and 223d09d0 ↗ after the primary review period ended.

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 test suites implemented appropriately?
- How does the system verify signatures in accounts?
- Could using both Pedersen and Poseidon hashes lead to potential clashes?
- Could unexpected reentrancy behavior result in system vulnerabilities?
- Could incorrect logic (e.g., in multi-sig, timelock, or governor) lead to an invalid state?

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 contracts, we discovered eight findings. No critical issues were found. One finding was of high impact, two were of medium impact, three were of low impact, and the remaining findings were 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	1
Medium	2
Low	3
Informational	2

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 contracts.

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 contracts 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 Contracts

Type	Cairo
Platform	StarkNet
Target	Contracts
Repository	https://github.com/OpenZeppelin/cairo-contracts ↗
Version	`3fdef278da4560c017bd09a241abb468161a94ce`

Programs	`packages/access/*`
	`packages/account/*`
	`packages/finance/*`
	`packages/introspection/*`
	`packages/merkle_tree/*`
	`packages/presets/*`
	`packages/security/*`
	`packages/token/*`
	`packages/upgrades/*`
	`packages/utils/*`

Target	Contracts
Repository	https://github.com/OpenZeppelin/cairo-contracts ↗
Version	`79391c619773745f397972420017096facb9a870`
Programs	`packages/governance/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 7.6 person-weeks. The assessment was conducted by four consultants over the course of eight calendar weeks.

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:

Junghoon Cho
Engineer
junghoon@zellic.io ↗

Jinseo Kim
Engineer
jinseo@zellic.io ↗

Jisub Kim
Engineer
jisub@zellic.io ↗

Daniel Lu
Engineer
daniel@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 19, 2024	Start of primary review period
November 20, 2024	Kick-off call
January 3, 2025	End of primary review period
January 13, 2024	Start of governance review period
January 17, 2024	End of governance review period

3. Detailed Findings

3.1. Type confusion on hash of operations

Target	governance/src/timelock/timelock_controller.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

TimelockController allows the operations to be scheduled and executed. An operation can be a single call or a batch of calls.

TimelockController does not store the entire operation (call(s), predecessor, and salt) in the contract storage. Instead, when the operation is scheduled, it only stores the hash of the operation, and later the executor has to provide the same operation to the `execute` or `execute_batch` functions so TimelockController can calculate the hash of the given operation and validate that it is scheduled correctly.

```
fn schedule(
    ref self: ComponentState<TContractState>,
    call: Call,
    predecessor: felt252,
    salt: felt252,
    delay: u64,
) {
    // (...)
    let id = Self::hash_operation(@self, call, predecessor, salt);
    self._schedule(id, delay);
    // (...)
}

fn schedule_batch(
    ref self: ComponentState<TContractState>,
    calls: Span<Call>,
    predecessor: felt252,
    salt: felt252,
    delay: u64,
) {
    // (...)
    let id = Self::hash_operation_batch(@self, calls, predecessor, salt);
    self._schedule(id, delay);
    // (...)
}
```

```
fn execute(
    ref self: ComponentState<TContractState>,
    call: Call,
    predecessor: felt252,
    salt: felt252,
) {
    // (...)
    let id = Self::hash_operation(@self, call, predecessor, salt);
    // (...)
}

fn execute_batch(
    ref self: ComponentState<TContractState>,
    calls: Span<Call>,
    predecessor: felt252,
    salt: felt252,
) {
    // (...)
    let id = Self::hash_operation_batch(@self, calls, predecessor, salt);
    // (...)
}
```

Here, there are two functions that hash the given operation: `hash_operation` and `hash_operation_batch`. Because the hash calculated from these functions are used indistinguishably, it is important to ensure that the possible return values of two functions do not overlap or only overlap when both operations are interchangeable. The actual hashing algorithm of these two functions can be found in these codes.

```
// File: governance/src/utils/call_impls.cairo

pub impl HashCallImpl<S, +HashStateTrait<S>, +Drop<S>> of Hash<Call, S> {
    fn update_state(mut state: S, value: Call) -> S {
        let Call { to, selector, calldata } = value;
        state =
    state.update_with(to).update_with(selector).update_with(calldata.len());
        for elem in calldata {
            state = state.update_with(*elem);
        };

        state
    }
}

pub impl HashCallsImpl<S, +HashStateTrait<S>, +Drop<S>> of Hash<Span<Call>, S>
    {
```

```
    fn update_state(mut state: S, value: Span<Call>) -> S {
        state = state.update_with(value.len());
        for elem in value {
            state = state.update_with(*elem);
        };
        state
    }
}

// File: governance/src/timelock/timelock_controller.cairo

fn hash_operation(
    self: @ComponentState<TContractState>, call: Call, predecessor: felt252,
    salt: felt252
) -> felt252 {
    PedersenTrait::new(0)
        .update_with(call)
        .update_with(predecessor)
        .update_with(salt)
        .finalize()
}

fn hash_operation_batch(
    self: @ComponentState<TContractState>,
    calls: Span<Call>,
    predecessor: felt252,
    salt: felt252
) -> felt252 {
    PedersenTrait::new(0)
        .update_with(calls)
        .update_with(predecessor)
        .update_with(salt)
        .finalize()
}
```

Here, however, because the way `Span<Call>` and `Call` are hashed, one can create two different operations that have different calls but derive into the same hash. For example, `hash_operation(Call { to: 1, selector: 123, calldata: array![1, 456].span() }, 789, 135)` would be equal to `hash_operation_batch(array![Call { to: 123, selector: 2, calldata: array![456] }].span(), 789, 135)`. This is because the first hash is derived from `[1 (to), 123 (selector), 2 (length of calldata), 1 (calldata[0]), 456 (calldata[1]), 789 (predecessor), 135 (salt)]` while the second hash is derived from `[1 (length of calls), 123 (to), 2 (selector), 1 (length of calldata), 456 (calldata[0]), 789 (predecessor), 135 (salt)]`, which are same.

