OtterSec

BAM OFT TON

Security Assessment

May 23rd, 2025 — Prepared by OtterSec

Robert Chen r@osec.io

Samuel Bétrisey sam@osec.io

Table of Contents

01 — Executive Summary

Overview

LayerZero Labs engaged OtterSec to assess the `BAM OFT` programs, a white-label version of Ethena OFT. A comparison of the source codes and of the resulting assembly codes was performed. This assessment, conducted on May 23rd, 2025, concluded that no vulnerabilities were introduced. This report also details the original findings from the Ethena OFT audit, which have already been remediated in the white-label version. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 8 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability where the function responsible for asserting the Opcode admin, extracts the opcode from a fixed offset utilizing unsanitized data, enabling the execution of unauthorized operations (OS-BAM-ADV-00). Additionally, in the receive function, the remote Eid should correspond to source Eid as the packet originates from a remote source. However, the remote Eid provided to the receiveOFT method is incorrect, preventing proper rate limit refills (OS-BAM-ADV-01). Furthermore, in the token admin module, the initialization method allows repeated calls, enabling the owner to alter the minter contract address, blocking the super admin and opcode admins from accessing the original contract (OS-BAM-ADV-02).

We also recommended marking the receiveOFT function as impure to ensure consistency and adherence to coding best practices (OS-BAM-SUG-00), and highlighted an error in the comment calculating the size of the storage OS-BAM-SUG-01). We further made a suggestion regarding a deserialization inconsistency in the transfer ownership functionality (OS-BAM-SUG-02).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/ton-bam-oft. This audit was performed against 7653327.

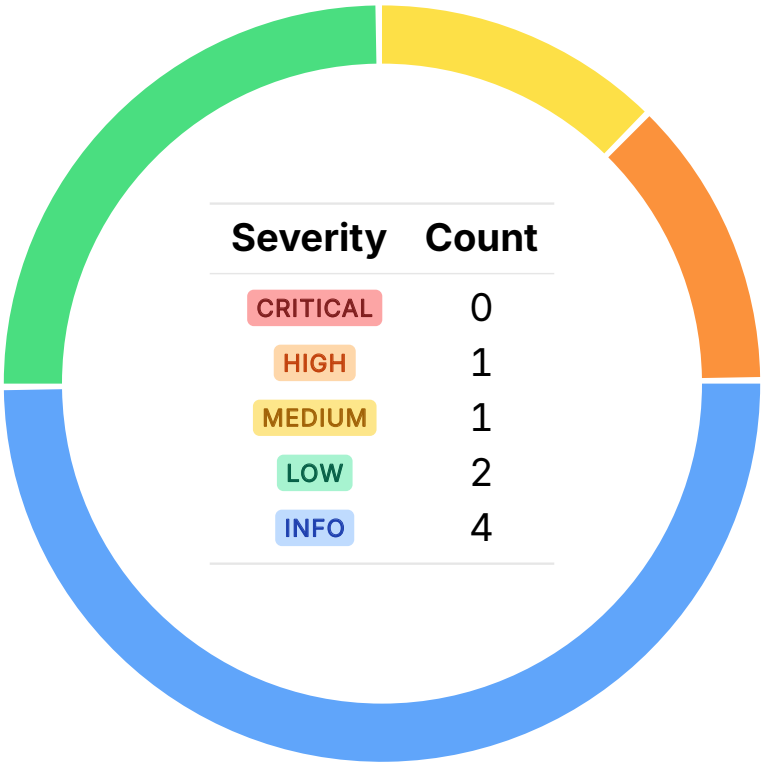
A brief description of the program is as follows:

Name	Description
BamOFT	FunC implementation of the OFT OApp.
Token	FunC implementation of the Jetton minter and wallet.
TokenAdmin	Contract administering the Jetton.

