Zellic

SushiSwap RouteProcessor3

Smart Contract Security Assessment

April 18, 2023

Prepared for:

Jared Grey

Sushiswap

Prepared by:

Filippo Cremonese and Junyi Wang

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 contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Sushiswap from April 13th to April 14th, 2023. During this engagement, Zellic reviewed SushiSwap RouteProcessor3's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 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 there any way for an attacker to exploit the route processor to steal funds?
- Are there any potential memory-related issues with the InputStream library that need to be addressed, especially in edge cases?
- Are there any user interactions with the project that may lead to fund losses or other security risks?

1.2 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.

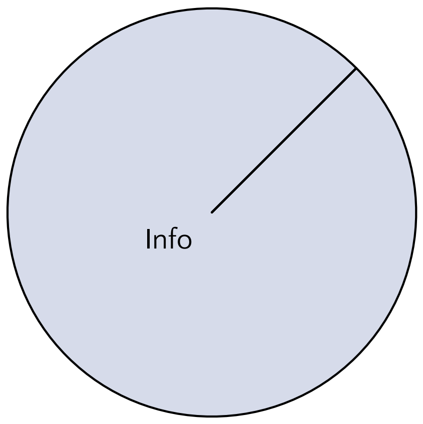
1.3 Results

During our assessment on the scoped SushiSwap RouteProcessor3 contracts, we discovered four findings. No critical issues were found. The four findings were informational in nature.

Additionally, Zellic created an attack surface analysis for Sushiswap's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

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



2 Introduction

2.1 About SushiSwap RouteProcessor3

SushiSwap RouteProcessor3 is a contract that processes complex swap routes. It can mix pools from different DEXes such as Sushiswap and Uniswap in one route. It also allows for liquidity merges and splits at arbitrary points of the route. The contract does not create routes; it merely processes routes that are created off-chain.

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 the contracts' 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 code base 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 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.

2.3 Scope

The engagement involved a review of the following targets:

SushiSwap RouteProcessor3 Contracts

Repository	https://github.com/sushiswap/sushiswap
Version	sushiswap: `38e553974e4d35ec85a3aceaaa4a353f053e64df`
Program	• RouteProcessor3
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese, Engineer
fcremo@zellic.io

Junyi Wang, Engineer
junyi@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

April 13, 2023	Start of primary review period
April 14, 2023	End of primary review period

3 Detailed Findings

3.1 The RouteProcessor3 should not hold nontransient tokens

- **Target**: RouteProcessor3.sol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

There are numerous ways in which tokens can be stolen if they are held by the RouteProcessor3.

For example, one way is by directly asking the contract to wrap or unwrap Ether and transfer it to the user.

```
function wrapNative(uint256 stream, address from, address tokenIn,
    uint256 amountIn) private {
    uint8 directionAndFake = stream.readUint8();
    address to = stream.readAddress();

    if (directionAndFake & 1 == 1) { // wrap native
        address wrapToken = stream.readAddress();
        if (directionAndFake & 2 == 0)
    IWETH(wrapToken).deposit{value: amountIn}();
        if (to ≠ address(this)) IERC20(wrapToken).safeTransfer(to,
    amountIn);
    } else { // unwrap native
        if (directionAndFake & 2 == 0) {
            if (from ≠ address(this))
    IERC20(tokenIn).safeTransferFrom(from, address(this), amountIn);
            IWETH(tokenIn).withdraw(amountIn);
        }
        payable(to).transfer(address(this).balance);
    }
}
```

The `wrapNative` function can be reached with `from == address(this)` by going from `processRouteInternal` to `processNative`, listed below,

```
function processNative(uint256 stream) private {
    uint256 amountTotal = address(this).balance;
    distributeAndSwap(stream, address(this), NATIVE_ADDRESS, amountTotal);
}
```

then requesting a `wrapNative` operation in the swap. The tokens belonging to RouteProcessor3 will then be wrapped or unwrapped and transferred to the user.

Impact

The RouteProcessor3 contract should not hold tokens except transiently, in the middle of a transaction.

Recommendations

Document prominently that the `RouteProcessor3` contract should not hold tokens.

Remediation

This issue has been acknowledged by Sushiswap.

3.2 The `safePermit` call can be front-run

- **Target**: RouteProcessor3.sol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

In the RouteProcessor3, a user can provide a cryptographically signed permit that, when consumed, will allow the contract to send tokens on behalf of the user.

```
function applyPermit(address tokenIn, uint256 stream) private {
    //address owner, address spender, uint value, uint deadline, uint8 v,
    bytes32 r, bytes32 s)
    uint256 value = stream.readUint();
    uint256 deadline = stream.readUint();
    uint8 v = stream.readUint8();
    bytes32 r = stream.readBytes32();
    bytes32 s = stream.readBytes32();
    IERC20Permit(tokenIn).safePermit(msg.sender, address(this), value,
    deadline, v, r, s);
}
```

The values of the signature are visible in the mempool until the transaction is executed. An attacker could use the genuine signature to invoke the exact call to `IERC20Permit.permit`.

