Zellic

Prepared for
Ryan Zarick
Isaac Zhang
LayerZero Labs

Prepared by
Aaron Esau
Nan Wang
Zellic

January 15, 2025

USDT TON
TON Application Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for LayerZero Labs from January 2nd to January 15th, 2025. During this engagement, Zellic reviewed USDT TON'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 erroneous operations lead to long-term service degradation or DOS, and are there measures to mitigate such risks?
- While the planner can only adjust credits, could this indirectly affect user funds?
- Can only authorized administrators modify critical parameters (such as peer address)?
- Do the functions have sufficient permission checks to prevent unauthorized calls?
- Is the 24-hour timelock for USDT withdrawals strictly enforced, and do expired requests indeed become invalid?
- In the event of abnormalities, are there administrator tools or event logs to facilitate investigation and fund recovery?

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 USDT TON contracts, we discovered one finding, which was informational in nature.

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

Breakdown of Finding Impacts

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

2. Introduction

2.1. About USDT TON

LayerZero Labs contributed the following description of USDT TON:

> LayerZero is a technology that enables applications to move data across blockchains, uniquely supporting censorship-resistant messages and permissionless development through immutable smart contracts.

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:

USDT TON Contracts

Type	FunC
Platform	TON

Target	LayerZero TON USDT OFT
Repository	https://github.com/LayerZero-Labs/i-devtools-ton ↗
Version	`c3c46856fd24c52f7a3ccd5f7f28d991735e200d`

Programs	`structs/Address.fc`
	`structs/CostAsserts.fc`
	`structs/Fee.fc`
	`structs/GasAsserts.fc`
	`structs/LayerZeroStructs.fc`
	`structs/MdError.fc`
	`structs/OFTCredits.fc`
	`structs/OFTSend.fc`
	`structs/RecoverUsdt.fc`
	`structs/SetPeer.fc`
	`structs/TokenTransfer.fc`
	`src/handler.fc`
	`src/main.fc`
	`src/oApp/handlerOApp.fc`
	`src/oApp/interface.fc`
	`src/oApp/storage.fc`
	`src/usdtOFT/consts.fc`
	`src/usdtOFT/handlerUsdt.fc`
	`src/usdtOFT/interface.fc`
	`src/usdtOFT/storage.fc`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 12 person-days. The assessment was conducted by two consultants over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Aaron Esau
Engineer
aaron@zellic.io ↗

Nan Wang
Engineer
nan@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 2, 2025	Start of primary review period
January 8, 2025	Kick-off call
January 13, 2025	Scope changed from `bc8c59ca` ↗ to `0c7b038f` ↗
January 15, 2025	End of primary review period
January 21, 2025	Scope changed from `0c7b038f` ↗ to `c3f8e990` ↗
February 11, 2025	Scope changed from `c3f8e990` ↗ to `c3c46856` ↗

3. Detailed Findings

3.1. Owner's ability to bypass USDT recovery timelock

Target	handlerUsdt.fc		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The owner may use the `RecoverUsdtInit` and `RecoverUsdt` opcodes to begin the recovery process of USDT tokens (to send arbitrary amounts) and to recover tokens after the timelock finalizes, respectively.

However, the owner also has the ability to send USDT tokens arbitrarily because of the OApp configuration opcode `SetPeer` (among others). When receiving a packet from a remote peer, the `_assertReceivePath` function (and internally `_assertUnpackedPath`) ensures that the packet is from a trusted peer, as determined by the owner-configured peer mapping:

```
() _assertUnpackedPath(int srcEid, int srcOApp, int dstEid, int dstOApp)
    impure inline {
    cell $OAppStorage = getOAppStorage();

    ;; make sure peer exists
    (int peerAddress, int peerExists) = $OAppStorage
        .OApp::getPeers()
        .cl::dict256::get<uint256>(dstEid);

    ;; throw if any part of the path is incorrect
    throw_unless(ERROR::WrongSrcEid, srcEid == $OAppStorage.OApp::getEid());
    throw_unless(ERROR::PeerNotSet, peerExists);
    throw_unless(ERROR::WrongSrcOApp, srcOApp == getContractAddress());
    throw_unless(ERROR::WrongPeer, peerAddress == dstOApp);
}
```

The owner could simply change the peer address to a malicious EID and OApp configuration to send an arbitrary withdrawal packet (`receiveOFT`), thereby bypassing the USDT recovery timelock. Additionally, the owner could falsify a credit-receiving packet (`receiveCredits`) and then use the `OP::WithdrawLocal` or `OP::WithdrawRemote` opcodes to withdraw using the false liquidity.

Other LayerZero configuration functions may enable the same bypass (e.g., configuring DVNs maliciously) as well.

Impact

The owner can immediately bypass the USDT recovery timelock to send arbitrary amounts of USDT tokens (limited by the `contractBalance`, but not constrained by any credit balance.)

