Zellic

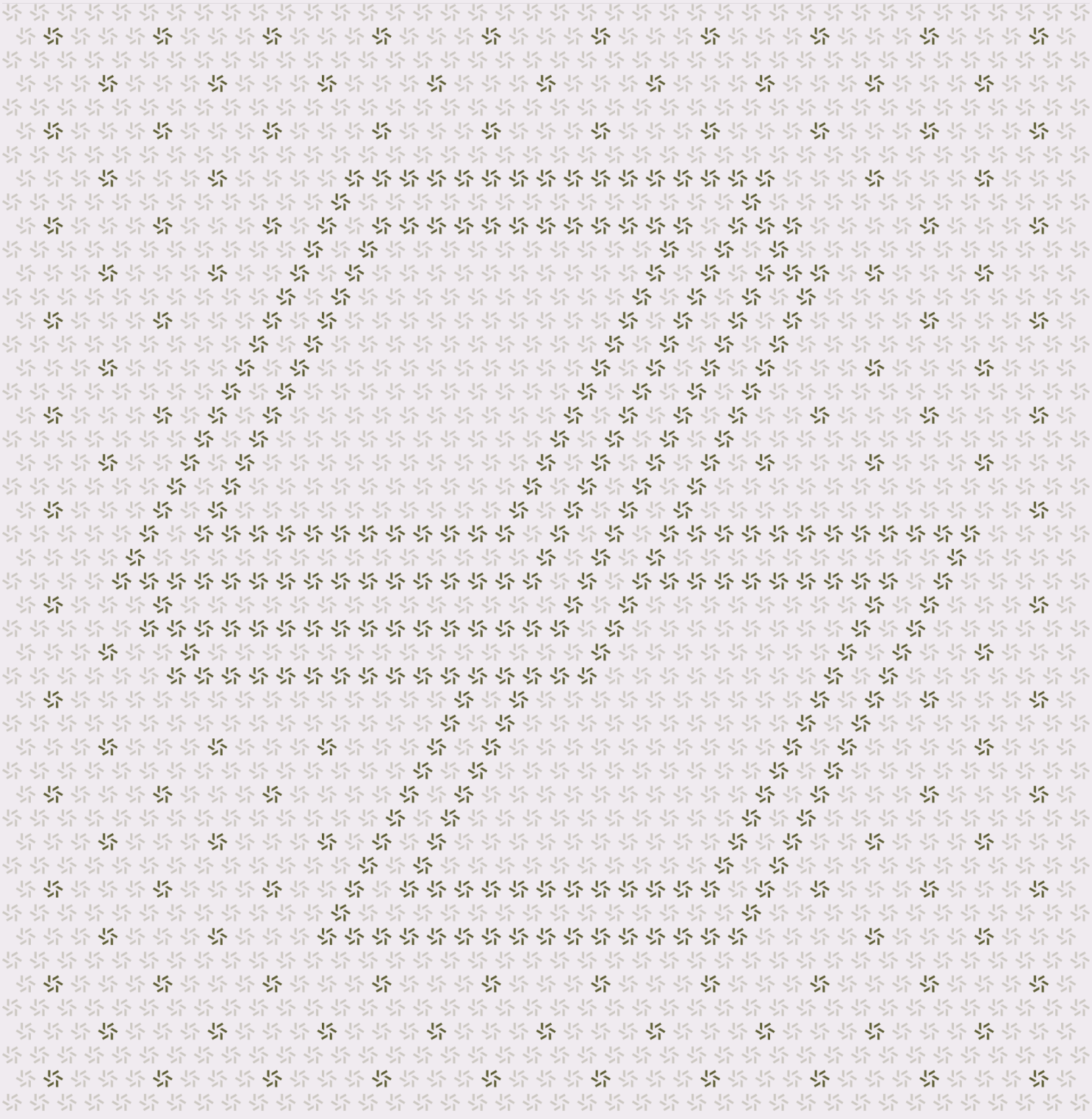
**Prepared for**
Levi Weber Costa
LayerZero Labs

**Prepared by**
Junghoon Cho
Jinseo Kim
Zellic

**September 30, 2025**

# LayerZero V2 Starknet

## Cairo 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 LayerZero Labs from September 8th to September 24th, 2025. During this engagement, Zellic reviewed LayerZero V2 Starknet's 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:

- Has LayerZero V2 Starknet been properly implemented according to the LayerZero V2 protocol specification?
- Is there a way to grief the Endpoint or ULN?
- Are there any bugs caused by the properties of Starknet when implementing the protocol in Cairo?
- Is the payment processed correctly in the interaction between OApp and the Endpoint?

