Zellic

# Trident Concentrated Liquidity Pool

## Smart Contract Security Assessment

**November 16, 2022**

*Prepared for:*

**Sarang**

SushiSwap

*Prepared by:*

**Katerina Belotskaia and Mark Griffin**

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 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 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 Executive Summary

Zellic conducted an audit for SushiSwap from October 17th to October 24th, 2022.

Our general overview of the code is that it was very well-organized and structured. The code coverage is high, and tests are included for the majority of the functions. The documentation was adequate, although it could be improved. The code was easy to comprehend, and in most cases, intuitive despite the complex mathematical constructs required to implement concentrated liquidity pools in a flexible and generalized manner.

We applaud SushiSwap for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of Trident Concentrated Liquidity Pool. The code demonstrated a dedication to optimization and has been streamlined in the details as well as clear in its overall architecture.

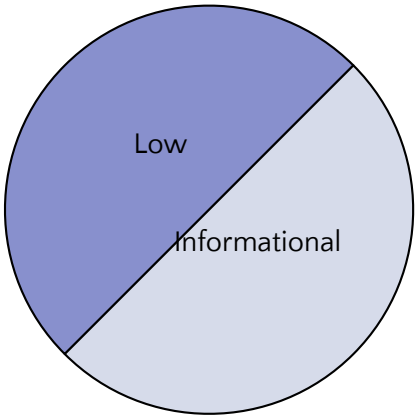
Zellic thoroughly reviewed the Trident Concentrated Liquidity Pool codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

Specifically, taking into account Trident Concentrated Liquidity Pool's threat model, we focused heavily on issues that would break core invariants, ensuring that minting, burning, and swapping always appropriately represent liquidity providers' positions; that users cannot extract excess value; and that the liquidity pool manager properly handles concurrent users and distributes fees fairly. We also focused on ensuring that cross-contract operations allow users to interact with the pool as desired while only transferring funds between approved or intended contracts.

During our assessment on the scoped Trident Concentrated Liquidity Pool contracts, we discovered two findings. Fortunately, no critical issues were found. Of the two findings, one was of low severity, and one was informational in nature, with additional Discussion items to consider (4) at the end of the document.

## Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| Critical | 0 |
| High | 0 |
| Medium | 0 |
| Low | 1 |
| Informational | 1 |



# 2 Introduction

## 2.1 About Trident Concentrated Liquidity Pool

Trident Concentrated Liquidity Pool is a liquidity pool that allows liquidity providers to provide liquidity for a specified price range. Providing liquidity on a narrower price range has an amplifying effect on the added liquidity, meaning traders will experience lesser price impacts. This makes the concentrated liquidity pool more capital efficient than the classic pool, with the tradeoff being that liquidity providers can suffer greater impermanent loss. Each concentrated liquidity pool supports two assets and is designed to be integrated with the larger Trident framework.

## 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 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, and so forth 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, and so forth.

**Complex integration risks.** Several high-profile 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. We perform a meticulous review of all of the contract's possible external interactions and summarize the asso-

ciated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

**Code maturity.** We review for possible 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, upgradeability weaknesses, centralization risks, and so forth.

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 being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

## 2.3 Scope

The engagement involved a review of the following targets:

### Trident Concentrated Liquidity Pool Contracts

| | |
|---|---|
| **Repository** | https://github.com/sushiswap/trident/ |
| **Versions** | `cc84e700dc8e1af26b86889f5e355b2373433364` |
| **Programs** | • ConcentratedLiquidityPool<br>• ConcentratedLiquidityPoolFactory<br>• ConcentratedLiquidityPoolHelper<br>• ConcentratedLiquidityPoolManager<br>• ConcentratedLiquidityPoolStaker<br>• TridentRouter<br>• TridentRouterLibrary |
| **Type** | Solidity |
| **Platform** | EVM-compatible |
| **Repository** | https://github.com/sushiswap/trident/pull/356 |
| **Versions** | `89b1cb6e17b9e2ec6c3b8825a8d78b8a0ab400d5` |
| **Programs** | • TridentRouter<br>• TridentRouterLibrary |
| **Type** | Solidity |
| **Platform** | EVM-compatible |

