Zellic

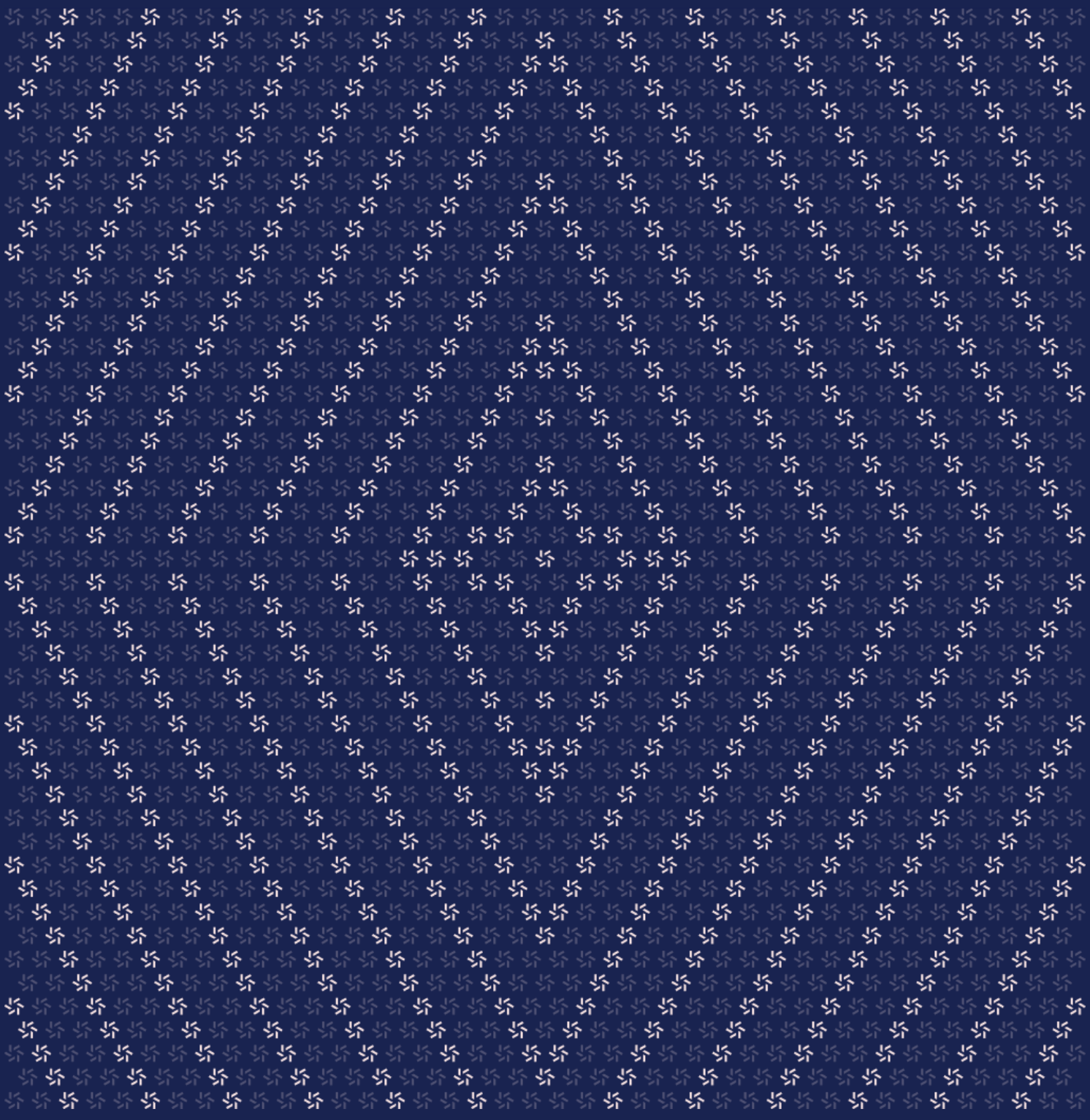
Prepared for
Matthias Hallgren
Pragma

Prepared by
Jinseo Kim
Jisub Kim
Ulrich Myhre
Zellic

July 9, 2024

Hyperlane Starknet

Smart Contract 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 Pragma from June 20th to July 9th, 2024. During this engagement, Zellic reviewed Hyperlane 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:

- Could message transmission between multiple chains be carried out correctly?
- Could message transmission be blocked?
- Could the hash result of Keccak be the same with the result in Solidity and off chain?
- Are there any differences between the current implementation and the Solidity implementation?

1.3. Non-goals and Limitations

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

- Hyperlane protocol implementation on other chains
- 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.

During this assessment, the lack of comprehensive test suites and the limited time-frame prevented us from fully assessing the scoped codebase and ensuring that the scoped codebase works as intended. We discuss this in Finding 3.21. ↗.

Based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes. We also recommend adopting a security-focused development workflow, including (but not limited to) augmenting the repository with comprehensive end-to-end tests that achieve 100% branch coverage using any common, maintainable testing framework, thoroughly documenting all function requirements, and training developers to have a security mindset while writing code.

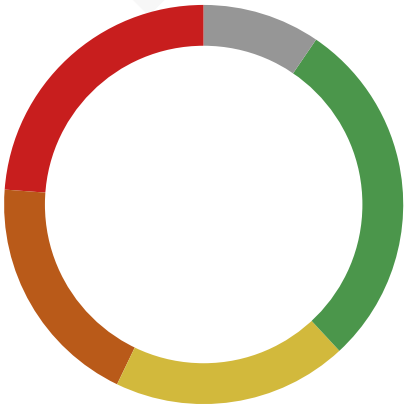
1.4. Results

During our assessment on the scoped Hyperlane Starknet contracts, we discovered 21 findings. Five critical issues were found. Four were of high impact, four were of medium impact, six were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	5
High	4
Medium	4
Low	6
Informational	2



2. Introduction

2.1. About Hyperlane Starknet

Pragma contributed the following description of Hyperlane Starknet:

> Hyperlane is the first universal and permissionless interoperability layer designed for the modular blockchain ecosystem. The purpose of this project is to define an implementation of the Hyperlane protocol for chains using the Starknet stack.