## 1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody

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 LayerZero V2 Starknet targets, we discovered seven findings. One critical issue was found. Two were of medium impact, two 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 LayerZero Labs in the Discussion section (4. ↗).

## Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| Critical | 1 |
| High | 0 |
| Medium | 2 |
| Low | 2 |
| Informational | 2 |

# 2. Introduction

## 2.1. About LayerZero V2 Starknet

LayerZero Labs contributed the following description of LayerZero V2 Starknet:

> The LayerZero V2-Starknet implementation maintains the same core architecture and functionality as LayerZero V2-EVM while adapting to Starknet's unique features and constraints. The protocol preserves LayerZero's core values of permissionlessness, immutability and censorship-resistance.

## 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:

### LayerZero V2 Starknet Targets

| | |
|---|---|
| **Type** | Cairo |
| **Platform** | Starknet |
| **Target** | EPv2-Starknet |
| **Repository** | https://github.com/LayerZero-Labs/EPv2-Starknet ↗ |
| **Version** | `3d76b69ac91964e760965f4a8e2f00f2670e60e9` |
| **Programs** | `layerzero/src/*`<br>`starkgate_mint_burn/src/*`<br>`starkgate_wbtc/src/*` |
| **Target** | Refer to section 4.3 for the files added in the subsequent review periods. |

## 2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3.9 person-weeks. The assessment was conducted by two consultants over the course of 2.6 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 ↗

**Pedro Moura**
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

**Junghoon Cho**
Engineer
junghoon@zellic.io ↗

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

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **September 8, 2025** | Start of primary review period |
| **September 24, 2025** | End of primary review period |
| **October 7, 2025** | Start of secondary review period |
| **October 7, 2025** | Scope changed from `3d76b69a` to `e9e31007` |
| **October 10, 2025** | End of secondary review period |
| **November 13, 2025** | Start of tertiary review period |
| **November 13, 2025** | Scope changed from `e9e31007` to `f66a9fb4` |
| **November 14, 2025** | End of tertiary review period |
| **November 24, 2025** | Start of quaternary review period |
| **November 24, 2025** | Scope changed from `f66a9fb4` to `5353a346` |
| **November 24, 2025** | End of quaternary review period |
| **December 1, 2025** | Start of quinary review period |
| **December 1, 2025** | Scope changed from `5353a346` to `c5a5532d` |
| **December 1, 2025** | End of quinary review period |

| | |
|---|---|
| **December 17, 2025** | Start of senary review period |
| **December 17, 2025** | Scope changed from `c5a5532d` ↗ to `f6b1b803` ↗ |
| **December 22, 2025** | End of senary review period |

# 3. Detailed Findings

## 3.1. Entry points accept an invalid `ByteArray` parameter

| **Target** | * | | |
|---|---|---|---|
| **Category** | Coding Mistakes | **Severity** | Critical |
| **Likelihood** | High | **Impact** | Critical |

### Description

The `ByteArray` is the built-in type of Cairo that represents an arbitrary vector of bytes. Multiple contracts of the project have entry points that accept a user-provided `ByteArray` parameter.

Internally, the `ByteArray` type is the following struct:

```
#[derive(Drop, Clone, PartialEq, Serde, Default)]
pub struct ByteArray {
    /// An array of full "words" of 31 bytes each.
    /// The first byte of each word in the byte array is the most significant
    byte in the word.
    pub(crate) data: Array<bytes31>,
    /// A `felt252` that actually represents a `bytes31`, with less than 31
    bytes.
    /// It is represented as a `felt252` to improve performance of building the
    byte array.
    /// The first byte is the most significant byte among the
    `pending_word_len` bytes in the word.
    pub(crate) pending_word: felt252,
    /// The number of bytes in `pending_word`.
    /// Its value should be in the range [0, 30].
    pub(crate) pending_word_len: usize,
}
```

As `ByteArray` is a simple struct, some `ByteArray` values can be invalid by breaking the above assumptions. For example, one may have `pending_word_len` greater than 30.