The severity is limited because the owner has the ability to change the planner address. An owner could send credits from other chains and then use the `OP::WithdrawLocal` or `OP::WithdrawRemote` opcodes.

As a side note, consider checking whether the `admin_address` of the USDT minter contract (out of scope) is controlled by the same entity as the owner. If so, then the owner can directly mint, defeating the purpose of these credit checks in `withdrawLocal`, `withdrawRemote`, and the timelock on the recovery functions in the first place.

Recommendations

Note that the owner's ability to configure the OApp is a feature intended to support future LayerZero upgrades, including security fixes and new features as technology advances. However, the ability to configure the OApp introduces centralization risk. It is up to LayerZero Labs to determine what risk is acceptable.

If LayerZero Labs determines that the ability for the owner to send USDT arbitrarily without a timelock is unacceptable, the configuration functions should include the same timelock requirement as the USDT recovery process. Otherwise, consider removing the timelock altogether for simplicity and to avoid a false reassurance about the centralization risks.

Remediation

This issue has been acknowledged by LayerZero Labs.

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. Fee bypass for small amounts

Note that there is a fee bypass in `sendOFT` that LayerZero Labs is aware of:

```
;; The rounding on the fee and amount calculation favors the sender,
;; In theory small enough transfers will not incur any fees
(int, int) _getAmountAndFee(cell $storage, int totalAmount) impure inline {
    int feeBps = $storage.UsdtOFT::getFeeBps();
    ;; totalAmount is always at most 128-bits, feeBps is at most 10000 == 13
    bits
    ;; so this multiplication can never overflow signed 257-bit
    int fee = ((totalAmount * feeBps) / BPS_DENOMINATOR);
    int amount = totalAmount - fee;
    return (amount, fee);
}
```

However, given the `BPS_DENOMINATOR` of `10000`, exploiting this bug to completely bypass fees does not make sense in practice; the maximum transferrable amount while bypassing fees is `ceil(BPS_DENOMINATOR / feeBps) - 1`, which is a very small amount compared to fees.

4.2. Importance of planner's credit management

The planner could DoS the USDT OFT users, but cannot steal or otherwise compromise any USDT.

The system relies on credits, which the planner is authorized to move across different chains. If the planner moves credits in a way such that a chain is left without any credits for other chains, then no OFTs may be transferred from that chain anymore.

This is an issue that the LayerZero Labs is already aware of.

5. Threat Model

This provides a threat model description of the assessed smart contracts. As time permitted, we analyzed each entity's capability in the contracts and created a written threat model. A threat model documents externally controllable inputs and how an attacker could leverage each input or capability to cause harm.

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

5.1. usdtOFT

The capabilities of each entity are as follows.

Permissionless

There are no assertions for the caller of the `Layerzero::OP::LZ_RECEIVE_PREPARE` opcode. A description of the function of the opcode is best described by the following comment in handlerOApp.fc:

```
;; lzReceive follows a two-phase commit pattern in TON
;; first, a permissionless call is made to lzReceivePrepare
;;      failure => permissionless retry at endpoint
;; second, the OApp performs a gas assertion and sends a clear request to the
    Packet
;;      failure => permissionless retry from step 1 at endpoint
;; third, the Packet locks itself and notifies the OApp to execute the request
;;      failure => OApp must unlock the Packet and retry from step 1
;; finally, the OApp clears and destroys the message
;;      failure => can be ignored, but the OApp will be blocklisted by the
    executor
```

The opcode ensures the remote peer in the user-controlled packet is correct, asserts that there is enough gas remaining to follow the receive flow, and calls `Channel::OP::LZ_RECEIVE_LOCK` on the caller — which is intended to be the channel contract (in LayerZero TON). If the caller is not the channel, there is no effect of the call.

Because the opcode makes no state changes, there is no harm in it being permissionless.

LayerZero channel

The LayerZero channel contract has the ability to call the following opcodes:

- `Layerzero::OP::LZ_RECEIVE_EXECUTE`
- `Layerzero::OP::CHANNEL_SEND_CALLBACK`

- `Layerzero::OP::BURN_CALLBACK`
- `Layerzero::OP::NILIFY_CALLBACK`

For information on the threat model of LayerZero, please refer to our LayerZero Endpoint V2 TON report (unreleased as of the time of this writing).

Planner

The planner has the ability to call the `OP::SendCredits` and `OP::SetCostAsserts` opcodes. This role typically handles cross-chain liquidity management by moving credits between different chains and updating the cost-related parameters used for gas and price checks.