Impact

One can execute the operation which is different from the scheduled one.

Note that this finding is only practically exploitable when there are contracts with low addresses in the chain, because the size of the batch calls operation will be the `to` parameter of the single call operation.

Recommendations

Consider modifying the hashing functions in the way they do not produce same hash values for different inputs.

Remediation

This issue has been acknowledged by OpenZeppelin, and a fix was implemented in commit `9b316967` ↗.

3.2. Incorrect check logic in `_change_quorum`

Target	governance/src/multisig/multisig.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

```
fn _change_quorum(ref self: ComponentState<TContractState>, new_quorum: u32) {
    let SignersInfo { quorum: old_quorum, signers_count } = self
        .Multisig_signers_info
        .read();
    if new_quorum != old_quorum {
        assert(new_quorum.is_non_zero(), Errors::ZERO_QUORUM);
        assert(new_quorum <= signers_count, Errors::QUORUM_TOO_HIGH);
        self.Multisig_signers_info.write(SignersInfo { quorum: new_quorum,
    signers_count });
        self.emit(QuorumUpdated { old_quorum, new_quorum });
    }
}
```

The `_change_quorum` function is internally called to set a new quorum, after the `add_signers` or `remove_signers` functions add or remove signers. The function validates the new quorum value to ensure it is not zero and that the quorum does not exceed the number of signers required to execute the transaction. However, this validation is performed only when the quorum value has been changed.

Impact

Even if the quorum has not been changed, if signers are removed in `remove_signers`, a situation may arise where the quorum exceeds the updated number of signers. For example, in a 4-of-5 multisig contract, if the quorum remains at 4 while 2 signers are removed, the quorum will exceed the number of signers (3), potentially leading to the contract becoming permanently inaccessible.

Recommendations

Modify the `_change_quorum` function to ensure that the two checks within the `if` statement are always performed.

Remediation

This issue has been acknowledged by OpenZeppelin, and a fix was implemented in commit bc9b5097 ↗.

3.3. Derivation of timelock salt may revert in rare cases

Target	governance/src/governor/extensions/governor_timelock_execution.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

When a transaction is executed in Governor with GovernorTimelockExecutionComponent, the salt of the operation (which is provided to the timelock that executes the operation) is calculated with the following `timelock_salt` function:

```
/// Computes the `TimelockController` operation salt as the XOR of
/// the governor address and `description_hash`.
///
/// It is computed with the governor address itself to avoid collisions across
/// governor instances using the same timelock.
fn timelock_salt(
    self: @ComponentState<TContractState>, description_hash: felt252
) -> felt252 {
    let description_hash: u256 = description_hash.into();
    let this: felt252 = starknet::get_contract_address().into();

    // Unwrap is safe since the u256 value came from a felt252.
    (this.into() ^ description_hash).try_into().unwrap()
}
```

This function returns the XOR of the governor address and the description hash. Because the bitwise XOR is not supported in the type `felt252`, the values are converted into the type `u256` to perform XOR. The XORed value will be reconverted into the type `felt252`. The comment claims that it is safe to unwrap the XORed value because both operands fit in the range of `felt252`.

However, this claim is not always true, because the felt is defined as a non-negative integer less than the prime $2^{251} + 17 \times 2^{192} + 1$. Both 2^{251} and 2^{250} fit in this range, but the XOR of these two values, $2^{251} + 2^{250}$, does not fit.

Impact

Because this XORed value is converted into `felt252` with `unwrap()`, it will revert in the explained cases. Note that the explained cases will be rare, because this only happens when one of the operands are greater than $2^{251} - 1$. Only `2^{-55}` of possible felt values are within this range.

If the description hash is greater than $2^{251} - 1$, a user can simply change the description to fix the issue. If the governor address is greater than $2^{251} - 1$, a user would have to perform 55-bit brute force in order to submit an operation.

Recommendations

Consider fixing the algorithm of the `timelock_salt` function. For example, the XOR operation can be replaced by addition.

Remediation

This issue has been acknowledged by OpenZeppelin, and a fix was implemented in commit `a789f50b` ↗.

3.4. Incorrect packing of `SignersInfo`

Target	governance/src/multisig/storage_utils.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

```
const _2_POW_32: NonZero<u128> = 0xffffffff;
[...]

pub impl SignersInfoStorePacking of StorePacking<SignersInfo, u128> {
    fn pack(value: SignersInfo) -> u128 {
        let SignersInfo { quorum, signers_count } = value;
        quorum.into() * _2_POW_32.into() + signers_count.into()
    }

    fn unpack(value: u128) -> SignersInfo {
        let (quorum, signers_count) = u128_safe_divmod(value, _2_POW_32);
        SignersInfo {
            quorum: quorum.try_into().unwrap(), signers_count:
    signers_count.try_into().unwrap()
        }
    }
}
```

The implementation of the `StorePacking` trait enables the `SignersInfo` struct, which stores the quorum and the number of signers, to be packed into and unpacked from a `u128` type. This is done by packing it as the quorum multiplied by (2^{32}), added to the `signers_count`. During unpacking, the value is extracted by dividing by (2^{32}) to obtain the quorum and using the remainder to get the `signers_count`.

However, the `_2_POW_32` constant, which defines the value of (2^{32}), is set to `0xffffffff` instead of `0x100000000`.

Impact

When `signers_count` is `0xffffffff`, the quorum is unpacked as the original packed value plus 1, and `signers_count` is unpacked as 0.