Because the built-in functions of Cairo handling `ByteArray` values assume that all input `ByteArray` values are valid, it is important to validate such inputs when they are untrusted. However, multiple entry points of the contracts in the project do not validate user-provided `ByteArray` values.

### Impact

The behaviors of `ByteArray`-handling built-in functions in Cairo are undefined when invalid `ByteArray` values are provided. The functions utilizing such built-in functions on the user-provided `ByteArray` values may unexpectedly panic or return inconsistent results (i.e., built-in functions return the set of results that cannot be derived from any vector of bytes).

For example, two different `ByteArray` instances could have the same hash digest under `compute_keccak_byte_array` because the `ByteArray::at` function (which is used by the `compute_keccak_byte_array` function) equally behaves for `ByteArray { data: [], pending_word: 0, pending_word_len: 1 }` versus `ByteArray { data: [], pending_word: 25600, pending_word_len: 1 }`. However, the `ByteArray::len` function will return different lengths for two instances.

Such issues can be directly exploited on the `lz_receive` entry point, which is the core entry point of the endpoint that receives the payload, verifies its hash, and sends it to the recipient. Because of the reason explained above, a malicious user may submit the invalid payload to the `lz_receive` entry point in order to make the recipient use the invalid payload.

### Recommendations

Consider verifying that all user-provided `ByteArray` instances are valid.

### Remediation

This issue has been acknowledged by LayerZero Labs, who informed us that they plan to coordinate with StarkWare to resolve it by implementing a fix in the Cairo language.

StarkWare has since resolved the issue in the Cairo language via PR #8516 ↗ and PR #8518 ↗. We reviewed the fix to the best of our abilities; however, please note that our audit did not include a review of the Cairo core library beyond the fix, which limited our ability to assess the full extent of the changes.

## 3.2. Decay of amount-in-flight in rate limit is reset when flow happens at opposite direction

| | | | |
|---|---|---|---|
| **Target** | RateLimiterComponent | | |
| **Category** | Coding Mistakes | **Severity** | Medium |
| **Likelihood** | High | **Impact** | Medium |

### Description

`RateLimiterComponent` implements the rate limiting system utilized in the `OFTMintBurnAdapter` contract. `RateLimiterComponent` allows the contract to rate-limit the inflow and outflow of tokens.

Specifically, the contract internally calculates the "amount-in-flight" for both directions; when tokens flow in, the amount of tokens flowing in is added to the inflow amount-in-flight and subtracted from the outflow amount-in-flight (however, subtraction saturates, i.e. the amount-in-flight will become zero when subtraction makes it negative), and vice versa. An amount-in-flight linearly decays over time. If an addition to an amount-in-flight lets it becomes greater than the configured limit, the contract would revert.

To implement the decay of the amount-in-flight, the contract stores the amount-in-flight and the last updated timestamp for that amount:

```
fn _get_flowable_amount(
    self: @ComponentState<TContractState>, limit: @RateLimit,
) -> FlowableAmount {
    let RateLimit { amount_in_flight, last_updated, limit, window } = limit;

    // Prevent division by zero.
    let window = max(*window, 1);
    let duration = get_block_timestamp() - *last_updated;

    // Presume linear decay.
    let amount_in_flight: u256 = (*amount_in_flight)
        .into()
        .saturating_sub((*limit).into() * duration.into() / window.into());

    FlowableAmount {
        // Although the amount in flight should never be above the limit, we
    double-check
        // that with saturating subtraction.
        amount_in_flight, flowable_amount:
    (*limit).into().saturating_sub(amount_in_flight),
```

```
    }
}

// (...)

fn _check_and_update_rate_limit(
    ref self: ComponentState<TContractState>,
    eid: u32,
    amount: u256,
    direction: RateLimitDirection,
) {
    // (...)

    if is_direction_enabled {
        // (...)

        let rate_limit = direction_entry.read();

        let FlowableAmount {
            amount_in_flight, flowable_amount,
        } = Self::_get_flowable_amount(@self, @rate_limit);

        // (...)

        direction_entry
            .write(
                RateLimit {
                    amount_in_flight: (amount_in_flight +
    amount).try_into().unwrap(),
                    last_updated: get_block_timestamp(),
                    ..rate_limit,
                },
            );
    }

    // (...)
}
```