03 — Findings

Overall, we reported 8 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	1
MEDIUM	1
LOW	2
INFO	4

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-BAM-ADV-00	HIGH	RESOLVED	The deserialization process is inconsistent, as `assertOpcodeAdmin` extracts the `opcode` from a fixed offset utilizing unsanitized data, creating a vulnerability that may be exploited by malicious `opcode` admins.
OS-BAM-ADV-01	MEDIUM	RESOLVED	`_lzReceiveExecute` assigns the local `EID` instead of the remote `EID`, preventing proper rate limit refills.
OS-BAM-ADV-02	LOW	RESOLVED	`initialize` allows repeated calls, enabling the owner to alter the `minterContract` address, blocking the super admin and opcode admins from accessing the original contract.
OS-BAM-ADV-03	LOW	RESOLVED	Due to an error in the opcode whitelist, the OApp cannot configure custom settings for its channel.

Inconsistent Deserialization HIGH OS-BAM-ADV-00

Description

The vulnerability arises from inconsistent deserialization and verification of the `opcode` across two contexts. In `assertOpcodeAdmin`, the `opcode` is extracted from a fixed offset utilizing `md::ExecuteParams::getOpcode()`, an unsanitized message data object. However, in `callContract`, the object is sanitized, and fields are retrieved via `typeinfo`, which may not align with the fixed offset.

>_ *ethenaoft-ton/src/tokenAdmin /handler.fc* FUNC

```
() assertOpcodeAdmin(cell $executeParams) impure inline {
    int opcode = $executeParams.md::ExecuteParams::getOpcode();
    int opcodeAdmin = getOpcodeAdmin(opcode);
    throw_unless(TokenAdmin::ERROR::OnlyAdmin, getCaller() == opcodeAdmin);
}
```

A malicious `opcode` admin could exploit inconsistent deserialization by sending a message with an allowed `opcode` at the fixed offset, which will pass verification in `assertOpcodeAdmin`. However, a different `opcode` value may be provided in the properly deserialized `typeinfo` within `callContract`, enabling the execution of unauthorized operations.

>_ *ethenaoft-ton/src/tokenAdmin /handler.fc* FUNC

```
tuple callContract(cell $DispatchCommand) impure inline {
    (cell $storage, tuple actions) = preamble();
    actions~pushAction<rawCall>(
        getMinterContract(),
        buildMessageForOpcode(
            md::DispatchCommand::sanitize($DispatchCommand)
        )
    );

    return actions;
}
```

Furthermore, `md::ExecuteParams::getOpcode`, designed for a `md::ExecuteParams` object, is currently utilized with `md::DispatchCommand`, relying on the assumption that the `opcode` field exists at the same offset in both structures. While this is valid in the current implementation, any future changes to the field layout will result in issues due to offset mismatches.

Remediation

Ensure all input messages are sanitized and utilize the same deserialization logic in both `assertOpcodeAdmin` and `callContract` for consistency.

Patch

Resolved in 2cfe5bd.

Incorrect EID Attribution MEDIUM OS-BAM-ADV-01

Description

The vulnerability arises from utilizing the wrong `EID` when processing an incoming packet in `_lzReceiveExecute`. In `_lzReceiveExecute`, the code attempts to extract the `remoteEid` to be passed to `receiveOFT`. However, it incorrectly extracts the destination `EID` (`DstEid`) of the path in the packet rather than the source `EID` (`SrcEid`).

>_ *ethena-usde-internal/ethenaoft-ton/src/ethenaOFT/handlerEthena.fc* FUNC

```
(cell, tuple) _lzReceiveExecute(cell $storage, tuple actions, cell $Packet) impure inline {
    cell message = $Packet.lz::Packet::getMessage();
    int remoteEid = $Packet.lz::Packet::getPath().lz::Path::getDstEid();
    [...]
    return ($storage, actions);
}
```

The destination `EID` is the local endpoint (the recipient of the packet), whereas the source `EID` represents the sender's endpoint from where the packet originated. Since `receiveOFT` is passed the wrong `EID` (local `EID` instead of remote `EID`), the rate-limit accounting for the remote sender will not be updated, as refilling the rate limit is subject to correctly attributing messages to their origin.

