Zellic

Prepared for
Abacus Works
Hyperlane

Prepared by
Junghoon Cho
Ulrich Myhre
Zellic

August 26, 2025

Hyperlane - Radix

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 Hyperlane from August 15th to August 22nd, 2025. During this engagement, Zellic reviewed Hyperlane - Radix'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:

- Is it possible to bypass Interchain Security Model (ISM) verification?
- Is it possible for sender addresses to be spoofed?
- Is the implementation of the Hyperlane protocol compatible with the standard?
- Are all access controls configured correctly?

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 Hyperlane - Radix targets, we discovered four findings. No critical issues were found. One finding was of high impact, one was of medium impact, one was of low impact, and the remaining finding was informational in nature.

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

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Hyperlane - Radix

Hyperlane contributed the following description of Hyperlane - Radix:

> This project is an implementation of Hyperlane for the Radix DLT, designed for a seamless interchain communication following the Hyperlane spec.

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:

Hyperlane - Radix Targets

Type	Scrypto
Platform	Radix

Target	hyperlane-radix
Repository	https://github.com/hyperlane-xyz/hyperlane-radix ↗
Version	`46932db2fef871d3165c115a598a92198b1fa750`
Programs	`src/*`

2.4. Project Overview

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

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 ↗

Ulrich Myhre
Engineer
unblvr@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

August 15, 2025	Kick-off call
August 15, 2025	Start of primary review period
August 22, 2025	End of primary review period

3. Detailed Findings

3.1. Warp Payload doesn't support custom divisibility

Target	warp_payload.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

When consuming a WarpPayload message, the `amount` field is treated as a `Decimal` type.

```
pub struct WarpPayload {
    /// 32-byte Address in destination convention
    pub recipient: Bytes32,
    /// 32-byte Amount
    pub amount: Decimal,
}

impl From<&RawWarpPayload> for WarpPayload {
    fn from(m: &RawWarpPayload) -> Self {
        let recipient: Bytes32 = m[0..32].into();

        // Next 32 bytes encode the amount
        // In the future it might be possible that the warp payload carries
    additional metadata
        let mut b = m[32..64].to_vec();
        b.reverse();
        let amount = U256::from_le_bytes(b.as_ref());

        let amount = I192::try_from(amount).expect("Invalid payload");
        let amount = Decimal::from_attos(amount);

        Self { recipient, amount }
    }
}
```

The incoming amount is 32 bytes or 256 bits, but the `Decimal` type on the other hand, is just valid for values up to 192 bits. This is because the Radix Decimal uses `Attos`, or units of size 10^{-18}. The valid range for a `Decimal` m is `-2^(192 - 1) <= m < 2^(192 - 1)`, meaning values above this amount will crash. Additionally, the built-in constants like `Decimal::one()` is actually equal to 1000000000000000000 in a WarpPayload.

More importantly, this code does not handle custom divisibility and incorrectly assumes that the token always uses 18 decimals.

Impact

If another chain uses custom decimals for their divisibility, this could lead to the payload being discarded or being parsed incorrectly. When it is discarded due to parsing errors, there are no custom errors displaying what went wrong. If the WarpRoute uses custom divisibility, this leads to values being scaled incorrectly.

Example: If someone sends 1,000,000 subunits of a token that uses divisibility 6, this turns into "0.000000000001" and will not correctly be parsed into the ResourceManager, which would have its own divisibility configured.

In general, it blocks everyone from setting up a WarpRoute with custom divisibility at all. Only 18 decimals will be supported.

Recommendations

Scale the amount by the proper divisibility and throw a custom error when a chain with custom divisibility tries to send an incompatible amount. Properly document the range of the incoming amount. Do not use `Decimal` for the amount, as it does not take the underlying divisibility into account. Instead scale it at the ingress.

Remediation

For the crash issue, Hyperlane states:

> That the Radix implementation only supports 24 bytes (instead of the 32) is totally fine and must be respected by the WarpRoute owner.