Recommendations

Modify the `_2_POW_32` constant, defined as `0xffffffff`, to `0x100000000`. It is recommended to consider backward compatibility to ensure that data from versions prior to the bug fix can still be unpacked correctly.

Remediation

This issue has been acknowledged by OpenZeppelin, and a fix was implemented in commit `5b9509eb` ↗.

This finding was brought to our attention by OpenZeppelin prior to the official report being submitted.

3.5. Token initializers assume all interfaces are exposed

Target	token/src/erc1155/erc1155.cairo, token/src/erc721/erc721.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The ERC-1155 and ERC-721 components give developers implementations for interfaces analogous to Ethereum's multitoken and NFT standards. These also include the optional `ERC1155MetadataURI` and `ERC721Metadata` traits, which can be embedded to expose additional token data. In the case of `ERC721Component`, this includes the NFT name, symbol, and token URIs.

Each of these components requires that the base contract includes `SRC5Component`. The initializers for `ERC1155Component` and `ERC721Component` both register two interfaces on this component: the base interface and the metadata interface.

However, exposing the metadata interface is explicitly optional. If a user does not do so, the initializer will incorrectly register the metadata interface anyway.

Impact

This can lead to confusion by external contracts and user interfaces that use the SRC5 interface to determine a contract's capabilities. Namely, they might mistakenly assume that a token supports metadata queries when it does not.

Recommendations

We recommend introducing an initialization process that correctly registers the interfaces that are actually exposed by the contract.

Remediation

OpenZeppelin acknowledged this issue and adjusted the two components in commits `ff710880` and `2444726a`.

3.6. A beneficiary could accelerate their vesting by self-deposits

Target	finance/src/vesting/vesting.cairo		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The Vesting component allows developers to implement custom vesting schedules by defining their own logic through the `VestingScheduleTrait`. While this provides flexibility, it does not prevent developers from creating schedules that could interact poorly with permissionless deposits. For example, if a vesting curve fails to satisfy the property `vested_t(allocation) + x < vested_t(allocation + x)`, a beneficiary could accelerate their vesting by depositing additional funds. This creates a potential exploit vector and compromises the intended behavior of the vesting mechanism.

Impact

The implemented vesting schedules could allow beneficiaries to manipulate the vesting process, such as accelerating their vested amounts through self-deposits, thus undermining the integrity of the vesting mechanism.

Recommendations

We recommend documenting clearly the risks associated with the designed vesting curves and emphasizing the importance of satisfying the invariants in custom schedules.

Remediation

This issue has been acknowledged by OpenZeppelin. OpenZeppelin has considered the approach of resolving the vested fraction instead of using an absolute amount. However, they noted that this method performs less effectively for certain custom vesting strategies where calculations rely on the total allocation and the absolute amount vested so far. They acknowledged the validity of the concern and agreed to address this in the documentation of the Vesting component.

3.7. Potential overflow with signed integers in average

Target	utils/src/math.cairo		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `average` function calculates the average of two inputs using bitwise operations. The implementation relies on the `BitAnd` and `BitXor` traits, which are currently not supported for signed integers. As a result, the function does not currently accept signed integers as inputs.

```
/// Returns the average of two numbers. The result is rounded down.
pub fn average<
    T,
    impl TDrop: Drop<T>,
    impl TCopy: Copy<T>,
    impl TAdd: Add<T>,
    impl TDiv: Div<T>,
    impl TBitAnd: BitAnd<T>,
    impl TBitXor: BitXor<T>,
    impl TInto: Into<u8, T>
>(
    a: T, b: T
) -> T {
    // (a + b) / 2 can overflow.
    (a & b) + (a ^ b) / 2_u8.into()
}
```

However, this could become a potential issue if support for `BitAnd` and `BitXor` traits is introduced for signed integers in the future. In such a case, passing signed integers as inputs to this function may lead to unexpected behavior or incorrect results.

Impact

Although there is no immediate impact, the behavior of this function may change if the language evolves to allow `BitAnd` and `BitXor` operations for signed integers. This could lead to unintended results when calculating averages with signed inputs.

Recommendations

We recommend tightening the bounds of this function, because the fact that a type supports addition, division, and bitwise operations does not guarantee that this function will work correctly.

Remediation

This issue has been acknowledged by OpenZeppelin, and a fix was implemented in commit b6b3bcb7 ↗.

3.8. Removed signer can revoke the confirmation

Target	governance/src/multisig/multisig.cairo		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

```
fn revoke_confirmation(ref self: ComponentState<TContractState>, id:
    TransactionID) {
    let caller = starknet::get_caller_address();
    self.assert_tx_exists(id);
    assert(!self.is_executed(id), Errors::TX_ALREADY_EXECUTED);
    assert(self.is_confirmed_by(id, caller), Errors::HAS_NOT_CONFIRMED);

    self.Multisig_tx_confirmed_by.write((id, caller), false);
    self.emit(ConfirmationRevoked { id, signer: caller });
}
```

A signer who previously confirmed a transaction can revoke their confirmation later using the `revoke_confirmation` function. The `revoke_confirmation` function does not check whether the caller is an active signer, allowing a removed signer to revoke their confirmation.

Although this has no practical effect since the `get_transaction_confirmations` function only counts confirmations from active signers, the `ConfirmationRevoked` event is still emitted.

Impact

Although it has no impact on the counting of confirmations, emitting an event to indicate that a confirmation has been revoked could lead to unnecessary confusion.

Recommendations

Add a check in the `revoke_confirmation` function to ensure that the caller is an active signer. However, since this does not affect critical on-chain state and is primarily intended to prevent unnecessary event emission, this issue can be addressed based on gas cost considerations.

Remediation

OpenZeppelin explained that considering the gas overhead and the additional checks required when displaying information in the off-chain UI, emitting an event in such situations is acceptable.