Remediation

Correct the `EID` extraction logic such that the `remoteEid` corresponds to the actual source of the packet.

Patch

Resolved in 9174f78.

Unrestricted Reinitialization of Minter Contract LOW OS-BAM-ADV-02

Description

`tokenAdmin::initialize` may be called repeatedly by the owner (which might be different than the super admin). Each call allows the overwriting of the `TokenAdmin::minterContract` address in the contract's storage. This change in the `minterContract` address will prevent the super admin and opcode admins from accessing the original `minterContract`.

>_ *ethenaoft-ton/src/tokenAdmin/handler.fc* FUNC

```
;; Step 2: initialize
;; mdAddress: (md=null, address=proxy), they're both 256.
tuple initialize(cell $Address) impure inline {
    (cell $storage, tuple actions) = preamble();
    setContractStorage(
        $storage.cl::set(
            TokenAdmin::minterContract,
            $Address.Address::getAddress()
        )
    );

    return actions;
}
```

Remediation

Store the initialization state in the contract to prevent subsequent initialization calls. Alternatively, set the `minterContract` during deployment in the initial state and remove the `initialize` function.

Patch

Resolved in 2cfe5bd.

Incorrect opcode whitelisted LOW OS-BAM-ADV-03

Description

In `setLzConfig`, the wrong opcode is whitelisted. Indeed, `Endpoint::OP::SET_EP_CONFIG_OAPP` is only callable by the controller.

Instead, the OApp should allow sending a message with the `Controller::OP::SET_EP_CONFIG_OAPP` opcode to the controller which will forward it to the endpoint. This issue prevents the configuration of custom settings for its channel.

>_ ethenaoft-ton/src/oApp/handlerOApp.fc FUNC

```
tuple setLzConfig(cell $Config) impure inline method_id {
    [...]
    (
        cell $SendPath,
        int forwardingAddress,
        int opCode,
        cell $innerConfig
    ) = lz::Config::deserialize($Config);

    _assertSendPath($SendPath);

    ifnot (
        (opCode == Endpoint::OP::SET_EP_CONFIG_OAPP) |
        (opCode == MsglibManager::OP::SET_OAPP_MSGLIB_SEND_CONFIG) |
        (opCode == MsglibManager::OP::SET_OAPP_MSGLIB_RECEIVE_CONFIG)
    ) {
        throw(ERROR::InvalidLzConfigOpCode);
    }
    [...]
}
```

Remediation

Replace `Endpoint::OP::SET_EP_CONFIG_OAPP` by `Controller::OP::SET_EP_CONFIG_OAPP`.

Patch

Resolved in 121c5c0.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-BAM-SUG-00	Recommendation to mark `handlerEthena::receiveOFT` as `impure` to ensure adherence to coding best practices.
OS-BAM-SUG-01	There is an error in the comment calculating the size of the storage in `jetton-minter::save_data`.
OS-BAM-SUG-02	Suggestion regarding a deserialization inconsistency in `transferOwnership`.
OS-BAM-SUG-03	The `ZRO` fee in `sendOFT` is not validated, potentially allowing misuse, especially if non-standard messaging libraries replace the default hardcoded fee of zero.

Missing Impure Specifier

OS-BAM-SUG-00

Description

`handlerEthena::receiveOFT` is missing the `impure` specifier. Although the compiler will not optimize `receiveOFT` because its return value is utilized, omitting `impure` deviates from the codebase's convention of marking all functions with side effects as `impure`. Furthermore, the functions utilized in `receiveOFT` are marked as `impure`.

