LayerZero Stargate

Smart Contract Security Assessment

March 6, 2022

Prepared for:

Ryan Zarick

LayerZero Labs

Prepared by:

Jasraj Bedi and Stephen Tong

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than to simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible.

To keep up with our latest endeavors and research, check out our website https://zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1. Introduction

1.1. About LayerZero Stargate

Stargate is a cross-chain token swapping bridge implemented by leveraging the LayerZero networks' arbitrary cross-chain messaging protocol. They aim to provide a unified interface for bridging tokens, along with cross-chain liquidity provisioning and instant finality guarantee of swaps.

The heart of stargate is the delta algorithm, keeping track of the token transfers and deposits to maintain a balanced liquidity pool for multiple chains. The liquidity is divided among the chains using a set of weights that can be adjusted, allowing liquidity to be concentrated among the high volume chains. Incentivization mechanisms also exist in case the liquidity is running low on some chain. For example, users are rewarded extra tokens when swapping from a chain with low liquidity.

1.2. 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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these "shallow" bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, etc. 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions, and summarize the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines, or code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, etc.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): *Critical*, *High*, *Medium*, *Low*, and *Informational*.

Similarly, Zellic organizes its reports such that the most *important* findings come first in the document, rather than impact alone. Thus, we may sometimes emphasize a "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 numerous soft factors, such as our clients' threat model, their business needs, project timelines, etc. We aim to provide useful and actionable advice to our clients that consider their long-term goals, rather than simply a list of security issues at present.

1.3. Scope

The engagement involved a review of the following targets:

Stargate

Repository	https://github.com/stargate-protocol/stargate
Versions	`095bd19195aacbcbdabc606e95b85f4695d3d614`
Type	Solidity
Platform	Ethereum cross-chain; any EVM compatible chain

1.4. Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered as financial or investment advice.

2. Executive Summary

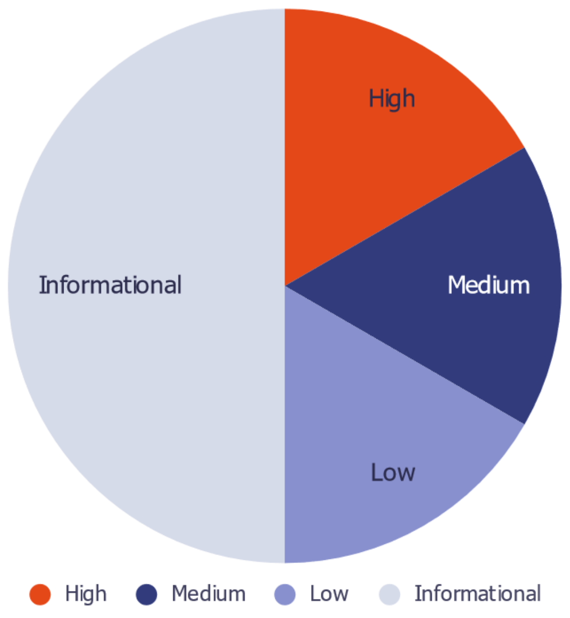
Zellic conducted an audit for LayerZero from 21st February to 4th March, 2022 on the scoped contracts and discovered 7 findings. **Fortunately, no critical issues were found.** We applaud LayerZero for their attention to detail and diligence in maintaining high code quality standards. Of the 7 findings, 1 was of high impact, and 1 was of medium impact, 1 was of low impact. The remaining findings were informational in nature.

Stargate is a cross-chain token swap bridge with instant finality guarantee (IFG). A serious bug in its internal accounting could be critical as it might break the IFG. Thus, for this audit, we focused heavily on accounting errors in addition to the normal suite of cross-chain specific issues.