For the divisibility issue:

This issue has been acknowledged by Hyperlane, and a fix was implemented in commit `1cc52db8` ↗.

3.2. Incorrect message digest generated in `verify` function

Target	src/contracts/isms/merkle_root_multisig_ism.rs		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Low	**Impact**	Medium

Description

The `verify` function of the `merkle_root_multisig_ism` module computes a digest with the passed metadata and the user's message, and calls the `verify_multisig` function with it as an argument for signature verification.

```
pub fn verify(&mut self, metadata: Vec<u8>, message: Vec<u8>) -> bool {
    let metadata: MultisigIsmMerkleRootMetadata = metadata.into();

    let message: HyperlaneMessage = message.into();
    let signed_root = merkle_root_from_branch(
        message.id(),
        &metadata.merkle_proof,
        metadata.message_index,
    );

    let digest = message.digest(
        metadata.origin_merkle_tree_hook,
        signed_root.into(),
        metadata.message_index,
    );

    verify_multisig(
        digest,
        &metadata.validator_signatures,
        &self.validators,
        self.threshold,
    )
}
```

At this point, the last argument of `digest` should be the checkpoint index signed by the validator, `metadata.signed_checkpoint_index`. However, in the current implementation, the leaf index of the message in the Merkle tree, `metadata.message_index`, is being passed instead.

Impact

If the leaf index of the message in the Merkle tree on the source chain is not synchronized with the checkpoint index at the time the validator signs, a valid message may fail to be verified.

Recommendations

Modify the `verify` function as below:

```
let digest = message.digest(
    metadata.origin_merkle_tree_hook,
    signed_root.into(),
    metadata.message_index,
    metadata.signed_checkpoint_index,
);
```

Remediation

This issue has been acknowledged by Hyperlane, and a fix was implemented in commit `30af26bf` ↗.

3.3. Reversed execution order of post-dispatch hooks

Target	src/contracts/mailbox.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

In Hyperlane, a Mailbox is configured with the following two post-dispatch hooks:

1. `required` : This hook is always invoked on every `dispatch` call, along with the value that covers the required fee.
2. `default` : This hook can be overridden by the caller, and is invoked after the `required` hook, with the remaining value.

In the current version, the execution of these hooks is implemented as follows:

```
let default_hook = hook.or(self.default_hook);
if let Some(default_hook) = default_hook {
    let result = ScryptoVmV1Api::object_call(
        default_hook.as_node_id(),
        "post_dispatch",
        scrypto_args!(hook_metadata.clone(), hyperlane_message.clone(),
    payment),
    );
    payment = scrypto_decode(&result)
        .expect(&format_error!("failed to decode post_dispatch result"));
            }
if let Some(required_hook) = self.required_hook {
    let result = ScryptoVmV1Api::object_call(
        required_hook.as_node_id(),
        "post_dispatch",
        scrypto_args!(hook_metadata, hyperlane_message.clone(), payment),
    );
    payment = scrypto_decode(&result)
        .expect(&format_error!("failed to decode post_dispatch result"));
}
```

As shown above, the `default_hook` - whether it is provided by the caller or the configured default hook - is executed before the `required_hook`.

Impact

If resources are consumed during the execution of the `default` hook, the `required` hook may fail due to insufficient resources, potentially causing the transaction to revert. Furthermore, this behavior represents a mismatch between the actual implementation and the execution order of the two hooks that is specified in the Hyperlane protocol documentation, and may cause confusion for users.

Recommendations

Modify the `dispatch` function so that the execution of `self.required_hook` precedes that of the `default_hook`.

Remediation

This issue has been acknowledged by Hyperlane, and a fix was implemented in commit `791c65c6` ↗.

3.4. Unused `InstantiationEvent` event

Target	src/contracts/mailbox.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

```
#[derive(ScryptoSbor, ScryptoEvent)]
pub struct InstantiationEvent {
    pub local_domain: u32,
}
```

In mailbox.rs, an `InstantiationEvent` is defined to be emitted when the `Mailbox` component is instantiated. However, in the current version of the code, this event is not emitted within the `instantiate` function.