## 2.4 Project Overview

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

### Contact Information

The following project managers were associated with the engagement:


**Jasraj Bedi**, Co-founder
jazzy@zellic.io

**Stephen Tong**, Co-founder
stephen@zellic.io


The following consultants were engaged to conduct the assessment:


**Katerina Belotskaia**, Engineer
kate@zellic.io

**Mark Griffin**, Engineer
mark@zellic.io

## 2.5 Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **October 17, 2022** | Start of primary review period |
| **October 24, 2022** | End of primary review period |

# 3 Detailed Findings

## 3.1 Lack of validation

- **Target**: ConcentratedLiquidityPoolManager, ConcentratedLiquidityPool, ConcentratedLiquidityPoolStaker
- **Category**: Business Logic
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

### Descriptions

Below are the listings of each area missing a check:

1. ConcentratedLiquidityPoolManager.sol
   - `burn()` - verify the existence of the `position` by checking `position.liquidity ≠ 0`.
2. ConcentratedLiquidityPool.sol
   - `mint()` - check that the contract balance for `token0` and `token1` is the same or greater after an external `mintCallback` call to an untrusted contract, even if the `amount0Actual` and `amount1Actual` values are zero.
   - `burn()` - check that `lower` is less than `upper`.
   - `swap()` - check that the `inAmount` is not equal to zero.
   - `_updateReserves` - check that the `reserve1` and `reserve0` are no less than `amountOut`.
3. ConcentratedLiquidityPoolStaker
   - `claimRewards()` - verify the existence of the `position` by checking `position.liquidity ≠ 0`.
4. Ticks.sol
   - `cross()` - add a check or remove the `unchecked` block wrapping `currentLiquidity -= ticks[nextTickToCross].liquidity` to prevent underflow.

### Impact

Code maturity is very important in high-assurance projects. Checks help safeguard against unfortunate situations that might occur, help reduce the risk of lost funds as a

result of a swap or burn of tokens, and improve UX. Adding extra reverts can help clarify the internal mechanisms and reduce potential bugs that future developers might introduce while building on this project.

Below are example of when the lack of checks can lead to incorrect behavior of the contract.

Inside the `Ticks.cross` function, there are no checks that `currentLiquidity` is less then `ticks[nextTickToCross].liquidity`. Due to this lack of check during the `swap`, an underflow may occur in the case `currentLiquidity == 0` and `Ticks.cross` functions will be called to update the `currentLiquidity` value.

```
function cross(
    mapping(int24 => IConcentratedLiquidityPoolStruct.Tick) storage
    ticks,
    int24 nextTickToCross,
    uint160 secondsGrowthGlobal,
    uint256 currentLiquidity,
    ...
    ) internal returns (uint256, int24) {
        ticks[nextTickToCross].secondsGrowthOutside = secondsGrowthGlobal
     - ticks[nextTickToCross].secondsGrowthOutside;

        if (zeroForOne) {
            unchecked {
                if ((nextTickToCross / int24(tickSpacing)) % 2 == 0) {
                    currentLiquidity -= ticks[nextTickToCross].liquidity;
                } else {
                    ...
                }
            }
         ...
```

## Recommendations

We recommend adding the requisite checks/reverts to the areas above and adding documentation to clarify reverts.

## Remediation

SushiSwap acknowledged this finding. The issue from first point has been fixed in commit 557fdf0e. The issue from third point has been fixed in commit 3df8346c.

## 3.2 Centralization risks

- **Target**: MasterDeployer, ConcentratedLiquidityPool
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

### Descriptions

There are a few centralization risks worth consideration.

1. The Owner may set the `barFee` in the value of 100% of the provider's liquidity fee.
2. The Owner can whitelist an unreliable factory contract address or delete a trusted one.

### Impact

If the private key is compromised, an attacker can get the entire amount of fee from the swaps.

### Recommendations

Use a multi-signature address wallet; this would prevent an attacker from causing economic damage if a private key were compromised.

### Remediation

SushiSwap acknowledged this finding.

# 4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

## 4.1 Checks-effects-interactions pattern

Consider refactoring the below functions to follow the checks-effects-interactions pattern.