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.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

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:

Hyperlane Starknet Contracts

Type	Cairo
Platform	Starknet

Target	hyperlane_starknet
Repository	https://github.com/astraly-labs/hyperlane_starknet ↗
Version	`f61ee04079a7446ab4fcdaf789635d5ac4282dc8`

Programs	`lib.cairo`
	`interfaces.cairo`
	`contracts/isms/pausable_ism.cairo`
	`contracts/isms/trusted_relayer_ism.cairo`
	`contracts/isms/noop_ism.cairo`
	`contracts/isms/routing/domain_routing_ism.cairo`
	`contracts/isms/routing/default_fallback_routing_ism.cairo`
	`contracts/isms/multisig/merkleroot_multisig_ism.cairo`
	`contracts/isms/multisig/messageid_multisig_ism.cairo`
	`contracts/isms/multisig/validator_announce.cairo`
	`contracts/isms/aggregation/aggregation.cairo`
	`contracts/mailbox.cairo`
	`contracts/libs/message.cairo`
	`contracts/libs/multisig/message_id_ism_metadata.cairo`
	`contracts/libs/multisig/merkleroot_ism_metadata.cairo`
	`contracts/libs/checkpoint_lib.cairo`
	`contracts/libs/aggregation_ism_metadata.cairo`
	`contracts/hooks/merkle_tree_hook.cairo`
	`contracts/hooks/libs/standard_hook_metadata.cairo`
	`contracts/hooks/protocol_fee.cairo`
	`contracts/client/mailboxclient_component.cairo`
	`contracts/client/mailboxclient.cairo`
	`utils/keccak256.cairo`
	`utils/store_arrays.cairo`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.7 person-weeks. The assessment was conducted by three consultants over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Jinseo Kim
Engineer
jinseo@zellic.io ↗

Jisub Kim
Engineer
jisub@zellic.io ↗

Ulrich Myhre
Engineer
unblvr@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 20, 2024	Start of primary review period
June 21, 2024	Kick-off call
July 9, 2024	End of primary review period

3. Detailed Findings

3.1. Aggregation ISM cannot skip ISMs

Target	aggregation_ism_metadata.cairo		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Aggregation ISM in the Hyperlane protocol is an ISM that returns true when m-of-n ISMs return true, where the threshold and the list of ISMs are predetermined. It is implemented in aggregation.cairo:

```
fn verify(self: @ContractState, _metadata: Bytes, _message: Message,) ->
    bool {
    let (isms, mut threshold) = self.modules_and_threshold(_message.clone());

    assert(threshold != 0, Errors::THRESHOLD_NOT_SET);
    let modules = self.build_modules_span();
    let mut cur_idx: u8 = 0;
    loop {
        if (threshold == 0) {
            break ();
        }
        if (cur_idx.into() == isms.len()) {
            break ();
        }
        if (!AggregationIsmMetadata::has_metadata(_metadata.clone(), cur_idx))
    {
            cur_idx += 1;
            continue;
        }
        let ism = IInterchainSecurityModuleDispatcher {
            contract_address: *modules.at(cur_idx.into())
        };

        let metadata = AggregationIsmMetadata::metadata_at(_metadata.clone(),
    cur_idx);
        assert(ism.verify(metadata, _message.clone()),
    Errors::VERIFICATION_FAILED);
        threshold -= 1;
        cur_idx += 1;
    };
    assert(threshold == 0, Errors::THRESHOLD_NOT_REACHED);
```

```
    true
}
```

Aggregation ISM is expected to skip ISMs that will return false or revert. To implement this behavior, it is checked if the given metadata includes the metadata for the ISM, and ISMs that do not have their metadatas are skipped.

However, the function `AggregationIsmMetadata::has_metadata` actually does not check the existence of the metadata:

```
fn has_metadata(_metadata: Bytes, _index: u8) -> bool {
    match metadata_range(_metadata, _index) {
        Result::Ok((_, _)) => true,
        Result::Err(_) => false
    }
}

// ...

fn metadata_range(_metadata: Bytes, _index: u8) -> Result<(u32, u32), u8> {
    let start = _index.into() * RANGE_SIZE * 2;
    let mid = start + RANGE_SIZE;
    let (_, mid_metadata) = _metadata.read_u32(mid.into());
    let (_, start_metadata) = _metadata.read_u32(start.into());
    Result::Ok((start_metadata, mid_metadata))
}
```

The function `has_metadata` returns false if the function `metadata_range` returns an error. However, the function `metadata_range` never returns an error; it should always return the normal result or revert (if `mid` or `start` overflows the size of `_metadata`). Either way, the ISM verification will fail at this point, although there are more ISMs that can be checked.

Impact

M-of-n Aggregation ISM will revert if one of the first m ISMs does not return true.

Recommendations

Consider modifying the function `has_metadata` in order to allow a relayer to specify ISMs to be skipped.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit 939b0435 ↗.

3.2. Incorrect splitting of a number in Keccak implementation

Target	keccak256.cairo		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

This project implements the Keccak function, which receives an array of input chunks that would be concatenated and hashed. The following is the function `concatenate_input`, which concatenates the given input array:

```
pub const FELT252_MASK:
    u256 = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;

// ...

#[derive(Copy, Drop, Serde, starknet::Store, Debug, PartialEq)]
pub struct ByteData {
    pub value: u256,
    pub size: usize
}

// ...

fn concatenate_input(bytes: Span<ByteData>) -> ByteArray {
    let mut output_string: ByteArray = Default::default();
    let mut cur_idx = 0;

    loop {
        if (cur_idx == bytes.len()) {
            break ();
        }
        let byte = *bytes.at(cur_idx);
        if (byte.size == 32) {
            // in order to store a 32-bytes entry in a ByteArray, we need to
    first append the upper 1-byte part , then the lower 31-bytes part
            let up_byte = (byte.value / FELT252_MASK).try_into().unwrap();
            output_string.append_word(up_byte, 1);
            let down_byte = (byte.value & FELT252_MASK).try_into().unwrap();
            output_string.append_word(down_byte, 31);
        } else {
```

```
                output_string.append_word(byte.value.try_into().unwrap(),
    byte.size);
            }
            cur_idx += 1;
    };
    output_string
}
```

It can be observed that the 32-byte chunk is split into two values and appended separately. This is because the function `append_word` receives the values as felt types, which can only safely store 31-byte data.

The most significant byte of the 32-byte chunk is obtained by dividing the chunk by `FELT256_MASK`. However, this method may result in an incorrect result in specific cases. For example, the most significant byte of `0x01FFFF(...total 32 bytes...)FFFF` is `0x01`, but the result using the above method would be `0x02`.

Impact

This bug may cause incorrect Keccak hash derivation, which may lead to the failure of message dispatching and processing.

Recommendations

Consider changing the logic obtaining the most significant byte of the 32-byte chunk.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit `d9152fc4` ↗.