Our general overview of the code is that it was very well structured. The code coverage is high and tests are included for the majority of the functionality. The documentation was adequate, although it can be improved. We would recommend LayerZero to clearly document every function argument, as it was a huge source of confusion on our side when reading the contract initially.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	1
Informational	4



3. Detailed Findings

3.1. Possible cross-chain desynchronization of token balances

- **Target:** Pool
- **Severity:** High
- **Impact:** High
- **Category:** Business Logic
- **Likelihood:** High

Description

The `lkbRemove` values in cross-chain swap (swap -> swapRemote) payload failed to account for the `eqReward`, potentially leading to desynchronization of token balances that can be swapped between two chains. In swapRemote, the amount transferred to the user includes `s.eqReward`, while not being reduced from the `cp.balance` on the source chain's side.

```
function swap(uint16 _dstChainId, uint256 _dstPoolId, address _from, uint256
_amountLD, uint256 _minAmountLD, bool newLiquidity) external nonReentrant
onlyRouter returns (SwapObj memory) {

...

uint256 amountSD = amountLDtoSD(_amountLD);

...

SwapObj memory s = IStargateFeeLibrary(feeLibrary).getFees(poolId, _dstPoolId,
_dstChainId, _from, amountSD);

...

// behaviours
//     - protocolFee: booked, stayed and withdrawn at remote.
//     - eqFee: booked, stayed and withdrawn at remote.
//     - lpFee: booked and stayed at remote, can be withdrawn anywhere

s.lkbRemove = amountSD.sub(s.lpFee); <-- eqReward should be included in lkbRemove
too!
```

Impact

The desynchronization of cross-chain balances could possibly lead to a case where the swap may not go through even though the source chain believes it would, breaking the IFG (Instant finality guarantee) and leading to user funds being permanently locked.

We created a proof of concept for this bug that breaks the IFG. An excerpt of the output is shown below:

```
--------------INITIAL STATE--------------

Balance on chain A -> 10000000000000
Balance on chain B -> 10000000000000
Balance on chain C -> 10000000000000

-----------------------------------------

Performing first swap : 2500 from C to B (1% eqFee fee)

------------------STATE------------------

Balance on chain A -> 10000000000000
Balance on chain B -> 12475000000000     // B deposited with 2475 (1% fee)
Balance on chain C ->  7500000000000     // C reduced by 2500

-----------------------------------------

Performing second swap : 2500 from C to A (0% fee)

------------------STATE------------------

Balance on chain A -> 12500000000000     // +2500
Balance on chain B -> 12475000000000
Balance on chain C ->  5000000000000     // -2500

-----------------------------------------

Performing third swap : 2500 from B to A (1% eqReward)

------------------STATE------------------

Balance on chain A -> 12500000000000     // +0: nothing swapped out!
Balance on chain B ->  9975000000000     // -2500
Balance on chain C ->  5000000000000

-----------------------------------------
```

We have provided the PoC to LayerZero for reproduction and remediation.

Recommendations

Account for `s.eqReward` while calculating `s.lkbRemove`:

```
s.lkbRemove = amountSD.sub(s.lpFee).add(s.eqReward);
```

Remediation

The issue has been acknowledged and fixed by LayerZero. It was also found during an internal audit of the code.

3.2. Insufficient equilibrium fee pool can cause swaps to fail

- **Target:** Pool
- **Severity:** Medium
- **Impact:** Medium
- **Category:** Business Logic
- **Likelihood:** Medium

Description

In the function `swap`, s.eqReward is subtracted from `eqFeePool` using SafeMath, which will revert if `s.eqReward` is larger than eqFeePool. This can happen either when eqFeePool is very low or a large amount of tokens are being swapped out.

```
require(!stopSwap, "Stargate: swap func stopped");
ChainPath storage cp = getAndCheckCP(_dstChainId, _dstPoolId);
require(cp.ready == true, "Stargate: counter chainPath is not ready");

uint256 amountSD = amountLDtoSD(_amountLD);
uint256 minAmountSD = amountLDtoSD(_minAmountLD);
// request fee params from library
SwapObj memory s = IStargateFeeLibrary(feeLibrary).getFees(poolId, _dstPoolId,
_dstChainId, _from, amountSD);

// equilibrium fee and reward. note eqFee/eqReward are separated from swap
liquidity
eqFeePool = eqFeePool.sub(s.eqReward); // This operation may underflow and revert!
```

Impact

Large swaps will fail, wasting users' gas (and money).

