Zellic

Prepared for
Henryk Sarat
Paxos

Prepared by
Katerina Belotskaia
Jaeeu Kim
Zellic

October 21, 2024

Cross Chain Contracts

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 Paxos from October 14th to October 18th, 2024. During this engagement, Zellic reviewed Cross Chain Contracts'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 the storage layout compatible after the upgrade?
- Is Paxos's smart contract upgrade process safe?
- Could LayerZero somehow break minting logic of a Paxos stablecoin?
- Are access controls implemented effectively to prevent unauthorized operations?
- Could an on-chain attacker exploit the system for unbounded minting?

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 Cross Chain Contracts contracts, we discovered one finding, which was of high impact.

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

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Cross Chain Contracts

Paxos contributed the following description of Cross Chain Contracts:

> OFTWrapper serves as a proxy for LayerZero's OFT standard, allowing LayerZero to mint and burn tokens as usual, which includes the same AML standards as a same chain transfer. When LayerZero initiates a mint or burn request, the OFTWrapper forwards it to the underlying token contract. For this to work, the underlying token must authorize OFTWrapper to perform minting and burning operations on its behalf.

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:

Cross Chain Contracts Contracts

Type	Solidity
Platform	EVM-compatible

Target	cross-chain-contracts
Repository	https://github.com/paxosglobal/cross-chain-contracts ↗
Version	`168b4c350ce10c3aa6ae18b55a1339bd8c5afd21`

Programs	`OFTWrapper`
	`LzEndpointFixture`
	`OFTWrapperFixture`
	`PaxosTokenFixture`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of one and a half person-weeks. 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 ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Jaeeu Kim
Engineer
jaeeu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 14, 2024	Kick-off call
October 14, 2024	Start of primary review period
October 18, 2024	End of primary review period

3. Detailed Findings

3.1. OFTWrapper is not compatible with the USDP token

Target	OFTWrapper, USDP		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The OFTWrapper is used to wrap the Paxos stable tokens to OFT (Omnichain Fungible Token). To initialize the OFTWrapper, the OFTCore contract is used. The OFTCore contract requires the `_localDecimals` parameter to calculate the `decimalConversionRate`, which is used to convert the token amount to the OFT amount.

```
contract OFTWrapper is OFTCore {
    //The Paxos token
    PaxosTokenV2 private immutable paxosToken;
    // ...
    constructor(
        address _paxosToken,
        address _lzEndpoint,
        address _delegate
    ) OFTCore(6, _lzEndpoint, _delegate) {
        paxosToken = PaxosTokenV2(_paxosToken);
    }
```

```
abstract contract OFTCore is IOFT, OApp, OAppPreCrimeSimulator,
    OAppOptionsType3 {
    // ...
    uint256 public immutable decimalConversionRate;

    constructor(uint8 _localDecimals, address _endpoint, address _delegate)
    OApp(_endpoint, _delegate) {
        if (_localDecimals < sharedDecimals()) revert InvalidLocalDecimals();
        decimalConversionRate = 10 ** (_localDecimals - sharedDecimals());
    }
```

The `_localDecimals` parameter is hardcoded to 6 in the OFTWrapper contract. This will be fine with PYUSD and USDG tokens, which have six decimals. However, this is not compatible with the USDP token, which has 18 decimals. This will cause the OFTWrapper to convert the token amount incorrectly.

```
// USDP.sol
function decimals() public view virtual override returns (uint8) {
        return 18;
}
```

Impact

For instance, the default `sharedDecimals` for OFT is set to 6, while the USDP token uses 18 decimals. However, due to the hardcoded decimal values, the `decimalConversionRate` will be incorrectly calculated as `10 ** (6 - 6) = 1`, not `10 ** (18 - 6) = 10 ** 12`. This will cause the OFTWrapper to convert the token amount incorrectly.