3.3. Improper optimization in Keccak implementation

Target	keccak256.cairo		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

For the inputs with zero, the Keccak implementation returns the precalculated hash in order to optimize in common cases:

```
const EMPTY_KECCAK:
    u256 = 0x70A4855D04D8FA7B3B2782CA53B600E5C003C7DCB27D7E923C23F7860146D2C5;

// ...

#[derive(Copy, Drop, Serde, starknet::Store, Debug, PartialEq)]
pub struct ByteData {
    pub value: u256,
    pub size: usize
}

// ...

fn keccak_cairo_words64(words: Words64, last_word_bytes: usize) -> u256 {
    if words.is_empty() {
        return EMPTY_KECCAK;
    }

    // ...
}

// ...

pub fn compute_keccak(bytes: Span<ByteData>) -> u256 {
    if (bytes.is_empty()) {
        return keccak_cairo_words64(array![].span(), 0);
    }
    if (*bytes.at(0).value == 0) {
        return keccak_cairo_words64(array![].span(), 0);
    }
    // ...
```

```
}
```

If the `value` field of the first element of the input array is zero, the function returns the Keccak hash of the empty data. However, it should be noted that this optimization is incorrect, because of these two reasons:

1. One, `ByteData` represents the `size`-byte data with `value`, and `value` can be zero in the case the data is not zero-byte. For example, the data 0x0000 is represented with `array![ ByteData { value: 0_u256, size: 2 } ]`.
2. Two, `ByteData` may contain other elements that will be concatenated and hashed together, such as `array![ ByteData { value: 0_u256, size: 0 }, ByteData { value: 1_u256, size: 1 } ]`.

Impact

This bug may cause incorrect Keccak hash derivation, which may lead to the failure of message dispatching and processing.

Recommendations

Consider adding the conditions for the optimization to be triggered.

Remediation

This issue has been acknowledged by Pragma, and fixes were implemented in the following commits:

- f396498a ↗
- 81d84cae ↗

3.4. Dynamic variable size for hash parameters

Target	checkpoint_lib.cairo, message.cairo, validator_announce.cairo		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `digest` and `domain_hash` functions in checkpoint_lib.cairo, the `format_message` function in message.cairo, and the `domain_hash` function in validator_announce.cairo try to follow the behavior of `abi.encodePacked` by appending the `ByteData` struct with the size calculated with the `u{64,256}_word_size`:

```
fn format_message(_message: Message) -> (u256, Message) {
    let sender: felt252 = _message.sender.into();
    let recipient: felt252 = _message.recipient.into();

    let mut input: Array<ByteData> = array![
        ByteData {
            value: _message.version.into(), size:
    u64_word_size(_message.version.into()).into()
        },
        ByteData {
            value: _message.nonce.into(), size:
    u64_word_size(_message.nonce.into()).into()
        },
        ByteData {
            value: _message.origin.into(), size:
    u64_word_size(_message.origin.into()).into()
        },
        ByteData { value: sender.into(), size: ADDRESS_SIZE },
        ByteData {
            value: _message.destination.into(),
            size: u64_word_size(_message.destination.into()).into()
        },
        ByteData { value: recipient.into(), size: ADDRESS_SIZE },
        ByteData {
            value: _message.body.size().into(),
            size: u64_word_size(_message.body.size().into()).into()
        },
    ];
```

```
    // ...
}
```

However, static type variables such as `uint32`, `uint256`, and `bytes32` take up the space of the result of the `abi.encodePacked` function as much as its static size, like `abi.encodePacked(bytes4(uint32(1)), bytes4(uint32(2)))` returns `0x0000000100000002`, not `0x0102`.

Since `u{64,256}_word_size` returns the minimum byte length to store the given value (e.g., `u64_word_size(1_u64) == 1_u8`), using these functions to encode the static type variables and implement the `abi.encodePacked` function may result in incorrect behavior for inputs that start with the zero byte.

In the case where the 128-bit chunk of the body starts with the zero bytes, for instance, if the chunk of the body looks like `0x000...(total 16 bytes)...001`, the word size function will return one and it will be interpreted as `0x01` when it is hashed.

Impact

This bug may result in generating messages incompatible to the ABI of the Hyperlane protocol, which may lead to the failure of message dispatching and processing.

Recommendations

Consider using the static size instead of dynamically calculated size when generating messages.

Remediation

This issue has been acknowledged by Pragma, and fixes were implemented in the following commits:

- 44423ee7 ↗
- 90fb0167 ↗
- 83b5cc8a ↗

3.5. Message incorrectly includes the size of body

Target	message.cairo		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `format_message` function in message.cairo appends the size of the message body before the message body:

```
fn format_message(_message: Message) -> (u256, Message) {
    let sender: felt252 = _message.sender.into();
    let recipient: felt252 = _message.recipient.into();

    let mut input: Array<ByteData> = array![
        // ...
        ByteData {
            value: _message.body.size().into(),
            size: u64_word_size(_message.body.size().into()).into()
        },
    ];

    // ...
}
```

However, this does not match with the behavior of the Hyperlane protocol, which does not append the size of the body:

```
function formatMessage(
    uint8 _version,
    uint32 _nonce,
    uint32 _originDomain,
    bytes32 _sender,
    uint32 _destinationDomain,
    bytes32 _recipient,
    bytes calldata _messageBody
) internal pure returns (bytes memory) {
    return
        abi.encodePacked(
            _version,
```

```
            _nonce,
            _originDomain,
            _sender,
            _destinationDomain,
            _recipient,
            _messageBody
        );
}
```

Impact

This can lead to incompatibility issues with the Hyperlane protocol, potentially causing message interpretation errors across different chains and implementations.

Recommendations

Consider removing the size when formatting a message into bytes.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit 055424fc↗.

3.6. Multisig ISM allows duplicated signatures

Target	merkleroot_multisig_ism.cairo, messageid_multisig_ism.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

Multisig ISM in the Hyperlane protocol is an Interchain Security module (ISM) that returns true when the signatures of m-of-n validators are provided, where the threshold and the list of validators are predetermined. It is implemented in merkleroot_multisig_ism.cairo and messageid_multisig_ism.cairo:

```
fn verify(self: @ContractState, _metadata: Bytes, _message: Message,) ->
    bool {
    assert(_metadata.clone().size() > 0, Errors::EMPTY_METADATA);
    let digest = self.digest(_metadata.clone(), _message.clone());
    let (validators, threshold) = self.validators_and_threshold(_message);
    assert(threshold > 0, Errors::NO_MULTISIG_THRESHOLD_FOR_MESSAGE);
    let mut i = 0;
    // for each couple (sig_s, sig_r) extracted from the metadata
    loop {
        if (i == threshold) {
            break ();
        }
        let signature = self.get_signature_at(_metadata.clone(), i);
        // we loop on the validators list public key in order to find a match
        let mut cur_idx = 0;
        let is_signer_in_list = loop {
            if (cur_idx == validators.len()) {
                break false;
            }
            let signer = *validators.at(cur_idx);
            if bool_is_eth_signature_valid(digest, signature, signer) {
                // we found a match
                break true;
            }
            cur_idx += 1;
        };
        assert(is_signer_in_list, Errors::NO_MATCH_FOR_SIGNATURE);
        i += 1;
    };
```

```
    true
}
```

However, this code allows a signature from one validator to be accepted more than once.