Recommendations

Compare `s.eqReward` to `s.eqFeePool` before subtracting and limit rewards up to the current `eqFeePool` to avoid arithmetic errors.

Remediation

The issue has been acknowledged by LayerZero. Their official response is reproduced below:

> *The equilibrium reward (eqReward) will also only be a fraction of the eqFeePool. This business logic constraint will be enforced in the feeLibrary.*

3.3. Unchecked use of mload can potentially lead to an out-of-bounds read

- **Target:** Multiple contracts
- **Severity:** Low
- **Impact:** Low
- **Category:** Code Maturity
- **Likelihood:** High

Description

A common pattern used in the codebase is to cast `bytes memory` into `address` using inline assembly, like so:

```
bytes memory _addr = ...;
address castedAddr;
assembly {
    castedAddr := mload(add(_addr, 20))
}
```

Casting `bytes memory _addr` to `address castedAddr`.

At the memory location pointed to by _addr, Solidity lays out 32 bytes storing _addr's size, followed by the contents of _addr. The first 20 bytes of the size field are discarded by the add instruction. The mload will load the last 12 bytes of the size field, followed by the first 20 bytes of the contents of _addr, assuming _addr is long enough. The 12 bytes of the size field are discarded when assigning to the destination variable toAddress.

If the byte array _addr is less than 20 bytes long, the mload will read out-of-bounds, returning bytes of undefined value.

We found this pattern used in Bridge.sol, OmnichainFungibleToken.sol, and Router.sol.

We believe this pattern was likely introduced as a gas optimization, but it is undocumented.

Impact

We audited all instances of the pattern and found that an attacker may be able to call the function with controlled contents for _addr. Chiefly, Router.redeemLocal is reachable by an attacker; and Bridge.lzReceive may be indirectly called through the LayerZero endpoint via Bridge.swap, via Router.swap.

Although we believe the bug currently has no security impact since it would simply lead to an invalid "to" address in each case, the potential for undefined behavior is concerning. In the past, even low-impact vulnerabilities have been chained with other bugs (2) to achieve critical security compromises.

Furthermore, the current implementation relies heavily on Solidity runtime implementation details, like the memory layout of bytes memory (32 bytes size field, followed by contents). In our experience, reliance on implementation details without adequate documentation is error-prone in general. Although StarGate has the Solidity version pinned at 0.7.6, this code may unexpectedly lead to future bugs which are challenging to detect from the code alone.

Recommendations

Add an explicit bounds check before each instance of the `mload(add(_x, 20))` pattern to ensure that the loaded buffer is at least 20 bytes long.

Remediation

The issue has been acknowledged by LayerZero. Their official response is reproduced below:

> *Using bytes instead of address for the _toAddress is for integration with other chains with different address schemes as EVM. if the _toÅddress is less than 20 bytes long, we consider that a user configuration problem.*

3.4. Usage of calldatacopy in inline assembly is unclear

- **Target:** Bridge
- **Severity:** n/a
- **Impact:** Informational
- **Category:** Code Maturity
- **Likelihood:** n/a

Description

The function `_packedBytesToAddr` is implemented correctly, but in a potentially misleading way that is confusing for readers.

```
function _packedBytesToAddr(bytes calldata _b) private returns (address) {
    address addr;
    assembly {
        let ptr := mload(0x40)
        calldatacopy(ptr, sub(_b.offset, 2), add(_b.length, 2))
        addr := mload(sub(ptr, 10))
    }
    return addr;
}
```

The function implicitly truncates the upper (first) 12 bytes of the `mload`ed contents when assigning to `addr`. The current implementation copies 2 bytes of `_b`'s length, which is laid out in calldata memory directly preceding the contents of `_b` at `_b.offset`. It unnecessarily `mloads` these 2 bytes as well as 10 bytes of currently in-use memory whose values are undefined. It is unclear and undocumented why these 12 bytes with undefined behavior are loaded then immediately discarded.