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. Reentrancy protection logic in timelock

When the TimelockController component executes the scheduled operation, it is important to check and prevent the reentrancy accordingly so the given operation cannot be executed multiple times unexpectedly.

The `execute` and `execute_batch` functions in the TimelockController component also have such check, but it is slightly different from the usual practice:

```
/// Executes a (Ready) operation containing a single Call.
///
/// (...)
///
/// NOTE: This function can reenter, but it doesn't pose a risk because
    `_after_call`
/// checks that the proposal is pending, thus any modifications to the
    operation during
/// reentrancy should be caught.
///
/// (...)
fn execute(
    ref self: ComponentState<TContractState>,
    call: Call,
    predecessor: felt252,
    salt: felt252
) {
    self.assert_only_role_or_open_role(EXECUTOR_ROLE);

    let id = Self::hash_operation(@self, call, predecessor, salt);
    self._before_call(id, predecessor);
    self._execute(call);
    self.emit(CallExecuted { id, index: 0, call });
    self._after_call(id);
}

fn _before_call(self: @ComponentState<TContractState>, id: felt252,
    predecessor: felt252) {
    assert(Timelock::is_operation_ready(self, id),
    Errors::EXPECTED_READY_OPERATION);
    assert(
        predecessor == 0 || Timelock::is_operation_done(self, predecessor),
        Errors::UNEXECUTED_PREDECESSOR
```

```
    );
}

fn _after_call(ref self: ComponentState<TContractState>, id: felt252) {
    assert(Timelock::is_operation_ready(@self, id),
    Errors::EXPECTED_READY_OPERATION);
    self.TimelockController_timestamps.write(id, DONE_TIMESTAMP);
}

fn is_operation_ready(self: @ComponentState<TContractState>, id: felt252) ->
    bool {
    Self::get_operation_state(self, id) == OperationState::Ready
}
```

Here, the function `_before_call` checks that the operation is ready, not changing any state. The function `_after_call` rechecks that the operation is still in the ready state, and changes the state into done.

As mentioned in the comment, this function can reenter, but the checks after the reenterancy can detect the reentrancy, because the state of the operation must be "ready" after the operation. If the operation reenters and executes twice, the outer execution will not pass the check of `_after_call` and will be reverted.

However, it is notable that this check is theoretically not perfect. Let's assume the reentrancy (executing the operation twice) allows the malicious party to upgrade the contract into the arbitrary one. The malicious party can then change the state of the operation to "ready" after the inner execution changes the state of the operation to "done".

One possible contract that could be vulnerable to this exploit would have the queue of the upgrade candidates. An admin, who can be malicious under the extreme threat model of timelock, can propose a new upgrade by pushing the class hash to this queue, and the function `upgrade()`, which is behind the timelock, pops the class hash from the queue, applies it, and reenters. A malicious admin can (1) push a benign upgrade into the queue, (2) schedule to `upgrade()` on timelock, (3) wait until it is ready, (4) push a malicious upgrade into the queue, and (5) invoke `upgrade()` twice by exploiting the reentrancy. The upgraded (malicious) contract would have the function that changes back the state of the ongoing upgrade operation to be ready, which allows the outer `upgrade()` call to finish normally.

This issue has been acknowledged by OpenZeppelin. OpenZeppelin stated that their security model does not consider a malicious admin to be a threat.

4.2. Malicious sequencers

Since sequencers can redefine whitelisted syscalls, they can in theory act maliciously. For instance, they could incorrectly report the timestamp. We note that the correctness of these crates relies

on the assumption that sequencers are well-behaved, until a proper protocol for decentralization arrives.

4.3. Transactions do not expire once submitted

In a MultiSig contract, once a transaction is submitted for signers' confirmation, it remains available indefinitely. This is because the MultiSig logic does not implement expiration based on time or other conditions. If executing a specific transaction outside a certain time frame could lead to significant losses, this implementation could pose a problem.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Configuring components

Many of OpenZeppelin's contracts are Cairo components that encapsulate state and logic for a base contract to use. But currently, components do not support parameterization, which poses a problem when they need to be configured. OpenZeppelin handles this in two different ways: configuring at runtime and configuring through traits.

Configuration at runtime

A number of components provide an `initializer` function that writes some configuration data into state. For instance, `ERC20Component` has one such function that sets the token name and symbol.

```
/// Initializes the contract by setting the token name and symbol.
/// To prevent reinitialization, this should only be used inside of a contract's
/// constructor.
fn initializer(
    ref self: ComponentState<TContractState>, name: ByteArray, symbol:
    ByteArray
) {
    self.ERC20_name.write(name);
    self.ERC20_symbol.write(symbol);
}
```

Another component that uses this pattern is `VestingComponent`, which has the vesting period and cliff set through an initializer.

```
/// Initializes the component by setting the vesting `start`, `duration` and
/// `cliff_duration`.
///
/// Requirements:
///
/// - `cliff_duration` must be less than or equal to `duration`.
fn initializer(
    ref self: ComponentState<TContractState>, start: u64, duration: u64,
    cliff_duration: u64
) {
```

```
    self.Vesting_start.write(start);
    self.Vesting_duration.write(duration);

    assert(cliff_duration <= duration, Errors::INVALID_CLIFF_DURATION);
    self.Vesting_cliff.write(start + cliff_duration);
}
```

This approach might be familiar to developers who have worked with Solidity contracts, which are generally configured this way during contract creation. It has some drawbacks, however. Most importantly, developers must remember to call the `initializer` function exactly once, before using any other functionality. This means that OpenZeppelin must clearly communicate these requirements in component documentation.

Configuration through traits

Some components are configured through traits. This is usually reserved for configuring component logic, such as setting a particular vesting curve or augmenting token actions through hooks.