Impact

A malicious relayer who can obtain a signature of one validator can make the Multisig ISM return true by submitting the signature from one validator multiple times.

Recommendations

Consider ensuring that given signatures are not duplicated.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit ac648a31 ↗.

3.7. The protocol fee hook will always be reverted

Target	protocol_fee.cairo		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The protocol fee hook is the hook that collects the protocol fee from the sender of the message. Specifically, the function `post_dispatch` transfers the protocol fee from the caller address to itself:

```
fn _post_dispatch(ref self: ContractState, _metadata: Bytes, _message:
    Message) {
    let token_dispatcher = IERC20Dispatcher { contract_address:
    self.fee_token.read() };
    let caller_address = get_caller_address();
    let contract_address = get_contract_address();
    let user_balance = token_dispatcher.balance_of(caller_address);
    assert(user_balance != 0, Errors::INSUFFICIENT_BALANCE);
    let protocol_fee = self.protocol_fee.read();
    assert(
        token_dispatcher.allowance(caller_address, contract_address)
    >= protocol_fee,
        Errors::INSUFFICIENT_ALLOWANCE
    );
    token_dispatcher.transfer_from(caller_address, contract_address,
    protocol_fee);
}
```

However, it should be noted that the caller of the function `post_dispatch` is the Mailbox contract, which is not intended to pay the fee, causing the fee collection to fail.

Impact

The protocol fee hook will be reverted always, which may cause a failure in the dispatching of a message.

Recommendations

Consider changing the logic of the fee collection, considering the exact flow of the fee.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit e6388f31 ↗.

3.8. The `contractAddress` type cannot use the 32-byte addressing mechanism

Target	mailbox.cairo, mailboxclient_component.cairo, message.cairo, interfaces.cairo, aggregation.cairo		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The Hyperlane protocol defines the sender and receiver address as a value of the `bytes32` type. This is to handle the messages from/to the chain that uses the 32-byte addressing mechanism.

However, we found that the addresses are defined as the type `starknet::contractAddress`, which is equivalent to the type `felt252`. Therefore, the implementation would not be able to handle an address that does not fit in `felt252`. For example, this affects the messages from/to the Neutron chain, which is the Cosmos-based chain with the digital key scheme `secp256r1`. (Do not confuse this with the cryptographic algorithm, which this digital key scheme is based on.)

Impact

This limitation prevents the system from supporting recipients using 32-byte addresses, potentially excluding some portion of users and limiting cross-chain compatibility and interoperability.

Recommendations

Change the type of sender and receiver `starknet::contractAddress` to `u256`.

Remediation

This issue has been acknowledged by Pragma, and fixes were implemented in the following commits:

- 8caaeaed ↗

3.9. Input arguments in the `Bytes` type may be invalid

Target	mailbox.cairo		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The `Bytes` type is defined in the third-party package `alexandria_bytes`:

```
/// Bytes is a cairo implementation of solidity Bytes in Big-endian.
/// It is a dynamic array of u128, where each element contains 16 bytes.
/// To save cost, the last element MUST be filled fully.
/// That means that every element should and MUST contain 16 bytes.
/// For example, if we have a Bytes with 33 bytes, we will have 3 elements.
/// Theoretically, the bytes look like this:
/// first element: [16 bytes]
/// second element: [16 bytes]
/// third element: [1 byte]
/// But in alexandria bytes, the last element should be padded with zero to make
/// it 16 bytes. So the alexandria bytes look like this:
/// first element: [16 bytes]
/// second element: [16 bytes]
/// third element: [1 byte] + [15 bytes zero padding]

/// Bytes is a dynamic array of u128, where each element contains 16 bytes.
/// - size: the number of bytes in the Bytes
/// - data: the data of the Bytes
#[derive(Drop, Clone, PartialEq, Serde)]
pub struct Bytes {
    size: usize,
    data: Array<u128>
}
```

Note that it does not enforce the value of `size` to be consistent with the content of `data`. The `size` variable can contain a value that does not match with the content of the `data` variable if it is crafted by untrusted parties.

Since the behavior of the methods for the invalid `Bytes` value is undefined, it should be checked if it is given from the external sources like arguments. However, the functions `dispatch` and `process` do not validate the messages and metadata that are the `Bytes` type.

Impact

Invalid messages and metadata will be forwarded to the hook, ISM, recipient, and off-chain component. If these parties do not handle the invalid `Bytes` value consistently, a malicious user may exploit those inconsistent behaviors.

Recommendations

Consider sanitizing or validating the arguments in the `Bytes` type.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit `2b38132e` ↗.

3.10. Modules cannot be removed from routing ISM

Target	default_fallback_routing_ism.cairo, default_routing_ism.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

Routing ISM in the Hyperlane protocol is an ISM that redirects the result from the ISM designated for the origin chain:

```
fn remove(ref self: ContractState, _domain: u32) {
    self.ownable.assert_only_owner();
    self._remove(_domain);
}

// ...

fn _remove(ref self: ContractState, _domain: u32) {
    let domain_index = match self.find_domain_index(_domain) {
        Option::Some(index) => index,
        Option::None => {
            panic_with_felt252(Errors::DOMAIN_NOT_FOUND);
            0
        }
    };
    let next_domain = self.domains.read(_domain);
    self.domains.write(domain_index, next_domain);
}

// ...

fn route(self: @ContractState, _message: Message) -> ContractAddress {
    self.modules.read(_message.origin)
}

// ...

fn verify(self: @ContractState, _metadata: Bytes, _message: Message) -> bool {
    let ism_address = self.route(_message.clone());
    let ism_dispatcher = IInterchainSecurityModuleDispatcher {
        contract_address: ism_address
```

```
    };
    ism_dispatcher.verify(_metadata, _message)
}
```

There is the `remove` function, which should remove the specified routing configuration from itself. However, it only removes the configuration from the list, and the storage variable `modules`, which defines the ISM that is used when a message is verified, is unchanged.