Especially when sending a value larger than 2^{64} (approximately 18.4 USDP), the function `_toSD`, which is used to convert the token amount to the OFT amount, will cast the value to `uint64`, and this will result in the loss of tokens.

```
function _toSD(uint256 _amountLD)
    internal view virtual returns (uint64 amountSD) {
    return uint64(_amountLD / decimalConversionRate);
}
```

Recommendations

Set the `_localDecimals` to the provided token's decimals.

```
contract OFTWrapper is OFTCore {
    //The Paxos token
    PaxosTokenV2 private immutable paxosToken;
    // ...
    constructor(
        address _paxosToken,
        address _lzEndpoint,
        address _delegate
    ) OFTCore(ERC20(_paxosToken).decimals(), _lzEndpoint, _delegate) {
        paxosToken = PaxosTokenV2(_paxosToken);
    }
```

Remediation

This issue has been acknowledged by Paxos, and a fix was implemented in commit 965f7a40 ↗.

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. Centralization risks

This section documents the roles defined in Cross Chain Contracts and highlights the potential risks associated with them according to the contract and the specifications:

OFTWrapper

- `Owner`
    - **Privileges:** Setting the rate limits for cross-chain message sending and setting the trusted peer address who sends messages from the other chains. Can update the delegate address.
    - **Restrictions:** Has no restrictions.
    - **Key-custody solution:** Assigned to a multi-sig wallet.
- `_delegate`
    - **Privileges:** Setting and updating cross-chain communication configs for the current contract inside the Endpoint contract.
    - **Restrictions:** Has no restrictions.
    - **Key-custody solution:** Assigned to a multi-sig wallet.

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. OFTWrapper

The OFTWrapper contract inherits the OFT Standard ↗. The OFT Standard is a contract that allows to transfer, across multiple chains, fungible tokens using the LayerZero protocol.

During the deployment of the OFTWrapper contract, the underlying token is set, which is the address of the ERC-20 Paxos token. This token address is stored in an immutable variable and cannot be changed in the future. As a result, the OFTWrapper contract is permanently associated with a specific Paxos token. Additionally, the OFTWrapper contract is not upgradable.

The trusted LayerZero (LZ) endpoint address and the delegate address are also assigned. The delegate responsible for implementing custom configurations for this contract is on the LZ protocol side. Furthermore, the rate-limit configuration is set, which controls how often cross-chain token transfers can be executed.

User-facing interface

The main functionality available to the user in this contract is the `send` function.

The `send` function allows users who own the underlying tokens to transfer them to the remote OFT contract on a supported destination chain. The caller specifies the sending parameters in the `_sendParam` structure, which contains the following data:

- `dstEid`, the destination ID supported by LZ
- `to`, the recipient address in `bytes32`
- `amountLD`, the amount of tokens to send
- `minAmountLD`, the minimum amount to send
- `extraOptions`, the additional setting, which can be an empty byte
- `composeMsg`, the composed message
- `oftCmd`, the OFT command to be executed (currently unused in this implementation)

Additionally, the caller provides the `_refundAddress`, which receives any unused fees.

The `send` function follows these steps:

1. Call the internal `_debit` function, which handles the transferred tokens.
2. Build the provided data into the `message` for transfer.

3. Interact with the endpoint to send the message via the `_lzSend` function.
4. Emit the `OFTSent` event.

```
function send(
    SendParam calldata _sendParam,
    MessagingFee calldata _fee,
    address _refundAddress
) external payable virtual returns (MessagingReceipt memory msgReceipt,
    OFTReceipt memory oftReceipt) {
    (uint256 amountSentLD, uint256 amountReceivedLD) = _debit(
        msg.sender,
        _sendParam.amountLD,
        _sendParam.minAmountLD,
        _sendParam.dstEid
    );

    (bytes memory message, bytes memory options)
    = _buildMsgAndOptions(_sendParam, amountReceivedLD);

    msgReceipt = _lzSend(_sendParam.dstEid, message, options, _fee,
    _refundAddress);
    oftReceipt = OFTReceipt(amountSentLD, amountReceivedLD);

    emit OFTSent(msgReceipt.guid, _sendParam.dstEid, msg.sender, amountSentLD,
    amountReceivedLD);
}
```

The `send` function is overridden in the OFTWrapper to implement custom logic. First, the transferred amount is adjusted to match the decimal numbers of the target token in the destination chain. Second, the rate limit is verified and updated. Finally, interaction with the underlying tokens is performed to execute all necessary logic for transferring the tokens.

```
function _debit(
    address _from,
    uint256 _amountLD,
    uint256 _minAmountLD,
    uint32 _dstEid
)
    internal
    override
    returns (uint256 amountSentLD, uint256 amountReceivedLD)
{
    (amountSentLD, amountReceivedLD) = _debitView(
        _amountLD,
        _minAmountLD,
        _dstEid
    );
    _checkAndUpdateRateLimit(_dstEid, amountSentLD);
    paxosToken.decreaseSupplyFromAddress(amountSentLD, _from);
}
```