Concretely, the vesting component has an external interface polymorphic over a `VestingScheduleTrait`, which must offer the following API.

```
/// A trait that defines the logic for calculating the vested amount based on a
    given timestamp.
pub trait VestingScheduleTrait<TContractState> {
    /// Calculates and returns the vested amount at a given `timestamp` based
    on the core
    /// vesting parameters.
    fn calculate_vested_amount(
        self: @ComponentState<TContractState>,
        token: ContractAddress,
        total_allocation: u256,
        timestamp: u64,
        start: u64,
        duration: u64,
        cliff: u64,
    ) -> u256;
}
```

When embedding the component's `Vesting` interface, developers choose a vesting schedule by implementing this trait or importing the provided `LinearVestingSchedule`.

Another example is token hooks. Looking at ERC-721, we see that all component logic is polymorphic over an `ERC721HooksTrait`.

```
pub trait ERC721HooksTrait<TContractState> {
    fn before_update(
        ref self: ComponentState<TContractState>,
        to: ContractAddress,
        token_id: u256,
        auth: ContractAddress
    ) {}

    fn after_update(
        ref self: ComponentState<TContractState>,
        to: ContractAddress,
        token_id: u256,
        auth: ContractAddress
    ) {}
}
```

During actions that update a token, the component will call these hooks.

```
fn update(
    ref self: ComponentState<TContractState>,
    to: ContractAddress,
    token_id: u256,
    auth: ContractAddress
) -> ContractAddress {
    Hooks::before_update(ref self, to, token_id, auth);

    // ...

    Hooks::after_update(ref self, to, token_id, auth);
```

Just like with vesting schedules, developers can implement custom hooks or use the provided default, which does nothing.

This approach has the benefit that all configuration is done through the type system and happens statically. However, it can be a bit more complex for developers to understand and use. More importantly, it is worth nothing that this does not work by making the component polymorphic over the configuration; it works by making its *traits* polymorphic over the configuration. This means that there is nothing stopping a developer from feeding different configurations to different parts of a component's interface, which could lead to unexpected behavior.

OpenZeppelin explained that the intended way to configure a component is to simply bring the trait into scope rather than explicitly pass it as a type parameter; developers who do so are not susceptible to this issue.

Presets

OpenZeppelin provides a number of preset contracts that use preconfigured components. For example, there is an ERC20Upgradeable contract that uses `OwnableComponent`, `ERC20Component`, and `UpgradeableComponent` together in order to set up an ERC-20 token where an owner can upgrade the contract and manage ownership.

5.2. Contract architecture

We will briefly discuss how each of the crates is designed, some requirements for their correctness, and whether these requirements are well-tested.

Access

The access crate includes a component for managing roles and a component for managing ownership. The first, called `AccessControlComponent`, has roles identified with `felt252` values. Each role has a single admin and a collection of members. By default, the public interface allows a role admin to grant or revoke the role from a member. A member can also renounce their role.

This component directly expects `felt252` values from developers and users, which simplifies the implementation and removes deserialization overhead. But we note that this approach could lead to errors if developers mix up role IDs across contracts.

The second component, `OwnableComponent`, keeps track of the owner address for the contract. The developer can choose between two implementations: one where ownership transfer completes immediately and another where the new owner must accept the transfer. In both cases, the component uses the zero address to represent no owner — for instance, when ownership is revoked. This should not be an issue in the current version of Starknet because the zero address will never be a caller, but we recommend using `Option<ContractAddress>` instead for clarity.

We point out that while revoking ownership and canceling ownership transfer both set the owner to the zero address, the former has a dedicated function while the latter does not. Additionally, neither of these actions emit special events and instead result in `OwnershipTransferred` and `OwnershipTransferStarted` to the zero address.

The correctness of access control is fairly straightforward: the `has_role` function should correctly check if a member has a role (with respect to grants and revocations), and only role admins should be able to grant roles or revoke them for others. Similarly, the ownership component should simply ensure that ownership retrieval and assertions are correct with respect to initialization and transfers. These facts are well-covered by the provided test cases.

However, **we recommend changing all access-control tests to call the `initializer` first** (possibly in `setup`) because this is how the component is intended to be used.

Account

Starknet accounts are smart contracts that validate and execute transactions. Unlike Ethereum's externally owned accounts (EOAs), Starknet features native-account abstraction, where all accounts are contracts.

This architecture is defined by the SNIP-6 standard, which specifies an API for sending transactions and validating signatures, ensuring interoperability with dApps, protocols, and other accounts. For more details on account abstraction, the Starknet Account Abstraction community post ↗ provides a comprehensive overview.

The account component provides logic for contracts to act as accounts, supporting advanced features such as multicall transactions, off-chain signing, and Ethereum-style signatures. At its core, it adheres to the SNIP-6 standard, which provides a consistent execution flow for accounts and ensures seamless interaction with dApps.

The SNIP-6 proposal defines the structure and functionality of compliant accounts. A key part of this standard is the `Call` struct, representing a transaction's details:

```
/// Represents a call to a target contract function.
struct Call {
    to: ContractAddress,
    selector: felt252,
    calldata: Array<felt252>
}
```

The core interface for SNIP-6–compliant accounts includes the following methods:

```
/// Standard Account Interface
#[starknet::interface]
pub trait ISRC6 {
    /// Executes a transaction through the account.
    fn __execute__(calls: Array<Call>) -> Array<Span<felt252>>;

    /// Asserts whether the transaction is valid to be executed.
    fn __validate__(calls: Array<Call>) -> felt252;

    /// Asserts whether a given signature for a given hash is valid.
    fn is_valid_signature(hash: felt252, signature: Array<felt252>) ->
    felt252;
}
```

To optimize performance, the calldata field of the `Call` struct has been updated to `Span<felt252>`. However, the interface ID remains unchanged for backward compatibility, and this inconsistency will be addressed in future releases. Additionally, accounts must implement the SRC5 interface, defined by SNIP-5, to allow introspection for verifying whether a contract functions as an account.