Impact

This does not cause any loss of funds, as the contract will not send funds on behalf of anyone except `msg.sender` and itself. However, it will cause a subsequent transaction to fail on `IERC20Permit.safePermit`, since the nonce will be incremented and the signature cannot be used again.

Recommendations

Potentially, ignore reverts caused by `safePermit` calls. The contract will revert anyway when attempting to transfer tokens that are not authorized. This prevents a frontrun from halting the transaction.

Remediation

This issue has been acknowledged by Sushiswap.

3.3 Overflow in `readBytes`

- **Target**: InputStream.sol
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

The `InputStream` library can be used to treat a `bytes` variable as a read-only stream, managing a cursor that is incremented automatically as the stream is consumed.

The `readBytes` function can be used to read a sequence of bytes from the stream. The sequence is encoded as a length, followed by the contents of the sequence. Since the length is user provided, we believe a potential integer overflow exists in the function.

```
function readBytes(uint256 stream)
    internal pure returns (bytes memory res) {
    assembly {
        let pos := mload(stream)
        res := add(pos, 32)
        let length := mload(res)
        mstore(stream, add(res, length))
    }
}
```

The `stream` variable keeps track of the current position of the stream. It is updated with the new position after the sequence is read, by adding the length of the sequence, which is user-provided. This addition can overflow.

Impact

This does not represent an exploitable security issue in the context of RouteProcessor3, since the data provided to `readBytes` is controlled by the same user that invokes the contract. We also believe no reasonable usage of the contract would trigger this bug by accident.

For these reasons, this is reported as informational, with the purpose of providing hardening suggestions for the `InputStream` library, which might be important if it was used in other contexts.

Recommendations

Ensure the calculation of the new stream position does not overflow.

Remediation

This issue has been acknowledged by Sushiswap.

3.4 Integer overflows and underflows in unchecked blocks

- **Target**: RouteProcessor3.sol
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

The contract makes sporadic use of `unchecked` blocks to improve gas efficiency. Solidity does not perform some safety checks when compiling code placed inside `unchecked` blocks; most notably, it does not check for integer overflows and underflows.

While most `unchecked` blocks are safe, some functions such as `distributeAndSwap` contain code that might potentially compute results that over/underflow.

```
function distributeAndSwap(uint256 stream, address from, address tokenIn,
    uint256 amountTotal) private {
  uint8 num = stream.readUint8();
  unchecked {
    for (uint256 i = 0; i < num; ++i) {
      uint16 share = stream.readUint16();
      uint256 amount = (amountTotal * share) / 65535; // OVERFLOW
      amountTotal -= amount; // UNDERFLOW
      swap(stream, from, tokenIn, amount);
    }
  }
}
```

In this case, the `amount` variable might overflow, and the `amountTotal -= amount` calculation might underflow.

Impact

This does not represent an exploitable security issue in the context of RouteProcessor3, since the arguments provided to the contract are controlled by the same user that invokes the contract and would be affected by an over/underflow. Furthermore, we believe no reasonable usage of the contract would trigger this bug by accident.

We report this issue as informational as a hardening suggestion.

Recommendations

Check over and underflows on calculations that might potentially cause such a condition.

Remediation

This issue has been acknowledged by Sushiswap.

4 Discussion

This section serves to record various observations we noted during the evaluation. Our focus was on conducting an analysis of the attack surface, which included external calls, reentrancy, and unexpected callers.

4.1 Reentrancy and unexpected callers

Two of the most dangerous threats faced by RouteProcessor3 are malicious calls that are reentrant or come from an unexpected/unauthorized contract.

Reentrancy

The attack surface for reentrant calls is reduced to the bare minimum. There are only four state-mutating public functions that do not require authorization: `receive`, `processRoute`, `transferValueAndprocessRoute`, and `uniswapV3SwapCallback`. The `receive` function does nothing and can be safely ignored for the purposes of this section.

The `processRoute` and `transferValueAndprocessRoute` functions are protected against reentrancy by locking the contract when they are first called using the `lock` modifier.

The `uniswapV3SwapCallback` function cannot be called directly; in other words, the call must be reentrant and follow the expected flow. This is because the caller is required to be the address stored in the `lastCalledPool` variable, which is set to `IMPOSSIBLE_POOL_ADDRESS` and only ever changed to a different value by `swapUniV3`, which is only reachable through `processRoute` or `transferValueAndprocessRoute`. The variable is reset to `IMPOSSIBLE_POOL_ADDRESS` by `uniswapV3SwapCallback` before performing any external call, enforcing a strict call flow that has to go through `swapUniV3` -> `(external call) Uniswap` -> `(reentrant call from uniswap) uniswapV3SwapCallback` -> `(return) swapUniV3`.