Impact

The current implementation is confusing and distracts readers from the overall objective of the function: to cast a `bytes calldata _b` into an `address addr` by loading its first 20 bytes. This is potentially misleading and should be avoided. The unnecessary operations also waste gas.

The current implementation loads bytes from memory whose values are undefined. Although these bytes are discarded, undefined behavior should be avoided whenever possible.

Relying on implementation-specific behavior is acceptable in certain situations, but it should be avoided whenever possible. In this situation, the function can be soundly rewritten in a more concise manner.

Recommendations

Replace the current implementation with the following suggested implementation, that is more clearly documented:

```
// Casts a bytes calldata _b into an address by loading its first 20 bytes.
// This function is necessary since Stargate is designed to work with any chain;
// and not all chains have address which are 20 bytes. Hence, addresses are
// represented as byte slices of arbitrary length, rather than a native address
type.
function packedBytesToAddr(bytes calldata _b) public view returns (address) {
    address addr;
    assembly {
        let ptr := mload(0x40) // Current free memory pointer
        calldatacopy(ptr, _b.offset, _b.length)
        addr := mload(sub(ptr, 12))
    }
    return addr;
}
```

Remediation

The issue has been acknowledged and fixed by LayerZero.

3.5. Missing test suite code coverage

- **Target:** Multiple contracts
- **Severity:** Low
- **Impact:** Informational
- **Category:** Code Maturity
- **Likelihood:** n/a

Description

Several functions in the smart contract are not covered by any unit or integration tests, to the best of our knowledge. We ran the entire Hardhat test suite and reviewed the Solcov coverage report. The following functions do not have test coverage:

Bridge.sol: `forceResumeReceive`, `setConfig`, `getConfig`, `setSendVersion`, `setReceiveVersion`, `getSendVersion`, `getReceiveVersion`

OmnichainFungibleToken.sol: `forceResumeReceive`, `setConfig`, `getConfig`, `setSendVersion`, `setReceiveVersion`, `getSendVersion`, `getReceiveVersion`

These functions are simply wrappers around the LayerZero API, so we do not see them as a significant issue.

LPTokenERC20.sol: `increaseAllowance`, `decreaseAllowance`, `permit`

The functions `increaseAllowance` and `decreaseAllowance` are simply wrappers around the OpenZeppelin ERC20 functionality, so we do not see these as a significant issue. The `permit` function's implementation seems to be the same as in UniswapV2, a battle-tested project, so we do not see it as an issue either.

We also scanned for functions which had inadequate branch coverage. Reaching 100% branch coverage is ideal, but we understand that, especially for projects under active development like LayerZero, it may not always be a top priority. The most concerning instances of missing branch coverage were:

LPStaking.sol: partial coverage in `getMultiplier`, `pendingStargate`, and `updatePool`

The implementation of LPStaking seems similar to PancakeSwap MasterChef, a battle-tested project. One key difference is the addition of a bonus period, controlled by the variable bonusEndBlock. We reviewed the updated `getMultiplier` function, but did not find any problems. The `pendingStargate` function is unchanged from MasterChef. The two uncovered branches in the `updatePool` function are also unchanged.

Pool.sol: partial coverage in `_delta`

The missing coverage corresponds to running the delta algorithm, but without a full update. It was simple to create new tests to cover this case based on the existing, extensive test suite.