- **`OP::SendCredits`.** The planner can invoke this opcode to transfer credits (liquidity) across supported chains. While the planner does not hold the authority to withdraw actual USDT to an external address (that remains under the owner's authority), it can reorder and rebalance the distribution of credits among chains. In a worst-case scenario where the planner's key is compromised or the planner acts maliciously, it could excessively drain or redirect credits from one chain to another, resulting in local liquidity shortages or disruptions for cross-chain operations.
- **`OP::SetCostAsserts`.** The planner can also modify cost-related assertions; raising these values too high may lead to unjustified transaction failures, whereas setting them too low could allow underpayment of fees and potentially disrupt cross-chain guarantees. When setting cost asserts, the user-input object is sanitized to ensure that only valid integer and structured fields are stored, preventing injection of malformed data structures.

Overall, the planner's abilities facilitate dynamic adjustments to cross-chain liquidity and cost configurations. However, because the planner cannot directly withdraw USDT to an external address, there is a baseline safeguard against total liquidity theft. The ultimate power to withdraw tokens or reassign the planner resides with the owner, thereby mitigating full control risks associated with the planner role.

Owner

The owner has the ability to call the following opcodes:

- **`OP::WithdrawLocal`.** The owner can withdraw USDT from the contract balance provided there are enough credits for the local EID (i.e., if `depositLocal` is called or sending bounces).

- **`OP::WithdrawRemote`.** The owner can send USDT to an arbitrary remote address, limited by the credits for that remote EID.

- **`OP::WithdrawFees`.** The owner can withdraw up to the `feeBalance` balance.

- **`OP::RecoverUsdtInit`** and **`OP::RecoverUsdt`.** The owner can arbitrarily withdraw USDT from the contract balance after a timelock period. Note that we reported a bypass for this in Finding 3.1. ↗.

- **`OP::SetPlannerAddress`.** Because of this, the owner inherits all of the abilities of the planner.
- **`OP::SetGasAsserts`.** The user-input object is sanitized. This opcode allows the owner to configure the gas assertions for sending and receiving specific types of messages.
- **`OP::SetFee`.** The user-input object is sanitized. This allows the owner to set an arbitrary fee at any time, provided it is an unsigned integer less than 100%.

The following opcodes are for managing the LayerZero OApp (handled in handlerOApp.fc). They are not specific to USDT TON, and all functions are typically included in all OApp implementations on TON.

- **`OP::DeployChannel`.** This deploys a LayerZero channel contract. This is typically done when intending to support a new path.
- **`OP::DeployConnection`.** This instructs the messaging library to deploy a new, sharded LayerZero UlnConnection contract (`deployUlnConnection` in ulnManager) or a new SmlConnection contract (`deployConnection` in smlManager). This is typically done when intending to support a new path.
- **`OP::ForceAbort`.** This aborts a pending send request. This capability is intended to prevent the send queue from being blocked if the `sendRequestQueue` (a `DeterministicInsertionCircularQueue` structure in LayerZero's channel contract) fills up for any reason. Note that when send requests are aborted, the path will not be refunded credits, and the sender will not be refunded fees or USDT on the local chain.
- **`OP::Burn`.** This permanently discards an inbound message that the local OApp no longer intends to execute. Under the hood, the channel marks the message as "executed" (so it cannot be triggered again), effectively burning away that nonce from future processing. This is useful when a message is deemed obsolete or invalid, ensuring it never reenters the execution flow.
- **`OP::Nilify`.** This removes an inbound message from the channel's queue without marking it as executed. This effectively frees up the message slot so it no longer blocks subsequent messages, but does not treat it as having been processed. It is helpful in scenarios where the message must be cleared out (e.g., it became irrelevant) but not formally "executed" in a business-logic sense.
- **`OP::SetPeer`.** This opcode configures valid outbound and inbound peer addresses for cross-chain interactions given an EID.
  - **Outbound**: For each `eid`, the contract uses an internally maintained "`eid → peerAddress`" mapping when sending cross-chain messages, eliminating the need for hardcoded addresses and simplifying updates.
  - **Inbound**: Upon receiving a message, the contract checks whether the sender matches the locally stored `peerAddress`. If they differ, the message is deemed invalid and rejected.

  This source verification is crucial for ensuring that cross-chain security-only messages from the recognized `peerAddress` are considered legitimate.
- **`OP::SetEnforcedOptions`.** The `setEnforcedOptions` function defines mandatory option

configurations for cross-chain messages associated with a specific `(msgType,dstEid)`. These enforced options can restrict particular aspects of message handling — for example, requiring a specific options version or disallowing certain advanced features. By enforcing these settings at the OApp contract level, any subsequent logic handling that message type must comply with these requirements, thereby ensuring that the processing flow aligns with the intended security and functional policies.

- **`OP::SetLzConfig`.** The owner may modify the endpoint or messaging-library configuration at any time. A full description of the centralization risks of this ability is documented in our LayerZero Endpoint V2 ↗ (EVM) report.

Tentative owner

The tentative owner may accept ownership by using the `OP::ClaimOwnership` opcode. However, the owner must first call `transferOwnership` with the tentative owner's address.

6. Assessment Results

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

During our assessment on the scoped USDT TON contracts, we discovered one finding, which 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.