We deem this architecture to be safe against reentrancy attacks.

Unexpected callers

It is paramount that the implementation of the `IUniswapV3SwapCallback` interface ensures the caller is a legitimate Uniswap pool, to prevent sending funds to an arbitrary address.

In order to do so, `RouteProcessor3` uses the `lastCalledPool` variable to store the address of the Uniswap pool that has to invoke `uniswapV3SwapCallback`. The variable is initialized to `IMPOSSIBLE_POOL_ADDRESS` (an invalid address) and is only ever changed to a different value in `swapUniV3`. The variable is also always restored to the `IMPOSSIB`

LE_POOL_ADDRESS before a transaction ends. Thus, it is not possible to call uniswapV3SwapCallback from an unexpected or otherwise incorrect address.

4.2 External calls

All external calls the RouteProcessor3 contract makes are on behalf of users. The external calls are public functions anyone can call without going through RouteProcessor3. This means the only functional difference is the `msg.sender`, which is set to RouteProcessor3 when calling an external function through the contract.

In order to steal user funds exploiting RouteProcessor3, the attacker must find an external call where calling it through RouteProcessor3 and calling it directly gives a different result. That is, `msg.sender == RouteProcessor3` gives special privileges.

We make the following assumptions:

- External contracts behave following a common convention. We assume that, for example, `IERC20.balanceOf` will not send funds to the caller.
- The RouteProcessor3 is not meant to hold funds between transactions. This is part of the design.

We will now analyze all external calls made by this contract. The following are all the non-state-mutating external calls made by RouteProcessor3.

```
IERC20(tokenOut).balanceOf(to)
```

```
bentoBox.strategyData(tokenIn).balance
```

```
bentoBox.totals(tokenIn).elastic
```

```
bentoBox.balanceOf(tokenIn, address(this));
```

```
IUniswapV2Pair(pool).getReserves();
```

These functions cannot mutate state and therefore cannot be directly used in an attack.

The following function calls transfer funds from the RouteProcessor3 itself.

```
IERC20(wrapToken).safeTransfer(to, amountIn);
```

```
IERC20(tokenIn).safeTransfer(address(bentoBox), amountIn);
```

```
IERC20(tokenIn).safeTransfer(pool, amountIn);
```

```
IERC20(tokenIn).safeTransfer(msg.sender, uint256(amount));
```

These function calls cannot be used in an attack since it only transfers from RouteProcessor3 itself, which is not supposed to hold funds.

The following function calls perform some action as the RouteProcessor3.

```
IERC20Permit(tokenIn).safePermit(msg.sender, address(this), value,
    deadline, v, r, s);
```

The permit must be constructed by the `msg.sender`, since the signature is checked.

```
IWETH(wrapToken).deposit{value: amountIn}();
```

```
IUniswapV2Pair(pool).swap(amount0Out, amount1Out, to, new bytes(0));
```

```
IPool(pool).swap(swapData);
```

```
IUniswapV3Pool(pool).swap(
      recipient,
      zeroForOne,
      int256(amountIn),
      zeroForOne ? MIN_SQRT_RATIO + 1 : MAX_SQRT_RATIO - 1,
      abi.encode(tokenIn)
    );
```

The following function calls transfer funds on behalf of a known address.

```
bentoBox.deposit(tokenIn, address(bentoBox), to, amountIn, 0);
```

```
bentoBox.withdraw(tokenIn, address(this), to, 0, amountIn);
```

```
IERC20(tokenIn).safeTransferFrom(msg.sender, address(this),
    uint256(amountIn));
```

These calls cannot be used to attack users since the payer is not controlled.

The following function calls transfer funds on behalf of the `from` address.

```
IERC20(tokenIn).safeTransferFrom(from, address(this), amountIn);
```

```
IERC20(tokenIn).safeTransferFrom(from, address(bentoBox), amountIn);
```

```
bentoBox.transfer(tokenIn, from, address(this), amountIn);
```

```
IERC20(tokenIn).safeTransferFrom(from, pool, amountIn);
```

```
bentoBox.transfer(tokenIn, from, pool, amountIn);
```

The following are all sources of the `from` variable.

```
distributeAndSwap(stream, address(this), NATIVE_ADDRESS, amountTotal);
```

```
distributeAndSwap(stream, address(this), token, amountTotal);
```

```
distributeAndSwap(stream, msg.sender, token, amountTotal);
```

```
swap(stream, address(this), token, 0);
swap(stream, address(this), token, amount);
```

The second parameter is the `from` variable. The `from` is either `address(this)` or `msg.sender`, which means the contract will not abuse another user's authorization.