Impact

This inconsistency can lead to confusion and potential security risks if other parts of the system rely on the module function to accurately reflect the current state of domain-module mappings.

Recommendations

Consider removing the module from the storage variable `modules` as well as when the configuration is removed.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit b3147211 ↗.

3.11. Routing ISM with the fallback configuration does not show fallback behavior

Target	default_fallback_routing_ism.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The behavior of Routing ISM should be well-defined for the case when the corresponding module does not exist in the Routing ISM. In this project, there are two implementations of Routing ISM: domain_routing_ism.cairo and default_fallback_routing_ism.cairo. The former should revert if the corresponding module does not exist, and the latter should fall back into the default ISM of the designated Mailbox.

```
fn module(self: @ContractState, _origin: u32) -> ContractAddress {
    let module = self.modules.read(_origin);
    if (module != contract_address_const::<0>()) {
        module
    } else {
        IMailboxDispatcher { contract_address: self.mailboxclient.mailbox() }
            .get_default_ism()
    }
}

// ...

fn route(self: @ContractState, _message: Message) -> ContractAddress {
    self.modules.read(_message.origin)
}

// ...

fn verify(self: @ContractState, _metadata: Bytes, _message: Message) -> bool {
    let ism_address = self.route(_message.clone());
    let ism_dispatcher = IInterchainSecurityModuleDispatcher {
        contract_address: ism_address
    };
    ism_dispatcher.verify(_metadata, _message)
}
```

The fallback behavior is implemented in the function `module`. However, the `route` function, which is

used by the `verify` function, does not use the `module` function but directly fetches the corresponding module. Because Starknet does not allow calls to the zero address, this will revert, implying the failure of message verification.

Impact

This can lead to unexpected failures in message processing, potentially disrupting cross-chain communication for new or unset origin domains.

Recommendations

Consider using the `module` function in the `route` function in order to improve the consistency of the code.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit `3db11f7e` ↗.

3.12. Owner address is not initialized

Target	merkle_tree_hook.cairo, validator_announce.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The contracts merkle_tree_hook.cairo and validator_announce.cairo embed the Ownable component. However, the Ownable component is not initialized in the constructor and so the owner address would not be configured.

We have found that the owner address is only checked for the features that do not affect the behavior of the contract (i.e., unused) in merkle_tree_hook.cairo. However, validator_announce.cairo only allows the owner to upgrade the contract; therefore this contract would not be updatable.

Impact

The contracts validator_announce.cairo and merkle_tree_hook.cairo cannot be upgraded.

Recommendations

Consider initializing the owner address in the constructor for both contracts.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit b3b2c967 ↗.

3.13. Incorrect size for fetching branches of the Merkle tree

Target	merkle_tree_hook.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `tree` function in the merkle_tree_hook.cairo returns the branches of the stored incremental Merkle tree. Specifically, it returns the array that is returned from the function `self._build_tree`, which semantically returns `array![ self.tree.read(0), self.tree.read(1), ..., self.tree.read(self.count.read() - 1) ]`.

However, it should be noted that the number of branches in this incremental Merkle tree is fixed to 32 in this contract and does not grow when the `count` variable, which represents the number of leaves, increases.

Impact

This could lead to unintended behavior and potential confusion for developers interacting with the contract.

Recommendations

Consider reading the correct number of elements when building the array of branches.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit 343b3810 ↗.

3.14. Message can be sent multiple times to an untrusted recipient

Target	mailbox.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The `process` function in the Mailbox contract is vulnerable to reentrancy through the `interchain_security_module` function, potentially allowing the same message to be sent multiple times.

Impact

We do not believe this poses a serious security risk because it is unlikely that the `interchain_security_module` function is implemented in a way triggering the reentrancy. We believe this finding can be only applied for the recipient that is actively exploiting this behavior; however, it does not pose a considerable security risk because a malicious recipient may just allow to receive any unchecked messages.

Nonetheless, we would recommend removing this behavior by recording the history of delivery before any external interactions (i.e., invoking the `interchain_security_module`, `verify`, and `handle` functions).

Recommendations

Consider recording the history of delivery before any external interactions.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit 6ec78842 ↗.

3.15. Announcing a new storage location overwrites the previous storage location

Target	validator_announce.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The validator creates signatures for the messages they endorse. These signatures should be relayed to relayers that are responsible to invoke the `process` function of the Mailbox. The validator-announcing contract helps this by allowing validators to announce their storage location (e.g., a S3 bucket) to relayers.

The validator can announce multiple storage locations in the Solidity implementation of the validator-announcing contract. In validator_announce.cairo, however, announcing a new storage location overwrites the existing storage location of the validator.

Impact

This divergence from the original implementation could lead to potential inconsistencies in validator announcements.

Recommendations

Consider implementing the validator-announcing contract in a way that announcing a new storage location does not overwrite the existing storage location.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit 2e400899 ↗.

3.16. Aggregation ISM misfunctions if more than 255 modules exist

Target	aggregation.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The following is the `verify` function of Aggregation ISM:

```
fn verify(self: @ContractState, _metadata: Bytes, _message: Message,) ->
    bool {
    let (isms, mut threshold) = self.modules_and_threshold(_message.clone());

    assert(threshold != 0, Errors::THRESHOLD_NOT_SET);
    let modules = self.build_modules_span();
    let mut cur_idx: u8 = 0;
    loop {
        // ...
        if (cur_idx.into() == isms.len()) {
            break ();
        }
        // ...
        cur_idx += 1;
    };
    // ...
}
```

The type of the `cur_idx` variable is `u8`, which can store up to 255. If Aggregation ISM contains more than 255 modules, the `cur_idx` variable may overflow, which will cause this function to revert.

Impact

This could lead to unexpected behavior or function failure in cases with a large number of modules above 255.

Recommendations

Consider preventing Aggregation ISM from being created with more than 255 modules.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit 484fe5e6↗.