The Starknet protocol uses several entry points to abstract account operations:

- `__validate__` — validates transactions, typically for signature verification, but also allows for custom validation mechanisms with some limitations
- `__execute__` — executes a transaction after successful validation
- `__validate_declare__` (optional) — validates transactions intended to declare other contracts
- `__validate_deploy__` (optional) — validates counterfactual deployment transactions

There is a critical validation in `execute` that prevents account-fund draining. Two essential validations occur in the `__execute__` function:

1. The caller check, which ensures the function is not called from another contract, `(get_caller_address().is_zero())`, preventing skipped validations
2. The transaction-version check, which confirms the transaction version is at least 1 `(get_tx_info().version >= 1)`, disallowing deprecated v0 transactions that bypass `__validate__` and rely on outdated signature verification

Ownership transfer can only be initiated by the current owner with a valid signature from the new owner.

The `execute_from_outside_v2` allows reentrancy for nested calls while ensuring state consistency via strict nonce checks. Nonces must be unique and are marked as used after execution to prevent replay attacks.

Signature validation ensures the caller matches the transaction details, supporting both Ethereum-style and StarkNet-native formats.

The robustness of the account component is ensured by comprehensive test coverage that validates critical aspects such as transaction execution, nonce management, signature validation, and so on. To further strengthen this coverage, we recommend adding edge test cases (e.g., a test verifying that very large `calldata` does not cause any errors during execution or would be handled properly).

Introducing a negative test for malformed `multicalls` would ensure that such scenarios fail gracefully without disrupting other valid calls within the same execution. Additionally, tests should address cases where `calldata` contains invalid or malformed values, such as incorrect data types, ensuring that such inputs are correctly identified and handled as errors.

Finally, adding a test to verify the behavior when attempting to set an invalid public key — particularly one involving invalid points in the `Secp256Point` — would help ensure the component can effectively handle such cases.

Finance

The vesting component provides logic for gradually releasing tokens for a designated beneficiary. The component requires that the parent contract is ownable and simply takes the owner as the re-

cipient of vested funds.

During construction, the component expects a start time, a duration, and a cliff. This data is passed to the configurable vesting schedule, which computes the absolute amount of tokens that should be released at any given time. The beneficiary has the right to call `release`, which always sends them the full vested amount.

The vesting schedule adapts to the total allocation, meaning that additional funds can be deposited to be vested. It tracks this total allocation by taking the sum of the contract balance and released amount.

We note that because the vesting component relies on the ownership component, it is not possible for the base contract to use the ownership component for anything else. Additionally, the ownership component exposes by default some functions that might be undesirable, such as `renounce`. Evidently, if the beneficiary calls `renounce`, they will no longer be able to withdraw their vested funds.

Vesting must work properly

- through ownership changes;
- with different vesting schedules;
- throughout the contract lifetime, including after the vesting period ends; and
- as the contract balance changes.

While most of these these situations are tested in the provided codebase, we recommend adding a test verifying that vesting correctly adapts to future deposits into the contract.

Governance

Governor

The Governor crate provides the on-chain governance system. The core concepts of Governor are proposal and vote: Once a proposal is submitted to the Governor component, the voting process starts after the voting delay for the configured voting period. During the vote period, users can vote on the proposal. If the vote succeeds, the proposal can be executed. GovernorComponent implements this core logic.

This component requires the detail of this logic to be defined. Specifically, how long are the voting delay and the voting period, how many votes are needed for a proposal, how the votes are counted, how the voting power of a user is calculated, and how a successful proposal is executed, are not defined in the GovernorComponent, and these should be defined in the extensions that implement GovernorSettingsTrait, GovernorQuorumTrait, GovernorCountingTrait, GovernorVotesTrait, and GovernorExecutionTrait respectively. This crate provides some examples of extensions that implement these traits, but the deployer of the GovernorComponent may decide to implement these traits by themselves.

GovernorSettingsTrait must implement the `voting_delay`, `voting_period`, and `proposal_threshold` functions, which returns the voting delay (the delay between the submission of the proposal and the beginning of the voting period), the duration of the voting period, and the

proposal threshold (the minimum voting power to submit a proposal). GovernorSettingsComponent implement this trait, allowing the configuration to be changed by the governance itself.

GovernorQuorumTrait must implement the `quorum` function, which returns the quorum to have the proposal be successful. GovernorVotesQuorumFractionComponent implements this trait with the `quorum` function that calculates the quorum as the configured fraction of the total supply, where the fraction can be changed by the governance itself.

GovernorCountingTrait must implement the `counting_mode`, `count_vote`, `has_voted`, `quorum_reached`, and `vote_succeeded` functions. GovernorCountingSimpleComponent implements this trait, providing the three voting options (For/Against/Abstain) to a voter, counting For and Abstain votes toward quorum, determining the vote to be successful when the For votes are greater than the Against votes.

GovernorVotesTrait must implement the `clock_mode`, `clock`, and `get_votes` functions. Both GovernorVotes and GovernorVotesQuorumFractionComponent implement this trait in the way querying to the token contract that implements the IVotes interface to calculate the voting power of a user.

GovernorExecutionTrait must implement the `state`, `executor`, `execute_operations`, `queue_operations`, `proposal_needs_queuing`, and `cancel_operations` functions. GovernorCoreExecutionComponent implements this trait in the way the governance contract directly executes the given calls, while GovernorTimelockExecutionComponent indirectly executes the given calls through the contract that implements the TimelockController.