Impact

Leaving unused events in the codebase reduces maintainability and may potentially lead to confusion for off-chain systems that expect the event.

Recommendations

Either remove the `InstantiationEvent`, or ensure that it is emitted within the `instantiate` function.

Remediation

This issue has been acknowledged by Hyperlane, and a fix was implemented in commit `4f989776` ↗.

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. `required` hook may potentially be executed twice

```
let default_hook = hook.or(self.default_hook);
if let Some(default_hook) = default_hook {
    let result = ScryptoVmV1Api::object_call(
        default_hook.as_node_id(),
        "post_dispatch",
        scrypto_args!(hook_metadata.clone(), hyperlane_message.clone(),
    payment),
    );
    payment = scrypto_decode(&result)
        .expect(&format_error!("failed to decode post_dispatch result"));
            }
if let Some(required_hook) = self.required_hook {
    let result = ScryptoVmV1Api::object_call(
        required_hook.as_node_id(),
        "post_dispatch",
        scrypto_args!(hook_metadata, hyperlane_message.clone(), payment),
    );
    payment = scrypto_decode(&result)
        .expect(&format_error!("failed to decode post_dispatch result"));
}
```

In the `dispatch` function, the post-dispatch hook `default_hook` can be overridden by the caller. If the caller passes a component address identical to that of the `required_hook` as the `hook` parameter, then the `post_dispatch` function of the `required_hook` may be executed twice.

If the `required_hook` is configured as an `InterchainGasPaymaster`, this could lead to duplicate payments; however, such a situation would result from end-user misconfiguration and cannot be prevented.

If the `required_hook` is configured as a `MerkleTreeHook`, the same message may be inserted into the merkle tree twice. This does not compromise the functionality or availability of the hook or ISM, but the `InsertedIntoTreeEvent` would be emitted twice with different indices.

4.2. Typos in the codebase

In the current version of the codebase, there are several typos in comments and module names. Some examples are shown below.

```
/// We can't assume that [...] only assisated with one resource)
/// We can't assume that [...] only associated with one resource)
```

```
mod message_id_mutlsig_ism {
mod message_id_multisig_ism {
```

```
/// that might be needed in order to perfrom a remote transfer
/// that might be needed in order to perform a remote transfer
```

To improve the project's maintainability and prevent potential errors and confusion, it is recommended to remove these typos.

This issue has been acknowledged by Hyperlane, and a fix was implemented in commit abcbeb86 ↗.

4.3. Test suite

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.

This project has decent test coverage, but there are still untested edge cases and negative test scenarios. For example, there are currently no tests ensuring that already-processed messages are rejected when processed by the Mailbox, or that functions such as `set_required_hook`, which should only be callable by the owner, are properly access-controlled.

Therefore, to ensure the project's overall security, we recommend implementing tests for these uncovered execution paths.

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.

4.4. Compromised owner account could maliciously update resource metadata

After our review period ended, Hyperlane requested a review of PR #28 ↗, an additional update to the Token Metadata.

In PR #28 ↗, a change was made to allow the owner of the `hyp_token` to update the resource metadata. This metadata includes the resource's `name`, `symbol`, and `description`.

In our opinion, having the `description` be updatable makes sense, as the description can contain information about the resource, including official links. There are real scenarios where this information may need to be updated.

However, allowing the `name` and `symbol` to be updatable opens a potential attack vector where a bad actor could, after compromising the owner's private key, modify the `name` and `symbol` to impersonate another token.

Although resources in general should be identified by their unique resource address, layman users are in danger of falling for scams using the modified token `name` and `symbol`, especially since this token would be a real token with real market value.

We've determined that the severity of such an attack would be medium, since it does require users to fall for a scam. We've also determined that the likelihood is low, since it requires a private key compromise.