3.17. ISM configuration of MailboxComponent is disregarded

Target	mailboxclient_component.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In the Hyperlane protocol, a recipient can specify the ISM to be used by having the public method `interchainSecurityModule()`. The Solidity implementation of MailboxClient contract allows a contract to specify the ISM through providing the getter/setter functions for ISM:

```
abstract contract MailboxClient is OwnableUpgradeable {
    // ...

    IInterchainSecurityModule public interchainSecurityModule;

    // ...

    function setInterchainSecurityModule(
        address _module
    ) public onlyContractOrNull(_module) onlyOwner {
        interchainSecurityModule = IInterchainSecurityModule(_module);
    }

    // ...
}
```

The mailboxclient_component.cairo also has getter/setter functions for its ISM:

```
#[starknet::component]
pub mod MailboxclientComponent {
    // ...

    #[storage]
    struct Storage {
        // ...
        interchain_security_module: ContractAddress,
    }

    // ...
```

```
    #[embeddable_as(MailboxClientImpl)]
    impl MailboxClient<
        // ...
    > of IMailboxClient<ComponentState<TContractState>> {
        // ...

        fn set_interchain_security_module(
            ref self: ComponentState<TContractState>, _module: ContractAddress
        ) {
            let ownable_comp = get_dep_component!(@self, Owner);
            ownable_comp.assert_only_owner();
            assert(_module != contract_address_const::<0>(),
    Errors::ADDRESS_CANNOT_BE_ZERO);
            self.interchain_security_module.write(_module);
        }

        // ...

        fn get_interchain_security_module(
            self: @ComponentState<TContractState>
        ) -> ContractAddress {
            self.interchain_security_module.read()
        }

        // ...
    }

    // ...
}
```

It should be noted that the storage variable does not automatically create the getter method for the variable. In this contract, the function `get_interchain_security_module()` is the getter method.

However, the method the Mailbox utilizes to fetch the ISM of the recipient is the `interchain_security_module()`, not the `get_interchain_security_module()`.

Impact

MailboxClient has getter/setter functions about ISM, which do not affect the logic of contract.

While this does not pose a direct security risk, this issue can still lead to confusion, potential misuse of the contracts, and inefficient code.

Recommendations

Consider renaming the method `get_interchain_security_module` to `interchain_security_module` in order to provide the expected functionality.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit cc203103 ↗.

3.18. Unclear behavior of the function `set_modules`

Target	aggregation.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The following is the code of the Aggregation ISM:

```
fn set_modules(ref self: ContractState, _modules: Span<ContractAddress>) {
    self.ownable.assert_only_owner();
    assert(!self.are_modules_stored(_modules),
    Errors::MODULES_ALREADY_STORED);
    let mut last_module = self.find_last_module();
    let mut cur_idx = 0;
    loop {
        if (cur_idx == _modules.len()) {
            break ();
        }
        let module = *_modules.at(cur_idx);
        assert(
            module != contract_address_const::<0>(),
    Errors::MODULE_ADDRESS_CANNOT_BE_NULL
        );
        self.modules.write(last_module, module);
        cur_idx += 1;
        last_module = module;
    }
}
```

This function appends the given modules to the existing list of modules; not replacing the existing list of modules.

Impact

This inconsistency between the function name and its behavior could lead to potential misuse.

Recommendations

Consider changing the behavior of this function or renaming this function to match the function name to its behavior.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit 6ef5aa2e ↗.

3.19. Incorrect size of `StoreFelt252Array`

Target	store_arrays.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The store_arrays.cairo implements a way to store the `Array<felt252>` as a storage variable:

```
pub impl StoreFelt252Array of Store<Array<felt252>> {
    fn read(address_domain: u32, base: StorageBaseAddress) ->
    SyscallResult<Array<felt252>> {
        StoreFelt252Array::read_at_offset(address_domain, base, 0)
    }

    fn write(
        address_domain: u32, base: StorageBaseAddress, value: Array<felt252>
    ) -> SyscallResult<()> {
        StoreFelt252Array::write_at_offset(address_domain, base, 0, value)
    }

    // ...

    fn size() -> u8 {
        1
    }
}
```

Note that the `size` function returns 1. However, this function should return 255, since this implementation may utilize up to 255 slots for its logic.

Impact

An array can be improperly stored in the storage, which may break the integrity of the stored data.

We believe this code is only used by validator_announce.cairo, which does not directly affect the logic of the rest of the contracts.

Recommendations

Consider modifying the `size` function to return 255 instead of 1.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit `5ec83ab7` ↗.

3.20. Unnecessary class function for signature conversion

Target	validator_announce.cairo		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `convert_to_signature` function is defined as a class function, but it does not interact with the contract state.

Impact

This issue does not pose a security risk but may lead to confusion or suboptimal code organization.

Recommendations

The `convert_to_signature` function can be defined as a free function outside the class structure. If kept within the class, adding self as a parameter makes it consistent with other class methods, though this is not strictly necessary for its functionality.

Remediation

This issue has been acknowledged by Pragma, and a fix was implemented in commit `fcb0a419` ↗.

3.21. Lack of comprehensive test suite

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

Description

During this audit, we observed a number of severe findings that affect the core logic of the codebase. Some of these findings could result in the failure of working on the production environment if not fixed, even if no malicious attack is assumed.

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include more than just surface-level functions. It is important to test the invariants required for ensuring security.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

Impact

Writing comprehensive test suites can prevent many logical errors existing in the codebase. Finding 3.1. ↗ could have been discovered and remediated before this audit if it was attempted to test the behavior of Aggregation ISM that skips ISMs without metadata.

It is also important to write comprehensive negative tests as well as positive tests. This helps not only uncover complicated issues but also understand the exact behavior of the codebase. Finding 3.3. ↗ discovers the straightforward counterexample of the premature optimization, and we believe that writing negative tests would have found the same counterexample and prevented this issue in advance.

We strongly recommend Pragma to strive for 100% code coverage. It has come to our attention that some trivial functions, such as `upgrade()`, do not have corresponding tests. Finding 3.12. ↗ is a case where achieving 100% code coverage could have discovered the issue.

We also recommend enhancing the integration tests because issues affecting the interactions between contracts cannot be easily discovered with unit tests. For instance, Finding 3.7. ↗ discovers the design error of the fee-forwarding mechanism, and we believe it could have been discovered with proper integration tests, including the protocol-fee hook contract.

Recommendations

Consider building a rigorous test suite that includes all functions and possibly attainable edge cases for the Hyperlane protocol.

Remediation

This issue has been acknowledged by Pragma.

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. Same message can be inserted into the Merkle tree hook multiple times

The same message can be inserted into the Merkle tree hook multiple times, as long as it is the latest dispatched message. We discussed if this behavior has been acknowledged and is acceptable on their side, because we were not aware of the way the off-chain components would interact with the Merkle tree hook contract.

The Hyperlane team has acknowledged that this is acceptable behavior and the off-chain component is responsible for handling this case correctly.

4.2. Nonce may overflow

The Mailbox contract has the `nonce` variable, which should be incremented per dispatched message and inserted in all dispatched messages.

The type of the `nonce` variable is `u32`, which is not infeasible to be overflowed. An attacker may spend a substantial amount of gas in order to increment the nonce to

$$2^32 - 1$$

which will disable the `dispatch` function.