However, the `_check_and_update_rate_limit` also updates the amount-in-flight of the opposite direction (if enabled), and this function updates the `last_updated` field, not applying the decay of the amount-in-flight:

```
fn _check_and_update_rate_limit(
    ref self: ComponentState<TContractState>,
    eid: u32,
    amount: u256,
```

```
    direction: RateLimitDirection,
) {
    // (...)

    if is_opposite_direction_enabled {
        // (...)

        let rate_limit = opposite_direction_entry.read();
        let amount_in_flight: u256 = rate_limit
            .amount_in_flight
            .into()
            .saturating_sub(amount);

        opposite_direction_entry
            .write(
                RateLimit {
                    amount_in_flight: amount_in_flight.try_into().unwrap(),
                    last_updated: get_block_timestamp(),
                    ..rate_limit,
                },
            );
    }
}
```

### Impact

Rate-limiting would be more restrictive than expected since decay of amount-in-flight is not properly applied.

### Recommendations

Consider applying the decay of amount-in-flight when the `last_updated` field is updated.

### Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `1e3fef41` ↗.

### 3.3. The `value` provided in the received message is not checked

| | | | |
|---|---|---|---|
| **Target** | layerzero/src/oapps/counter/counter.cairo | | |
| **Category** | Coding Mistakes | **Severity** | Medium |
| **Likelihood** | Medium | **Impact** | Medium |

#### Description

```
impl OAppHooks of OAppCoreComponent::OAppHooks<ContractState> {
    fn _lz_receive(
        ref self: OAppCoreComponent::ComponentState<ContractState>,
        origin: Origin,
        guid: Bytes32,
        message: ByteArray,
        executor: ContractAddress,
        extra_data: ByteArray,
        value: u256,
    ) {
        [...]
        if msg_type == VANILLA_TYPE {
            contract.count.write(contract.count.read() + 1);

            // For VANILLA_TYPE, we don't have value encoded in the message
    anymore
            // The value check was part of the Solidity implementation but not
    needed for tests

            contract._increment_inbound(origin.src_eid);
```

`OmniCounter` contract is a sample contract to explain how developers can build on Starknet using LayerZero. The contract includes example implementations of basic LayerZero functions such as `lz_compose` and `_lz_receive`.

As noted in the comments within the `_lz_receive` function, the check for the value encoded in the message is skipped in the Cairo implementation.

#### Impact

In the Solidity implementation, there is a check that compares the value encoded in the message with `msg.value` to ensure that the executor has provided the correct amount of `msg.value`.

It is still important to ensure that the value parameter provided in the message is matched to what provided by the endpoint (through the function parameter of the `_lz_receive` function) the in the Cairo implementation of LayerZero. Skipping this check on the sample contract, `OmniCounter`, may confuse developers on the way to correctly implement LayerZero-based contracts.

### Recommendations

Add a check that compares the value of the `value` argument with the value encoded in the message's `value` field.

### Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `81665b6c` ↗.

### 3.4. The `quorum_change_admin` function does not check the calldata selector

| | | | |
|---|---|---|---|
| **Target** | layerzero/src/workers/dvn/structs.cairo | | |
| **Category** | Coding Mistakes | **Severity** | High |
| **Likelihood** | Low | **Impact** | Low |

#### Description

The `quorum_change_admin` function can grant the DVN admin role to a new address using signatures signed by a multi-sig. This function takes `vid`, `call_data`, `expiration`, and `signatures` as arguments and performs the following validations:

- Whether `expiration` exceeds the current block's timestamp
- Whether the target of `call_data` is the same as the current contract address
- Whether `vid` is the same as the `vid` defined in the contract
- Whether the signatures are valid for the hash computed from the set of `vid`, `call_data`, and `expiration`
- Whether the computed hash has already been used

However, the function does not check whether the selector of the provided `call_data` points to the `quorum_change_admin` function.

#### Impact

If the multi-sig signs calldata for invoking another function with an address as an argument, it is possible to call the `quorum_change_admin` function using those signatures.

#### Recommendations

Modify the `quorum_change_admin` function to include a check that `call_data.selector` equals the selector for `quorum_change_admin`.

#### Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `907eb6f0` ↗.