Router.sol: `retryRevert`, `clearCachedSwap`, `redeemLocalCheckOnRemote`, `_redeemLocalCallback`, `_swapRemote`

The try-catch error handling code related to reverts is untested. It appears that a Bridge mock is needed to test the revert functionality. We assume LayerZero plans to add tests for this code once a Bridge mock for testing has been implemented.

We reviewed all untested functions with increased scrutiny. Fortunately, we did not find any additional vulnerabilities.

Overall, the project demonstrates excellent test coverage across the whole code base, and we applaud LayerZero for their commitment to thorough testing.

Impact

Because correctness is so critically important when developing smart contracts, we recommend that all projects strive for 100% code coverage. Testing should be an essential part of the software development lifecycle. No matter how simple a function may be, untested code is always prone to bugs.

Recommendations

Expand the test suite so that all functions and their branches are covered by unit or integration tests.

Remediation

The issue has been fixed by LayerZero. They have added additional test coverage based on our recommendations.

3.6. Ideally, the usage of the LayerZero API should be more thoroughly documented

- **Target:** Multiple contracts
- **Severity:** Low
- **Impact:** Informational
- **Category:** Code Maturity
- **Likelihood:** n/a

Description

Several of the key contracts (Pool.sol, Bridge.sol, Router.sol, OmnichainFungibleToken.sol) implement or use the LayerZero API in some way. However, the way LayerZero is used is undocumented and confusing to new developers.

Impact

Since Stargate will be a first-party consumer of LayerZero API, it is important to have clear documentation on the API usage so that it can serve as a reference to other developers. If they cannot find adequate usage examples, it might lead to mis-usage of the API.

Recommendations

Add additional documentation specifically about how the LayerZero API is used, and how the code is interacting with it

Remediation

The issue has been acknowledged by LayerZero. Additional documentation will be added.

3.7. Use of balanceOf in fee calculations may lead to unfavorable rewarding incentives

- **Target:** StargateFeeLibraryV02
- **Severity:** Low
- **Impact:** Informational
- **Category:** Business Logic
- **Likelihood:** Low

Description

The `currentAssetSD` amount is set to the `balanceOf` the pool, which is manipulatable by sending tokens directly to the pool. The tokens would not be registered in the `deltaCredit` (and be unusable by any remote chains) but can lead to wrong `eqReward` calculation.

```
function getFees(
...
) external view override returns (Pool.SwapObj memory s) {
...
   uint256 currentAssetSD =
IERC20(tokenAddress).balanceOf(address(pool)).div(pool.convertRate());
   uint256 lpAsset = pool.totalLiquidity();
   if (lpAsset > currentAssetSD) {
       // in deficit
       uint256 poolDeficit = lpAsset.sub(currentAssetSD);
       uint256 rewardPoolSize = pool.eqFeePool();
           // reward capped at rewardPoolSize
           uint256 eqRewards = rewardPoolSize.mul(_amountSD).div(poolDeficit);
```

Impact

The calculated `eqReward` would be lower than it should be, reducing the incentive for the user to swap from low to high liquidity chains.

Recommendations

Keep an internal record of deposited tokens and use it to calculate the `eqReward`.

Remediation

The issue has been acknowledged by LayerZero. Because it is a minor issue and is not currently causing any issues, they plan to fix it in the future, in the V3 of the FeeLibrary.

4. Discussion

In this section, we discuss miscellaneous interesting observations during the audit that are noteworthy and merit some consideration.

In a parallel, internal audit by the LayerZero team, the following issues were identified and fixed:

- The variable `Bool ready` was moved to the first place in struct `Chainpath` for compact packing as a gas optimization.
- The type of `dstChainId` was fixed to be `uint16` instead of `uint256`, making it consistent all across the project.
- The `eqReward` is now also added to `deltaCredit` even in the case when no liquidity is added (i.e. during a remote LP redeem).
- Added `creditObj` in `quoteLayerZeroFee()` to correctly calculate the LayerZero Fee.