- In `ConcentratedLiquidityPool.sol:mint`, move `_transfer` below the changes to the reserves.

```
if (amount0Fees > 0) {
        reserve0 -= uint128(amount0Fees);
        _transfer(token0, amount0Fees, msg.sender, false);
        reserve0 -= uint128(amount0Fees);
}

if (amount1Fees > 0) {
        reserve1 -= uint128(amount1Fees);
        _transfer(token1, amount1Fees, msg.sender, false);
        reserve1 -= uint128(amount1Fees);
}
```

- In `ConcentratedLiquidityPoolManager:burn`, move the `position.pool.burn` call below the state changes.

```
if (amount < position.liquidity) {
        (token0Amount, token1Amount, , ) =

    position.pool.burn(position.lower, position.upper, amount);
        positions[tokenId].feeGrowthInside0 = feeGrowthInside0;
        positions[tokenId].feeGrowthInside1 = feeGrowthInside1;
        positions[tokenId].liquidity -= amount;
        (token0Amount, token1Amount, , ) =
```

```
        position.pool.burn(position.lower, position.upper, amount);
    } else {
            amount = position.liquidity;
            position.liquidity = 0;
            (token0Amount, token1Amount, , ) = position.pool.burn(position.
        lower, position.upper, amount);
            burn(tokenId);
            delete positions[tokenId];
    }
```

### Remediation

SushiSwap acknowledged this finding.

## 4.2 Increasing test suite code coverage

The line coverage from the existing unit tests cover most of the functions reviewed, but in a codebase with complex mathematical calculations, line coverage does not fully address the state space. For example, if a line is within in an `unchecked` block and an integer overflow could occur, we would consider the overflow to be an important and distinctly different behavior that would not be reflected in line coverage. While line coverage is imperfect, we highly suggest implementing unit tests to represent the most complex expected multi-user interactions, such as multiple users minting/burning/collecting at different times through the `ConcentratedLiquidityPoolManager` with both overlapping and disjoint liquidity ranges.

Here is a list of functions and an additional special case that are not covered by the existing unit tests:

- `ConcentratedLiquidityPool.sol`
  - `getAmountIn`
  - `updateBarFee`
  - within `swap` , the “overflow” case is uncovered
- `ConcentratedLiquidityPoolStaker.sol`
  - `reclaimIncentive`
  - `getReward`
- `TridentRouter.sol`
  - `harvest`

In addition, while we could use specific measures of code complexity (such as cyclomatic complexity, Halstead complexity, or simply lines of code), some of the key

functions in `ConcentratedLiquidityPool` have an objectively large potential state space in terms of data variables and control flows. For such functions (such as `swap`, `mint`, `getAmountIn`, and `burn`), it is difficult to fully comprehend and reason about all of the possible behaviors, so we highly recommend going beyond unit tests and leveraging fuzzing, symbolic execution, or formal methods to increase assurance. While we were unable to apply these methods during the time given for the audit, we strongly encourage the development team to pursue these avenues to decrease the likelihood that unexpected behavior could occur.

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.

### Remediation

SushiSwap acknowledged this finding. Tests for `ConcentratedLiquidityPoolStaker.sol` have been added in commit 2b11bb33.

## 4.3 Tension between security and gas optimization

Throughout the code reviewed, input checks are deferred and pushed down to the lowest possible level. This makes sense since the system is advertised as a gas-optimized alternative and redundant checks cost gas, but this is in tension with best security practices of defense in depth. While using unchecked blocks and deferring input checks to other functions reduces gas costs, it increases the risk that unexpected values will not be caught, and undesirable behavior can result. Also, many common gas-optimization strategies result in code that is more complex or more difficult to read and reason about, which is at odds with best security practices.

Our team was impressed by the quality of the code and did not find any correctness issues due to optimizations during the time of the audit, but overzealous gas optimization strategies may obscure bugs and add to the difficulty of maintaining all of the security invariants in the case of code updates.

### Remediation

SushiSwap acknowledged this finding.

# 5 Audit Results

At the time of our audit, the code was not deployed to mainnet EVM.

During our audit, we discovered two findings. Of these, one was low risk and one was a suggestion (informational). SushiSwap acknowledged all findings and implemented fixes.

## 5.1 Disclaimers

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 additional code added to the assessed project after the audit version of 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 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.

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.