### 3.5. The `value` function may panic when processing messages without `value` field

| Target | layerzero/src/oapps/counter/msg_codec.cairo | | |
|---|---|---|---|
| **Category** | Coding Mistakes | **Severity** | Medium |
| **Likelihood** | Low | **Impact** | Low |

#### Description

```
pub fn value(message: @ByteArray) -> u256 {
    let (_, value_data) = message.read_u256(VALUE_OFFSET);
    value_data
}
```

The `value` function returns the `value` field in `u256` form from the message. Note that this field is optional and can be omitted:

```
pub fn encode(msg_type: u8, src_eid: u32) -> ByteArray {
    let mut message: ByteArray = Default::default();
    message.append_u8(msg_type);
    message.append_u32(src_eid);
    message
}

pub fn encode_with_value(msg_type: u8, src_eid: u32, value: u256) ->
    ByteArray {
    let mut message: ByteArray = Default::default();
    message.append_u8(msg_type);
    message.append_u32(src_eid);
    message.append_u256(value);
    message
}
```

However, if the `value` field is missing in that message, the `value` function will panic due to an out-of-bounds access on the byte array:

```
/// Read a u256 from ByteArray
#[inline(always)]
fn read_u256(self: @ByteArray, offset: usize) -> (usize, u256) {
    read_uint::<u256>(self, offset, 32)
```

```
}

// (...)

fn read_uint<T, +Add<T>, +Mul<T>, +Zero<T>, +TryInto<felt252, T>, +Drop<T>,
    +Into<u8, T>>(
    self: @ByteArray, offset: usize, size: usize,
) -> (usize, T) {
    assert(offset + size <= self.len(), 'out of bound');
    // (...)
}
```

### Impact

The function is implemented in the `msg_codec` contract, which is imported by the `OmniCounter` contract used as an example to explain how developers can build on Starknet using LayerZero. If the current implementation of the `value` function is used as-is, a transaction may revert when receiving a message that does not contain a `value` field.

### Recommendations

Modify the function so that it returns zero when the length of the message is less than or equal to `VALUE_OFFSET`.

### Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `81665b6c` ↗.

### 3.6. Function `assign_job` of DVN is pausable

| Target | dvn.cairo | | |
|---|---|---|---|
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

#### Description

The DVN contract has the `assign_job` and `quote` functions, which can be paused by the default admin:

```
fn assign_job(ref self: ContractState, params: QuoteParams) -> u256 {
    self.worker_base._assert_not_paused();
    // (...)
    self.quote(params)
}

// (...)

fn quote(self: @ContractState, params: QuoteParams) -> u256 {
    self.worker_base._assert_not_paused();
    // (...)
}
```

However, it should be noted that the EVM implementation of the DVN contract does not allow these functions to be paused:

```
function assignJob(
    AssignJobParam calldata _param,
    bytes calldata _options
) external payable onlyRole(MESSAGE_LIB_ROLE) onlyAcl(_param.sender)
    returns (uint256 totalFee) {
    // (...)
}

// (...)

function getFee(
    uint32 _dstEid,
    uint64 _confirmations,
    address _sender,
```

```
    bytes calldata _options
) external view onlyAcl(_sender) returns (uint256 fee) {
    // (...)
}
```

### Impact

This finding simply documents the difference of the Cairo implementation of LayerZero protocol compared to the EVM implementation. When the same protocol is implemented across multiple ecosystems, differences in logic can cause confusion for developers and users and negatively impact maintainability.

### Recommendations

Consider not checking whether the contract is paused on the `assign_job` function of the DVN contract.

### Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `9f3e6744` ↗.

### 3.7. The `_check_and_update_rate_limit` function reverts when rate limit is exceeded which is not as the comment explains

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

#### Description

The comment of the `_check_and_update_rate_limit` function explains that the function would return `true` if the rate limit is exceeded and `false` otherwise. However, it does not return any result, but reverts when the rate limit is exceeded:

```
/// Checks and updates the rate limit for the given endpoint ID and amount.
///
/// For each direction we need to check and update the rate limit, and also
    update the
/// opposite direction.
/// but in case the direction is disabled, we don't need to update the rate
    limit.
///
/// # Arguments
/// * `eid` - The endpoint ID.
/// * `amount` - The amount to check and update.
/// * `direction` - The direction of the rate limit.
///
/// # Returns
/// * `true` if the rate limit is exceeded, `false` otherwise.
fn _check_and_update_rate_limit(
    ref self: ComponentState<TContractState>,
    eid: u32,
    amount: u256,
    direction: RateLimitDirection,
) {
    // (...)

    if is_direction_enabled {
        // (...)
        assert_with_byte_array(amount <= flowable_amount,
    err_rate_limit_exceeded());
        // (...)
    }
```

```
    // (...)
}
```

### Impact

This does not affect the business logic of the project, but may confuse developers on the exact behavior of this function.