Pragma has acknowledged the issue and decided to leave the nonce as the `u32` type as of now. We also agree with their approach, because the size of the nonce is defined in the Hyperlane protocol and the consensus of Hyperlane protocol implementations should be required when they make changes.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Message

The message is the core data structure used by the Hyperlane protocol. It is a packed data structure that contains all the information needed to route a message from one domain to another. The structure of the message is as follows:

```
#[derive(Serde, starknet::Store, Drop, Clone)]
pub struct Message {
    pub version: u8,
    pub nonce: u32,
    pub origin: u32,
    pub sender: u256,
    pub destination: u32,
    pub recipient: u256,
    pub body: Bytes,
}
```

There was an issue that the sender and the recipient were the `ContractAddress` type, which is `felt252`, so if the sender or the recipient is using the 32-bytes addressing mechanism, those will not be a valid address. See Finding 3.8. ↗ for more details.

5.2. Mailbox

The mailbox is the entry point for developers to send and receive messages from.

mailbox contract

The mailbox contract handles message dispatching, processing, and security checks for cross-chain communication.

Key functions for mailbox are as follows:

- `dispatch` — sends messages to the destination domain and recipient
- `process` — receives and processes incoming messages

- `quote_dispatch` — computes quote for dispatching a message to the destination domain and recipient
- `recipient_ism` — determines the ISM for a recipient

The `dispatch` function is to send messages to the destination domain and recipient. The key steps of the function are as below:

1. Build the message using the destination domain, sender, and provided details.
2. Generate a unique message ID.
3. Emit `Dispatch` and `DispatchId` events.
4. Handle fee collection and distribution:
    - Calculate fees for required and default hooks.
    - Check if a sufficient fee is provided.
    - Verify the sender's balance and allowance.
    - Transfer fees to the respective hooks.
5. Execute post-dispatch hooks.

The `process` function is to receive and process incoming messages from other sources. The key steps of the function are as below:

1. Verify the message version and destination.
2. Check if the message has already been delivered.
3. Record the delivery in storage.
4. Determine the recipient's ISM.
5. Emit `Process` and `ProcessId` events.
6. Verify the message using the recipient's ISM.
7. Call the recipient's handle function with the message details.

mailboxclient contract

This contract serves as a `proxy` for the Mailbox client in the Hyperlane cross-chain communication system. It integrates MailboxclientComponent, OwnableComponent, and UpgradeableComponent to provide mailbox functionality with ownership management and upgrade capabilities.

The `constructor` function initializes the contract with mailbox address and owner, and the `upgrade` function allows the owner to upgrade the contract implementation.

mailboxclient_component contract

The `MailboxclientComponent` is a crucial part of the Hyperlane cross-chain messaging system, providing core functionality for managing mailbox interactions, hooks, and ISMs.

MailboxClient exposes functions that allow subclasses to easily send messages to the Mailbox via the mailbox storage variable and permission message delivery via the `onlyMailbox` modifier.

Key functions for mailboxclient_component are as follows:

- `set_hook` — sets the custom hook address
- `set_interchain_security_module` — sets the ISM address
- `_MailboxClient_initialize` — initializes the mailbox client configuration
- `_dispatch` — dispatches a message to a destination domain and recipient
- `quote_dispatch` — computes quote for dispatching a message to the destination domain and recipient
- `initialize` — initializes the mailbox client with a mailbox address

5.3. Hooks

Post-dispatch hooks allow developers to configure additional origin chain behavior with message content dispatched via the Mailbox. This allows developers to integrate third-party/native bridges, make additional chain commitments, or require custom fees all while maintaining a consistent single-call Mailbox interface.

protocol_fee contract

This contract implements a post-dispatch hook for collecting protocol fees in the Hyperlane cross-chain messaging system. It manages fee collection, beneficiary settings, and provides functionality for fee quotes and collection.

Key functions for protocol_fee are as follows:

- `post_dispatch` — processes the protocol fee after message dispatch
- `quote_dispatch` — provides a quote for the protocol fee
- `set_protocol_fee` — sets the protocol fee and can be called by the owner
- `set_beneficiary` — sets the beneficiary of collected fees and can be called by the owner
- `collect_protocol_fees` — transfers collected fees to the beneficiary

merkle_tree_hook contract

This contract implements a post-dispatch hook for maintaining a Merkle tree of dispatched messages in the Hyperlane cross-chain messaging system. It provides functionality for inserting messages into the tree, calculating roots, and managing the tree structure.

Key functions for `merkle_tree_hook` are as follows:

- `post_dispatch` — processes the message after dispatch by inserting it into the Merkle tree
- `quote_dispatch` — provides a quote for the hook operation (returns 0 in this implementation)
- `_insert` — inserts a new node into the Merkle tree
- `_root` — calculates the current root of the Merkle tree
- `_branch_root` — calculates the root given a leaf, branch, and index

> The same message can be inserted into the Merkle tree hook multiple times, as long as it is the latest dispatched message. We discussed with the Pragma team and got the response that this is the same with the Solidity implementation too and it is acceptable behavior. Even if the message is inserted multiple times, it will only be delivered once. See Discussion point 4.1. ↗ for more details.

5.4. Interchain Security modules

Hyperlane is secured by ISMs. ISMs are smart contracts that are responsible for verifying that interchain messages being delivered on the destination chain were actually sent on the origin chain.

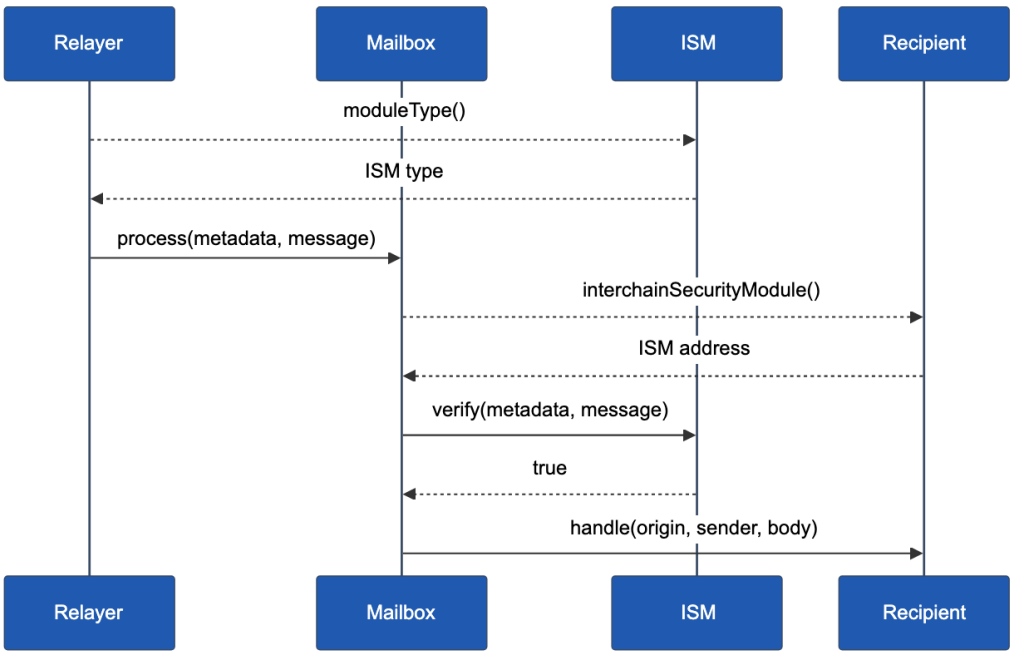
Hyperlane developers can optionally override the Mailbox's default ISM by specifying an application-specific ISM, which they can configure, compose, and customize according to the needs of their application.