Our recommendation is to prevent the `name` and `symbol` from being updatable. The `description` can stay `updatable`.

5. System Design

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

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

5.1. Component: Mailbox

Description

Mailbox is the chain-local endpoint of Hyperlane messaging. It is responsible for dispatching outbound messages and running any configured hooks, and processing inbound messages after the configured Interchain Security Module (ISM) verifies them.

Mailbox provides a public interface for following actions:

1. Dispatch : Users invoke a `dispatch` function in order to send interchain messages. When called, it constructs a `HyperlaneMessage` from the arguments passed, including the nonce, the domain identifier of the source chain, the domain identifier of the destination chain, the sender's address, the recipient's address, and the message to be sent. It then invokes the configured post-dispatch hooks, emit events to notify that a message is sent, and returns any leftover payment buckets to the caller.

2. Process : Relayers invoke a `process` function in order to deliver interchain messages to this component. When called, it passes the message and metadata to the recipient's ISM to verify the message. If the message is verified, the recipient's `handle` function will be invoked with the delivered message as an argument, finalizing the delivery.

3. Quote : Users can call the `quote_dispatch` function to check the fee incurred when invoking `dispatch`.

On instantiation, the component reserves its address, mints a 1-of-1 owner badge, and sets authentication on methods so that only the owner can change the configured addresses of the ISM and hooks.

Invariants

- The destination domain of the message must be equal to the Mailbox's `local_domain`. This is enforced by a check in the `process` function, preventing any mis-routed messages to be processed.
- Each message can be processed only once. This is enforced by a check in the `process` function, preventing replay of messages.

- The sender of the message must be equal to the address of the account or component that called the `dispatch` function. This is enforced by a check in the `dispatch` function, preventing the sender of the message from being spoofed.

Test coverage

Cases covered

- Instantiation of the Mailbox component
- `process` : Domain mismatch is rejected
- `process` : Unsupported message version is rejected
- `process` : Revert when recipient doesn't implement expected app/ISM interface
- `dispatch` : Message is successfully dispatched
- `dispatch` : Forged sender address is rejected

Cases not covered

- `process` : Replay protection
- `dispatch` : Hook interaction
- `set_default_ism/set_default_hook/set_required_hook` : Access control

After the primary audit period, Hyperlane introduced negative tests covering these cases in PR 29 ↗ to improve the project's test coverage.

Attack surface

- Externally controllable inputs to `dispatch` function :
    - `destination_domain`, `recipient_address` and `message_body` affect message ID, hook execution, and emitted events. Malicious payloads can target the recipient app and should be gated by recipient logic on the destination chain.
    - Callers can route payment through setting arbitrary `hook` and `hook_metadata` when calling this function; a malicious hook can steal the caller's own payment. The owner-set `required_hook` is always invoked; if the owner configures a malicious hook, it can overly extract the user payments. However, this is an expected centralization risk akin to setting fee collectors.
    - Hooks can consume the `payment` provided by the caller. If the payment is insufficient to cover the gas, the transaction will be reverted.
    - If `claimed_account_sender` is forged, the `verify_message_sender` function prevents spoofing by asserting rules against the current caller's proofs or global-caller context.
- Externally controllable inputs to `process` function :
    - `metadata` and `raw_message` are fully controlled by an attacker or relayer. The

check for the destination chain and replay protection are performed as mentioned above, and the final verification is carried out by the recipient's ISM.

    - `visible_components` is forwarded to the recipient. Recipients may require specific component addresses, e.g., for deposits. A malicious relayer can pass arbitrary addresses, and the recipients are responsible for rejecting those with their own logic.

- Owner-controlled configuration risk : The owner can set or modify `default_ism`, `default_hook` and `required_hook`. If compromised or malicious, they can weaken verification or overly extract user payments.

6. Assessment Results

During our assessment on the scoped Hyperlane - Radix targets, we discovered four findings. No critical issues were found. One finding was of high impact, one was of medium impact, one was of low impact, and the remaining finding was 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.