### Recommendations

Consider correcting the comment to explain the behavior of the function correctly.

### Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `1e3fef41` ↗.

# 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. The `_clear_payload` function does not check the origin's nonce

The `_clear_payload` function is called during message execution to clear verified payloads. Before updating the lazy inbound nonce to the provided nonce, the function contains logic to check whether all nonces between the current nonce and the provided nonce (inclusive) have payload hashes. This logic is implemented as follows in the Starknet and EVM versions:

```
// Starknet implementation
fn _clear_payload(
    ref self: ComponentState<TContractState>,
    receiver: ContractAddress,
    origin: @Origin,
    payload: @ByteArray,
) {
    [...]
    // If this nonce is beyond the current checkpoint, validate ordered
    execution
    if *origin.nonce > current_nonce {
        // Ensure that no nonce between current_nonce and origin.nonce is
    missing a payload
        // hash.  Because if it is, it means we have an unverified nonce in
    between
        // and we can't execute the packet until everything before it is
    committed
        for nonce in current_nonce + 1..*origin.nonce {
            assert_with_byte_array(
                self._has_payload_hash(receiver, *origin.src_eid,
    *origin.sender, nonce),
                err_invalid_nonce(),
            );
        }
        // Update the lazy nonce to the current execution point
        lazy_nonce_entry.write(*origin.nonce);
    }
    [...]
}
```

```
// EVM implementation
if (_nonce > currentNonce) {
```

```
    unchecked {
        // try to lazily update the inboundNonce till the _nonce
        for (uint64 i = currentNonce + 1; i <= _nonce; ++i) {
            if (!_hasPayloadHash(_receiver, _srcEid, _sender, i))
    revert Errors.LZ_InvalidNonce(i);
        }
        lazyInboundNonce[_receiver][_srcEid][_sender] = _nonce;
    }
}
```

However, in the Starknet version of the code, since Cairo's `..` operator returns an exclusive range, the check for `origin.nonce` is not performed. This does not pose a problem because the check for that nonce is performed later in the function. Nevertheless, this issue is worth noting as it introduces a behavioral difference between the two implementations, and it was mentioned during the audit as it represents a typical bug pattern that can arise when the same protocol is translated into different smart contract languages.

### 4.2. The `_reset_rate_limits` function is unused

`RateLimiterComponent` allows the rate limit to be reset (i.e. set amount-in-flight to zero) by invoking the `_reset_rate_limits` internal function. However, `OFTMintBurnAdapter`, the only contract using `RateLimiterComponent` in the project, does not use this function. It is not possible to reset the rate limit of the contract in other ways.

If it is needed to reset the rate limit of `OFTMintBurnAdapter` contract, consider adding the function that invokes the `_reset_rate_limits` function.

This issue has been acknowledged by LayerZeroLabs, and a fix was implemented in commit 1e3fef41 ↗

### 4.3. Scope change details

A scope change occurred on **October 7th, 2025**. The change was from 3d76b69a ↗ to e9e31007 ↗.

An additional scope change occurred on **November 13rd, 2025**. The change was from e9e31007 ↗ to f66a9fb4 ↗. Zellic conducted a diff audit between the previous scope and the new scope.

The following components were added to the codebase:

- `avalon_oft/src/*`
- `starkgate_solvbtc/src/*`

An additional scope change occurred on **November 24th, 2025**. The change was from f66a9fb4 to 5353a346. Zellic conducted a diff audit between the previous scope and the new scope.

The following components were added to the codebase:

- `oft_mint_burn/src/*`

An additional scope change occurred on **December 1st, 2025**. The change was from 5353a346 to c5a5532d. Zellic conducted a diff audit between the previous scope and the new scope.

The following components were added to the codebase:

- `oft/src/*`
- `oft_adapter/src/*`

An additional scope change occurred on **December 17th, 2025**. The change was from c5a5532d to f6b1b803. Zellic conducted a diff audit between the previous scope and the new scope.