MultiSig

The multi-signature mechanism is an essential component for enhancing both the security and governance of smart contract transactions. If a protocol's administration relies on the decision of a single entity, it poses a significant security risk in case that entity is compromised.

The MultiSig crate provides a basic multi-signature implementation, allowing multiple signers to be registered in the contract. When a transaction is submitted, it requires approval from a defined number of signers—known as the quorum—before it can be executed.

Once a transaction is introduced via the `submit_transaction` function, registered signers can approve it using the `confirm_transaction` function. Additionally, confirmations can be revoked at any time based on the signer's discretion.

When the required quorum of confirmations is met, the transaction becomes executable. After execution, the contract records its execution state to prevent duplicate processing.

Signers, identified by their contract addresses, can be freely added or removed using the `add_signers` and `remove_signers` functions. The quorum can also be modified via the `change_quorum` function, as long as the new value does not exceed the total number of registered signers. By default, the component is self-administered, meaning that any modifications to signers or quorum must go through the same multi-signature approval process as other transactions.

For user convenience, the crate provides functions to query the number of signers who have confirmed a transaction, the confirmation status of each signer, and the execution state of transactions.

Timelock

Decentralization is a core principle of blockchain protocols, especially in DeFi, where financial activities take place. Various administrative operations performed by a protocol's owner can directly affect users financially.

The Timelock mechanism introduces a delay before protocol changes are applied, allowing users time to make decisions, take action, or even exit before the changes take effect. This eliminates the need for users to place unnecessary trust in protocol operators, thereby promoting greater decentralization.

This component operates based on three key roles:

- PROPOSER_ROLE : Schedules transactions that will be executed after the delay period.
- CANCELLER_ROLE : Cancels scheduled transactions before they are executed.
- EXECUTOR_ROLE : Executes transactions once the delay period has elapsed. This role can be set as an open role by granting it to the zero address, allowing anyone to trigger execution.

Different transactions can have different delay periods, but each delay must be at least the minimum delay defined by the contract. This minimum delay parameter can be updated using the `update_delay` function, but any modifications to it must also go through the timelock mechanism before taking effect.

Votes

The Votes crate allows users to mint voting power, track it, and delegate their voting power to others.

To integrate this component, the `VotingUnitsTrait` must be implemented to define how voting units are assigned to a given account. Default implementations are provided for ERC20 and ERC721 tokens, where ERC20 voting units are determined by token balances, and ERC721 voting units correspond to the number of tokens owned. For proper tracking of voting power, the `transfer_voting_units` function must be called whenever a transfer, mint, or burn operation occurs.

The Votes component operates based on a checkpoint system. Every time an event such as minting, burning, or transferring occurs, the system records the changes in voting units at that specific time. This allows users to retrieve past voting power balances using dedicated query functions.

Additionally, users can delegate their voting power via off-chain signatures following the SNIP12 standard. This functionality is provided by the `delegate_by_sig` function, which first validates whether the signature is expired or has already been used before delegating the voting units.

Introspection

Standards often require smart contracts to implement introspection mechanisms ↗ for smooth interoperability.

In Ethereum, the EIP-165 standard defines how contracts should declare their support for a given interface and how other contracts may query this support. Similarly, Starknet provides a mechanism for interface introspection through the SRC5 standard.

The SRC5 component allows contracts to expose and manage the interfaces they implement. It provides functionality to register, deregister, and verify interface support efficiently, ensuring compliance with the SRC5 standard.

The SRC5 standard defines the interface identifier as the XOR of all extended function selectors in the interface. For example,

```
struct Call {
    to: ContractAddress,
    selector: felt252,
    calldata: Array<felt252>
}

trait IAccount {
    fn supports_interface(felt252) -> bool;
    fn is_valid_signature(felt252, Array<felt252>) -> bool;
    fn __execute__(Array<Call>) -> Array<Span<felt252>>;
    fn __validate__(Array<Call>) -> felt252;
    fn __validate_declare__(felt252) -> felt252;
}
```

Python can compute the interface ID using the `starknet_keccak` function:

```
from starkware.starknet.public.abi import starknet_keccak

extended_function_selector_signatures_list = [
    'supports_interface(felt252)->E((),())',
    'is_valid_signature(felt252,Array<felt252>)->E((),())',
    '__execute__(Array<(ContractAddress,felt252,Array<felt252>)>)-
    >Array<(@Array<felt252>)>',
    '__validate__(Array<(ContractAddress,felt252,Array<felt252>)>)->felt252',
    '__validate_declare__(felt252)->felt252'
]

def main():
    interface_id = 0x0
    for function_signature in extended_function_selector_signatures_list:
        function_id = starknet_keccak(function_signature.encode())
        interface_id ^= function_id
    print(f'IAccount ID: {hex(interface_id)}')

if __name__ == "__main__":
```

```
    main()
```

For SRC5, the identifier for the introspection interface ISRC5 is `0x3f918d17e5ee77373b56385708f855659a07f75997f365cf87748628532a055`.

An SRC5-compliant contract must implement the following interface:

```
trait ISRC5 {
    fn supports_interface(interface_id: felt252) -> bool;
}
```

The `supports_interface` function returns

- true for the ISRC5 interface ID `0x3f918d17e5ee7737...55`,
- true for any other implemented interface ID, and
- false for unsupported interface IDs.

We found that ERC-721 and ERC-1155 support SRC5 but ERC-20 does not. However, OpenZeppelin mentioned that they did not implement it because EIP-165 is not part of the EIP-20 standard.

The tests for these components are fairly comprehensive; we recommend adding edge cases for interface IDs such as 0, the maximum `felt252` value, or negative values.