>_ *ethenaoft-ton/src/ethenaOFT/handlerEthena.fc* FUNC

```
(cell, tuple) receiveOFT(cell $storage, tuple actions, cell sendOftEncoded, int remoteEid) {
    (int to, int amount) = _decodeSendOFT(sendOftEncoded);
    cell $RateLimits = $storage.EthenaOFT::getRateLimits();
    (cell $RateLimitBucket, int exists) = $RateLimits.cl::dict256::get<cellRef>(remoteEid);
    if (exists) {
        $storage = $storage.EthenaOFT::setRateLimit(
            remoteEid,
            RateLimitBucket::refill($RateLimitBucket, amount)
        );
    }

    actions = _sendTokens($storage, actions, to, amount);
    return ($storage, actions);
}
```

Remediation

Ensure to mark `handlerEthena::receiveOFT` as `impure`.

Patch

Resolved in 9174f78.

Presence of Erroneous Comment OS-BAM-SUG-01

Description

The comment in `jetton-minter::save_data` incorrectly states that `store_coins` always requires 128 bits to store the `total_supply`, implying a fixed space utilization regardless of the value's magnitude. However, coins are stored as variable-length integers, utilizing between 4 and 124 bits. While this inconsistency does not impact the contract because the actual data size is smaller and still fits within a cell, the comment is misleading and should be corrected for clarity.

>_ *ethenaoft-ton/src/token/jetton-minter.fc* FUNC

```
() save_data([...]) impure inline {
    set_data(
        begin_cell()
        .store_coins(total_supply) ;; 128
        .store_slice(admin_address) ;; + 267 = 395
        .store_slice(next_admin_address) ;; + 267 = 662
        .store_slice(mint_to_authority) ;; + 267 = 929
        .store_ref(jetton_wallet_code)
        .store_ref(metadata_uri)
        .end_cell()
    );
}
```

Remediation

Update the comment to reflect the correct description.

Patch

Resolved in 2cfe5bd.

Deserialization Discrepancy OS-BAM-SUG-02

Description

`transferOwnership` in `ethenaoft` stores the new owner's address at a fixed offset utilizing `$Address.Address::getAddress()`. However, when emitting the `TentativeOwnerSet` event, the new owner's address is retrieved via `$Address.Address::sanitize()`, which depends on `cl::get<address>` to parse the cell based on its type information. This discrepancy may result in the address stored as the new tentative owner differing from the address recorded in the event. Utilize the correct method to extract the address.

>_ *ethenaoft-ton/src/oApp/handlerOApp.fc* FUNC

```
tuple transferOwnership(cell $Address) impure inline {
    (cell $storage, tuple actions) = preamble();
    setContractStorage(
        setOAppStorage(
            $storage,
            getOAppStorage().cl::set(
                OApp::tentativeOwner,
                $Address.Address::getAddress()
            )
        )
    );
    actions~pushAction<event>(
        EVENT::TentativeOwnerSet,
        $Address.Address::sanitize()
    );

    return actions;
}
```

Remediation

Implement the above-mentioned suggestion.

Patch

Resolved in 9174f78.

Failure to Validate ZRO Fee

OS-BAM-SUG-03

Description

`sendOFT` lacks validation for the `zroFee` parameter, which is paid by the `OApp` rather than the end user. The `zroFee` value is directly extracted from the `$OFTSend.OFTSend::getLzSendInfo` call without validation. Currently, `_quoteWorkers` in `ULN` hardcodes the `zroFee` to zero, mitigating the issue within the existing implementation. However, this approach creates a dependency on the current `msglib`. If the system transitions to a `msglib` where `zroFee` is dynamically set, the absence of validation will pose a significant risk.

Remediation

Ensure the `zroFee` is explicitly validated within `sendOFT` before it is passed to `_lzSend`.

Patch

The LayerZero team has acknowledged this issue but has opted not to address it, citing no current plans to implement `zro` payments in `msglib`. Should `zro` payments be enabled in the future, the contracts will be upgraded to support this functionality.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.