THREE SIGMA

M^0 Portal-lite Bridge

Security Review

M^0

Disclaimer

Security Review

M^O Portal-lite Bridge

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Security Review

M^O Portal-lite Bridge

Table of Contents

Summary

Security Review

M^O Portal-lite Bridge

Summary

Three Sigma audited M0 Portal-lite in a 2.4 person week engagement. The audit was conducted from 28/04/2025 to 05/05/2025.

Protocol Description

M0 Portal Lite is a streamlined blockchain bridging solution within the M0 multichain ecosystem, designed exclusively for EVM-compatible chains. Following a one-to-many communication model, Spoke chains interact solely with the Ethereum Hub, but not directly with each other. It employs a lock-and-release mechanism on Ethereum and a mint-and-burn process on Spokes, secured via Hyperlane cross-chain messaging. This protocol ensures consistent yield earning and governance for the $M token across all chains.

Scope

Security Review

M^O Portal-lite Bridge

Scope

Filepath	nSLOC
src/abstract/components/Blacklistable.sol	40
src/abstract/MExtension.sol	79
src/access/PausableOwnable.sol	40
src/bridges/hyperlane/HyperlaneBridge.sol	39
src/bridges/hyperlane/libs/StandardHookMetadata.sol	52
src/HubPortal.sol	66
src/libs/BytesParser.sol	64
src/libs/PayloadEncoder.sol	79
src/libs/SafeCall.sol	8
src/libs/TypeConverter.sol	9
src/MYieldToOne.sol	103
src/Portal.sol	163
src/SpokePortal.sol	61
Total	**803**

Methodology

Security Review

M^O Portal-lite Bridge

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

M^O Portal-lite Bridge

Project Dashboard

Application Summary

Name	Portal-lite
Repository	https://github.com/m0-foundation/m-portal-lite/ https://github.com/felixprotocol/m-extensions
Commit	m-portal-lite(099ea8e) m-extensions(12442fa)
Language	Solidity
Platform	Ethereum

Engagement Summary

Timeline	28/04/2025 to 05/05/2025
№ of Auditors	2
Review Time	2.4 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	1	1	0
High	0	0	0
Medium	0	0	0
Low	2	1	1
None	0	0	0

Category Breakdown

Suggestion	0
Documentation	0
Bug	3
Optimization	0
Good Code Practices	0

Risk Section

Security Review

M^O Portal-lite Bridge

Risk Section

- **Administrative Key Management Risks**: The system makes use of administrative keys to manage critical operations. Compromise or misuse of these keys could result in unauthorized actions and financial losses.
- **Upgradability Risk**: Administrators have the capability to update contract logic at any moment due to the project's upgradable design. Contract upgradability, while beneficial, also poses the risk of harmful modifications if administrative privileges or upgrade processes are compromised.

Findings

Security Review

M^O Portal-lite Bridge

Findings

3S-M^0 Portal-lite-C01

The protocol accepts forged messages due to missing sender authentication

Id	3S-M^0 Portal-lite-C01
Classification	Critical
Impact	Critical
Likelihood	High
Category	Bug
Status	Addressed in #c8f7eda.

Description

The core functionality of the protocol is the exchange of tokens and information between the Hub and the Spoke — two components located on different chains. This is achieved using the Hyperlane service, which enables cross-chain messaging while ensuring message authenticity. The integration with Hyperlane is handled in the **HyperlaneBridge** contract. Message sending is done via the **sendMessage** function, which calls the **dispatch** function of Hyperlane's **Mailbox** contract. **Mailbox** is the main Hyperlane contract responsible for sending and receiving messages. Within **Mailbox.dispatch**, the message is built, and one of its fields is **sender**, which is set to **msg.sender**. In this case, **msg.sender** is the address of HyperlaneBridge.

When receiving messages, the relayer calls **Mailbox.process**, which then invokes **HyperlaneBridge.handle** on the recipient chain. In handle, the only check is that **msg.sender** is the **Mailbox** address, and then it calls **Portal.receiveMessage**. There, it checks that **msg.sender** is the **HyperlaneBridge** address and proceeds to process the message through **_receiveMLikeToken** or **_receiveCustomPayload**.

However, at no point in this workflow is there a check that the **sender** field (included in the dispatched message) actually corresponds to the **HyperlaneBridge** on the source chain. As a result, anyone can send a message to either side and trigger any available action — such as transferring tokens, modifying the index and so on.

This allows a malicious user to craft a message pretending to transfer tokens from the Spoke to the Hub, without actually holding or burning tokens on the Spoke, and thereby unlock tokens on the Hub. This could lead to complete draining of tokens from the Hub.

Steps to Reproduce:

1. The attacker deploys a contract on the Spoke that mimics the functionality of the HyperlaneBridge.

2. They craft a fake message that requests a token transfer from Spoke to Hub, without actually burning any MTokens.

3. The Hub-side implementation accepts the message due to lack of sender validation and unlocks the requested amount of tokens.

4. As a result, the attacker drains all MTokens from the HubPortal.

Recommendation

Before accepting a message, ensure that the **sender** field provided by Hyperlane matches the expected **peer[sourceChainId_]**.