LayerZero-facing interface

The main OFTWrapper function exposed to the LZ endpoint is `lzReceive`. This function can only be triggered by the `endpoint` address, which is immutable and cannot be changed. Access control for this function also includes verifying the sender's address from the trusted source chain ID.

```
function lzReceive(
    Origin calldata _origin,
    bytes32 _guid,
    bytes calldata _message,
    address _executor,
    bytes calldata _extraData
) public payable virtual {
    if (address(endpoint) != msg.sender) revert OnlyEndpoint(msg.sender);
    if (_getPeerOrRevert(_origin.srcEid) != _origin.sender)
    revert OnlyPeer(_origin.srcEid, _origin.sender);
    _lzReceive(_origin, _guid, _message, _executor, _extraData);
}
```

The process of receiving tokens is mainly handled by the `_credit` function. This function, along with the `_debit` function, is overridden in the OFTWrapper contract and only performs external calls to the `increaseSupplyToAddress` function of the underlying token contract. A crucial point is that if the execution of the `lzReceive` function is reverted for any reason, it can be executed again to retry receiving the tokens until the process is successfully completed or until the `delegate` clears the message.

Optionally, if the sender specified the composed message, it will be saved inside the endpoint contact and can be delivered to the receiver using `endpoint.lzCompose()`, which should be called separately and is not part of the current token-receiving process. The `endpoint.sendCompose` function only saves the hash of this composed message.

```
function _lzReceive(
    Origin calldata _origin,
    bytes32 _guid,
    bytes calldata _message,
    address /*_executor*/, // @dev unused in the default implementation.
    bytes calldata /*_extraData*/ // @dev unused in the default implementation.
) internal virtual override {
    address toAddress = _message.sendTo().bytes32ToAddress();
    uint256 amountReceivedLD = _credit(toAddress, _toLD(_message.amountSD()),
    _origin.srcEid);

    if (_message.isComposed()) {
        bytes memory composeMsg = OFTComposeMsgCodec.encode(
            _origin.nonce,
            _origin.srcEid,
            amountReceivedLD,
            _message.composeMsg()
        );

        endpoint.sendCompose(toAddress, _guid, 0 /* the index of the composed
    message*/, composeMsg);
    }

    emit OFTReceived(_guid, _origin.srcEid, toAddress, amountReceivedLD);
}

function _credit(
    address _to,
    uint256 _amountLD,
    uint32 /*_srcEid*/
) internal override returns (uint256 amountReceivedLD) {
    paxosToken.increaseSupplyToAddress(_amountLD, _to);
    return _amountLD;
}
```

Owner-facing interface

The owner can update the previously set delegate address using the `setDelegate` function.

The delegate, in addition to the OApp contract itself, has the ability to manage cross-chain transfer configurations using the following functions:

- `setSendLibrary`
- `setReceiveLibrary`
- `setReceiveLibraryTimeout`
- `setConfig`

Additionally, the delegate has access to critical functionality, such as skipping the next nonce using the `skip` function to prevent message verification and clearing the received message payload using the `clear` function. It is especially important to use these functions carefully, because once the message payload is deleted, it cannot be recovered.

The `nilify` and `burn` functions are also available to the `delegate`, helping to mitigate malicious acts by DVNs. For a more detailed description of these functions, refer to the LZ documentation ↗.

It is important to note that the owner of the OFTWrapper contract does not have access to the end-point configuration, unless the owner's address is directly assigned as the `delegate`.

Additionally, the functionality available only to the owner account includes the `setRateLimits` function, which updates the rate-limit configuration for cross-chain message transfers, and the `setPeer` function, which sets the trusted sender associated with the specified source endpoint. The rate limit applies to the destination endpoint ID and ensures that the amount of tokens sent to another chain falls within the rate-limit constraints. If the amount exceeds the limit, the transaction will be reverted.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to Ethereum Mainnet.

During our assessment on the scoped Cross Chain Contracts contracts, we discovered one finding, which was of high impact.

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.