Merkle trees

OpenZeppelin provides a number of utility functions for working with Merkle proofs. While generic over any commutative hash `(felt252, felt252) -> felt252`, the implementations for a commutative Pedersen commitment and commutative Poseidon hash are provided.

```
/// Returns true if a `leaf` can be proved to be a part of a Merkle tree
/// defined by `root`. For this, a `proof` must be provided, containing
/// sibling hashes on the branch from the leaf to the root of the tree. Each
/// pair of leaves and each pair of pre-images are assumed to be sorted.
pub fn verify<impl Hasher: CommutativeHasher>(
    proof: Span<felt252>, root: felt252, leaf: felt252
) -> bool {
    process_proof::<Hasher>(proof, leaf) == root
}

/// Version of `verify` using Perdersen as the hashing function.
pub fn verify_pedersen(proof: Span<felt252>, root: felt252, leaf: felt252) ->
    bool {
    verify::<PedersenCHasher>(proof, root, leaf)
}
```

```
/// Version of `verify` using Poseidon as the hashing function.
pub fn verify_poseidon(proof: Span<felt252>, root: felt252, leaf: felt252) ->
    bool {
    verify::<PoseidonCHasher>(proof, root, leaf)
}
```

Since this library demands that the provided hash function be commutative, proof verification is very simple. The `process_proof` function simply computes

$$\text{hash}(\text{hash}(\dots \text{hash}(\text{hash}(\text{leaf}, p_0), p_1), \dots), p_n)$$

where "leaf" is the leaf to be verified and $p_0, p_1, \dots, p_n$ is the proof of its inclusion. It also provides logic for simultaneously checking the inclusion of multiple leaves.

Something of note is how the provided hash functions are implemented. Both obtain the commutativity property by sorting the operands before hashing them. Given sorted inputs a and b, the Poseidon version simply takes the hash of (a, b). But the Pedersen version takes the commitment of $(0, a, b, 2)$, reflecting the array-hashing standard used by Starknet. This slightly interacts with a warning given by the library.

```
/// WARNING: You should avoid using leaf values that are two felt252 long prior
    to
/// hashing, or use a different hash function for hashing leaves and pre-images.
/// This is because the concatenation of a sorted pair of internal nodes in
/// the Merkle tree could be reinterpreted as a leaf value.
```

It is worth noting to developers the specific way that the Pedersen commitment and Poseidon hash are implemented to clarify what "using leaf values that are two felt252 long" means.

Beyond the correct verification of correct proofs with both hash functions, it is essential that the library correctly rejects invalid proofs. The library by design cannot reject proofs of leaves that are in fact internal nodes. But it should be able to reject some invalid multiproofs; for instance, it is critical that all leaves are consumed during verification. In general, the positive and negative tests provided cover these cases well. Although there is a test case that checks the failure of a valid multiproof with a flag removed, **we recommend also testing the failure of a valid multiproof with an extra leaf**.

Security

The security crate provides components for nonconstructor initialization, reentrancy prevention, and pausing. Each component encapsulates a single flag, denoting whether the contract was initialized, needs to be locked, or is paused. They expose functions to manage these flags and assert against them.

It is worth noting that because the language does not currently support any modifier- or decorator-like syntax, the developer must do some amount of manual management. Most notably, the reentrancy-guard component is used as so.

```
fn some_function(ref self: ComponentState<TContractState>) {
    self.reentrancy_guard_component.start();

    // logic

    self.reentrancy_guard_component.end();
}
```

Forgetting to call `end` could cause the contract to be locked indefinitely. Forgetting to call `start` could introduce reentrancy issues throughout the contract. We note that a way to prevent this style of error would be to issue a resource in `start` that must be collected in `end` (i.e., cannot be otherwise dropped).

```
// proposed usage
fn some_function(ref self: ComponentState<TContractState>) {
    let obligation = self.reentrancy_guard_component.start();

    // logic

    self.reentrancy_guard_component.end(obligation);
}
```

OpenZeppelin explained that this style of error will be addressed by adopting future language features instead.

The tests for these components are fairly comprehensive, covering the entire state spaces and API surfaces of each one.

Token

The contracts include `ERC1155Component`, `ERC20Component`, `ERC721Component`, and `ERC2981Component`, which are tokens analogous to the multitoken, fungible token, nonfungible token, and NFT royalty standards in Ethereum. These simply encapsulate token state (balances, approvals, and metadata) and provide logic for the standard interfaces.

In `ERC20Permit`, since there's no `u64` type in SNIP-12, the type used for the `expiry` parameter is `u128`:

```
selector!(
  "\"Message\"(
    \"recipient\":\"ContractAddress\",
    \"amount\":\"u256\",
    \"nonce\":\"felt\",
    \"expiry\":\"u128\"
  )\"u256\"(
```

```
    \"low\":\"u128\",
    \"high\":\"u128\"
  )"
);
```

In general, these implementations must correctly

- maintain and report user balances;
- check transfer permissions, including with operators, approvals, and permits if implemented; and
- perform acceptance checks during transfers and mints, if implemented.

The provided test cases cover these requirements well, including negative cases like signature replay.

Upgrades

The upgrades crate provides a component for managing contract upgrades. The component provides a function for invoking the `replace_class` syscall. It also has a variant that follows this with a `call_contract` syscall to call a selector on the new class. Note that encapsulating no state of its own, this is a full component (rather than just a library) in order to emit the `Upgraded` event. OpenZeppelin has included thorough positive and negative test cases for this component.

5.3. Additional note: Camel-case interfaces

A number of the components include camel-case variants for backwards compatibility. For example, the SRC6 interface includes the `is_valid_signature` function as well as the wrapper camel-case variant `isValidSignature` for compatibility with protocols that still use the older convention.

OpenZeppelin remarked that the camel-case interfaces will be removed after regenesis and that this approach ensures that older protocols can continue to function without requiring immediate changes.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to mainnet.

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

6.1. 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.