3S-M^0 Portal-lite-L01

Stale index on Spoke chain leads to incorrect yield and transfer calculations

Id	3S-M^0 Portal-lite-L01
Classification	Low
Impact	Low
Likelihood	Medium
Category	Bug
Status	Addressed with note: “We'll have a bot running to propagate M token index to Spoke chains”

Description

In M^0, the accumulated yield is calculated based on a coefficient called the earning index. In the context of the bridge implementation, the index value from the Hub is synchronized with the Spoke (it is not calculated based on rate and time) so that users on the respective Spoke chain can utilize the accumulated yield. This synchronization occurs when tokens are sent or when a message is explicitly sent using the **sendMTokenIndex** function.

It turns out there is no mechanism to guarantee regular synchronization of the index between the Hub and the Spoke. If a long period passes without synchronization, the index value on the Spoke could become significantly lower than that on the Hub.

This directly impacts the MToken functions that rely on the index:

1) Token transfers between users. During a transfer, the amount by which the balances are adjusted is calculated using the functions **_getPrincipalAmountRoundedUp** and **_getPrincipalAmountRoundedDown**.

```
// If this is an in-kind transfer, then...
if (senderIsEarning_ == _balances[recipient_].isEarning) {
    // NOTE: When subtracting a present amount from an earner, round the principal up in favor of the protocol.
    return
        _transferAmountInKind(
            sender_,
            recipient_,
            senderIsEarning_ ? _getPrincipalAmountRoundedUp(safeAmount_) : safeAmount_
```

```
        );
}
// If this is not an in-kind transfer, then...
if (senderIsEarning_) {
    // sender is earning, recipient is not
    _subtractEarningAmount(sender_, _getPrincipalAmountRoundedUp(safeAmount_));
    _addNonEarningAmount(recipient_, safeAmount_);
} else {
    // sender is not earning, recipient is
    _subtractNonEarningAmount(sender_, safeAmount_);
    _addEarningAmount(recipient_, _getPrincipalAmountRoundedDown(safeAmount_));
}
```

In these cases, the **presentAmount** is divided by the index to determine the principal amount. If the index is lower than it should be due to staleness, the calculated principal amount transferred between users will be larger than if the up-to-date index were used.

2) Starting to earn. This is done through the **startEarning** function. It takes the current balance and converts it into principal using **_getPrincipalAmountRoundedDown**. If the index is stale and therefore lower than it should be, the user will receive more principal and therefore more yield than they should.

The same applies analogously to **stopEarning**.

```
uint112 principalAmount_ = _getPrincipalAmountRoundedDown(amount_);
_balances[account_].rawBalance = principalAmount_;
unchecked {
    principalOfTotalEarningSupply += principalAmount_;
    totalNonEarningSupply -= amount_;
}
```

Recommendation

We propose two possible solutions, both with their tradeoffs:

1) Implement an off-chain bot that regularly synchronizes the index between the Hub and the Spoke. This solution comes at a higher cost, and does not reduce the errors to zero (but close).

2) Restrict the ability to call **startEarning** so that it can only be triggered from the Hub, which would also send the fresh index to the Spoke. This solution solves the global accountancy errors due to **startEarning** but not the errors during transfers.

3S-M^0 Portal-lite-L02

Lack of message sequencing in cross-chain communication allows registry state inconsistencies

Id	3S-M^0 Portal-lite-L02
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Acknowledged with note: “The probability of it is very low as updates of Registrar keys or list status are not frequent. If message reordering occurs, it can be solved by propagating the same key again.”

Description

The HyperlaneBridge contract facilitates cross-chain messaging between the Hub and Spoke chains without implementing a message sequencing mechanism. This creates a vulnerability where registry-related state changes can be processed out of order, potentially leading to inconsistent states between chains.

In the current implementation, messages are sent via **HyperlaneBridge::sendMessage**:

```
function sendMessage(
    uint256 destinationChainId_,
    uint256 gasLimit_,
    address refundAddress_,
    bytes memory payload_
) external payable returns (bytes32 messageId_) {
    if (msg.sender != portal) revert NotPortal();
    // No sequencing or nonce mechanism present
    messageId_ = mailbox_.dispatch{value: msg.value}(
        destinationDomain_,
        peer_,
        payload_,
        metadata_
```

```
    );
}
```

And received via **Portal::receiveMessage**:

```
function receiveMessage(
    uint256 sourceChainId_,
    address sender_,
    bytes calldata payload_
) external {
    if (msg.sender != bridge) revert NotBridge();
    PayloadType payloadType_ = payload_.getPayloadType();
    if (payloadType_ == PayloadType.Token) {
        _receiveMLikeToken(sourceChainId_, sender_, payload_);
        return;
    }
    _receiveCustomPayload(payloadType_, payload_);
}
```

Neither function includes a mechanism to track message order or reject out-of-sequence messages. While token transfers are independent operations where ordering is less critical, and index updates are protected by value comparison checks (**if (index_ > _currentIndex())**), registry operations remain vulnerable:

1. Registry key updates (**PayloadType.Key**): No checks prevent an older key value from overwriting a newer one

2. List status changes (**PayloadType.List**): No sequence verification protects against outdated list status changes

Steps to Reproduce:

1. An admin sends message A to update a registry key to value X through **HubPortal::sendRegistrarKey**

2. Shortly after, the admin sends message B to update the same key to value Y

3. Due to cross-chain latency or validator priorities, message B arrives and is processed first

4. Message A arrives later and is processed by **SpokePortal::_receiveCustomPayload**, overwriting the newer value Y with the older value X

5. The system now has an inconsistent registry state where an outdated key value is applied after a newer one

---

Recommendation

Implement a per-source chain nonce mechanism to ensure messages are processed in the correct order for registry operations.