# 5. System Design

As part of the audit process, Zellic performed a technical comparison between the Starknet implementation of Endpoint V2, translated from the original Solidity code for EVM-compatible blockchains, and its EVM counterpart. The goal was to identify vulnerabilities and bugs introduced during the translation, as well as incorrect implementations that deviate from the protocol's specification.

This provides an overview of the differences between the Starknet and EVM implementations of LayerZero Endpoint V2.

## 5.1. Key differences between the Starknet and EVM implementations

### Catching reverts in external contract calls

The LayerZero Endpoint V2 implementation often invokes functions on other contracts to query status or transfer assets. To keep request processing and the execution flow robust, many places handle reverts from those calls.

Below is one example from the EVM implementation:

```
function initializable(Origin memory _origin, address _receiver)
    public view returns (bool) {
    try endpoint.initializable(_origin, _receiver)
    returns (bool _initializable) {
        return _initializable;
    } catch {
        return false;
    }
}

// endpoint.initializable
function initializable(Origin calldata _origin, address _receiver)
    external view returns (bool) {
    return _initializable(_origin, _receiver,
    lazyInboundNonce[_receiver][_origin.srcEid][_origin.sender]);
}

// endpoint._initializable
function _initializable(
    Origin calldata _origin,
    address _receiver,
    uint64 _lazyInboundNonce
) internal view returns (bool) {
    return
        _lazyInboundNonce > 0 || // allowInitializePath already checked
        ILayerZeroReceiver(_receiver).allowInitializePath(_origin);
}
```

The `initializable` function executes `endpoint.initializable` inside a try–catch block. This way, even if `_receiver.allowInitializePath` reverts, execution falls back to the catch branch and safely returns false.

The Starknet implementation also supports this kind of error handling using Cairo's Safe Dispatcher ↗. The same logical path is implemented as follows:

```
#[feature("safe_dispatcher")]
fn _safe_endpoint_initializable(
    self: @ContractState, origin: Origin, receiver: ContractAddress,
) -> bool {
    let endpoint_dispatcher = IEndpointSafeDispatcher {
        contract_address: self.endpoint.read(),
    };
    endpoint_dispatcher.initializable(origin, receiver).unwrap_or(false)
}
```

However, on Starknet, even with a Safe Dispatcher, some cases still cause an immediate revert without returning control to the caller, such as:

- Failure in a Cairo Zero contract
- Library call with a non-existent class hash
- Contract call with a non-existent contract address
- Failure in the deploy syscall
- Using the deploy syscall with a non-existent class hash
- Using the replace_class syscall with a non-existent class hash

In the example above, if `receiver` is a non-existent contract, the Safe Dispatcher call will not return false; instead, the transaction itself will revert. Because of this, the Starknet and EVM implementations can exhibit minor differences in execution flow. That said, no security or functional issues have been found to arise from these differences.

### Transfer of the Protocol Fee

In the EVM implementation, users can pay the protocol fee with either the native currency or the LayerZero token.

- Using the native currency : The user includes the required native fee in `msg.value` when invoking the OApp's function. The OApp verifies the amount, then calls the Endpoint and forwards `msg.value` unchanged.
- Using the LayerZero token: The user first approves the OApp for the required fee. The OApp then calls `transferFrom` to move tokens directly from the user to the Endpoint.

On Starknet, all payments are tokenized. As in the EVM token case, the user first approves the OApp for the required fee. The difference is that, instead of transferring directly from the user to the Endpoint, the OApp first pulls the tokens from the user into the OApp contract using the

allowance. The OApp then approves the Endpoint to take those freshly received tokens. The Endpoint subsequently pulls the approved tokens from the OApp. Therefore, unlike on EVM, the user's tokens pass through the OApp before reaching the Endpoint.

# 6. Assessment Results

During our assessment on the scoped LayerZero V2 Starknet targets, we discovered seven findings. One critical issue was found. Two were of medium impact, two were of low impact, and the remaining findings were informational in nature.

This project is the Starknet implementation of LayerZero V2, intended to maintain the core architecture and functionality of the EVM implementation. Overall, the project has successfully translated the existing Solidity implementation into Cairo, and no major functional deviations from the EVM version have been identified. However, as mentioned in Finding 3.1. ↗, the current implementation contains several entry points that make use of user-provided `ByteArray` values without validation, which could lead to serious security issues. Therefore, we believe it is essential that this finding be thoroughly addressed and that the corresponding fix be carefully reviewed before the project is deployed to mainnet.

## 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.