The primary function that ISMs must implement is `verify()`. The Mailbox will call `IInterchainSecurityModule.verify()` before delivering a message to its recipient. If `verify()` reverts or returns false, the message will not be delivered.

The `verify()` function takes two parameters:

1. `_metadata`. This consists of arbitrary bytes provided by Relayer. Typically, these bytes are specific to the ISM. For example, for Multisig ISM, `_metadata` must include validator signatures.
2. `_message`. This consists of the Hyperlane message being verified. ISMs can use this to inspect details about the message being verified. For example, Multisig ISM could change validator sets based on the origin chain of the message.

The following shows a simplified sequence diagram of an interchain message being verified and delivered on the destination chain.



Multisig ISM

The Multisig ISM is one of the most commonly used ISM types. These ISMs verify that m-of-n validators have attested to the validity of a particular interchain message. It should be implemented to check if the metadata provided to verify satisfies a quorum of signatures from a set of configured validators.

message_id_multisig contract

This contract implements a multi-sig ISM for message verification in a cross-chain communication system. It manages a set of validators and a threshold for message verification.

The structure of the `message_id_multisig` metadata is as follows:

- [0: 32] Origin Merkle tree address
- [32: 64] Signed checkpoint root
- [64: 68] Signed checkpoint index
- [68:????] Validator signatures (length := threshold * 65)

In order to verify the message, the following checks must pass:

1. The metadata must not be empty.

2. The calculated digest must match the message and metadata.
3. The number of valid signatures must meet or exceed the threshold.
4. Each signature must be from a recognized validator in the list.
5. The threshold must be greater than zero.

merkleroot_multisig contract

This contract implements a Merkle root–based multi-sig ISM for verifying cross-chain messages. It manages a set of validators and a threshold for message verification, incorporating Merkle proofs in the verification process.

The structure of the `merkleroot_multisig` metadata is as follows:

- [0: 32] Origin Merkle tree address
- [32: 36] Index of message ID in Merkle tree
- [36: 68] Signed checkpoint message ID
- [68:1092] Merkle proof
- [1092:1096] Signed checkpoint index (computed from proof and index)
- [1096:????] Validator signatures (length := threshold * 65)

In order to verify the message, the following checks must pass:

1. The metadata must not be empty.
2. The message index in the metadata must be valid (not greater than the signed index).
3. The Merkle proof in the metadata must be valid for the given message.
4. The calculated digest, including the Merkle root, must be correct.
5. The number of valid signatures must meet or exceed the threshold.
6. Each signature must be from a recognized validator in the list.
7. The threshold must be greater than zero.

Routing ISM

Developers can use a Routing ISM to delegate message verification to a different ISM. This allows developers to change security models based on message content or application context.

This ISM simply switches security models depending on the origin chain of the message. A simple use case for this is to use different Multisig ISM validator sets for each chain.

Eventually, you could imagine a DomainRoutingIsm routing to different light-client–based ISMs, depending on the type of consensus protocol used on the origin chain.

Aggregation ISM

Developers can use an Aggregation ISM to combine security from multiple ISMs. Simply put, an Aggregation ISM requires that m-of-n ISMs verify a particular interchain message.

Developers can configure, for each origin chain, a set of n ISMs, and the number of ISMs needed to verify a message.

Aggregation ISMs can aggregate the security of any ISMs. For example, users can deploy a Multisig ISM with their own validator set and deploy an Aggregation ISM that aggregates that ISM with the Hyperlane default ISM.

The structure of the merkleroot_multisig metadata is as follows:

- [????:????] Metadata start/end `uint32` ranges, packed as `uint64`
- [????:????] ISM metadata, packed encoding

> There was an issue that the valid message could be unverifiable in the `verify` function. See Finding 3.1. ↗ for more details.

In order to verify the message, the following checks must pass:

1. The threshold must be set and not zero.
2. The number of modules must be sufficient to potentially meet the threshold.
3. Each module in the list must be checked if it has corresponding metadata.
4. For modules with metadata, their individual verify function must return true.
5. The number of successful verifications must reach the threshold.
6. No module verification should fail during the process.
7. The verification process must continue until the threshold is met or all modules are checked.
8. After checking all modules, the number of successful verifications must equal the threshold.
9. No assertions should fail or errors be thrown during the entire process.

noop ISM

The `verify` function always returns true so that all verifications will succeed.

```
fn verify(self: @ContractState, _metadata: Bytes, _message: Message) -> bool {
    true
}
```

Pausable ISM

The `verify` function returns true if it is not paused by the owner.

```
fn verify(self: @ContractState, _metadata: Bytes, _message: Message) -> bool {
    self.pausable.assert_not_paused();
    true
}
```

Validator Announcing contract

The `Validators` announce their signature storage location so that the relayer can fetch and verify their signatures.

> The `announce` function in the contract handles a few steps to announce a validator-signature storage location. This was different from the original Solidity implementation, but it has been acknowledged. See Finding 3.15. ↗ for more details.

1. It converts input parameters and generates a `replay_id` using a Poseidon hash.
2. It checks for replay protection to prevent duplicate announcements.
3. It calculates the announcement digest and verifies the provided signature.
4. It adds the validator to the list if new or updates existing validator information.
5. It stores the new storage location for the validator and updates related counts.
6. Reverts if the announcement already occurred or if the given signature is invalid.

6. Assessment Results

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

During our assessment on the scoped Hyperlane Starknet contracts, we discovered 21 findings. Five critical issues were found. Four were of high impact, four were of medium impact, six 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.