Since all external calls do not grant any special privileges, the attacker cannot abuse this contract to attack other users.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

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

5.1 Module: RouteProcessor3.sol

Function: `pause()`

This function allows callers with appropriate authorization to pause the contract.

Branches and code coverage (including function calls)

- The contract is paused.
  - ☐ Test Coverage

Negative behavior

- Unauthorized callers cannot call this function.
  - ☐ Test Coverage

Function: `processRouteInternal(address tokenIn, uint256 amountIn, address tokenOut, uint256 amountOutMin, address to, bytes memory route)`

This private function handles calls to the `processRoute` and `transferValueAndprocessRoute` public functions.

It processes the route defined by the caller one step at a time, performing the requested operations by dispatching the task to appropriate subfunctions.

Inputs

- `tokenIn`
  - **Control**: Arbitrary.
  - **Constraints**: Should be an ERC20 token address or `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`.
  - **Impact**: Determines the asset used as input for `applyPermit` and input bal-

ance checks.

- `amountIn`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the amount of input asset to be used by the route.
- `tokenOut`
  - **Control**: Arbitrary.
  - **Constraints**: Should be an ERC20 token address or `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`.
  - **Impact**: Determines the asset used as output for the transaction.
- `amountOutMin`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the minimum amount of output asset resulting from the transaction (slippage protection).
- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the intented recipient of the output asset.
- `route`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the route (sequence of operations and parameters) to be used.

Branches and code coverage (including function calls)

The function first reads the initial input and output balances to be used at the end for the slippage protection checks.

The `route` byte buffer is read to identify the required steps in the route. Every step is represented by a byte in the stream (with values from 1 to 6), followed by the data required to perform that step. Only the first byte required to identify the required operation is read by `processRouteInternal`; the route byte stream is passed to functions implementing the details of each specific operation.

Finally, the function reads the final balances and ensures they satisfy the minimum expected values to protect against slippage, MEV, and other risks.

Intended branches

- ERC20 swap operations using `address(this)` balance are handled (`processMyER`

C20).
  - ☑ Test Coverage
- ERC20 swap operations using the `msg.sender` balance are handled (`processUserERC20`).
  - ☑ Test Coverage
- ETH swap operations using the `address(this)` balance are handled (`processNative`).
  - ☑ Test Coverage
- ERC20 operations using the `address(this)` balance already in the pool are handled (`processOnePool`).
  - ☐ Test Coverage (unclear if tests exercise this functionality)
- Swap operations with Bento tokens are handled (`processInsideBento`).
  - ☐ Test Coverage (unclear if tests exercise this functionality)
- ERC20 permit operations are handled (`applyPermit`).
  - ☑ Test Coverage (tests exercise this functionality)
- Anti-slippage checks are performed.
  - ☐ Test Coverage (tests check output balance, but they do not look for a revert)

Negative behavior

- Invalid operations (`commandCode` not between 1 and 6) cause a revert.
  - ☐ Test Coverage
- Unexpected final input balance causes a revert.
  - ☐ Test Coverage (tests do not appear to look for the revert)
- Unexpected final output balance causes a revert.
  - ☐ Test Coverage (tests do not appear to look for the revert; however, they do check balances)

Function: `resume()`

This function allows callers with appropriate authorization to unpause the contract.

Branches and code coverage (including function calls)

- The contract is unpaused.
  - ☐ Test Coverage

Negative behavior

- Unauthorized callers cannot call this function.
  - ☐ Test Coverage

Function: `setPriviledge(address user, bool priviledge)`

This function allows authorized callers (the owner of the contract) to grant or revoke privileges to other addresses. Currently, privileged users can pause/unpause the contract.

Inputs

- `user`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the user on which privileges are granted or revoked.
- `priviledge`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines whether privileges are granted or revoked.

Branches and code coverage (including function calls)

Intended branches

- Privileges are granted or revoked on the provided address.
  - ☐ Test Coverage

Negative behavior

- Unauthorized addresses cannot call this function.
  - ☐ Test Coverage

Function: `uniswapV3SwapCallback(int256 amount0Delta, int256 amount1Delta, byte[] data)`

This function implements the `IUniswapV3SwapCallback` interface. It is called by UniswapV3 when executing a swap to request a transfer of the collateral for a swap.

Inputs

- `amount0Delta`
  - **Control**: Arbitrary.
  - **Constraints**: Arbitrary.
  - **Impact**: If positive, it determines the amount of the first asset in the pair that must be sent to the pool.
- `amount1Delta`

  - **Control**: Arbitrary.
  - **Constraints**: Arbitrary.
  - **Impact**: If positive, it determines the amount of the second asset in the pair that must be sent to the pool.
- `data`
  - **Control**: Arbitrary.
  - **Constraints**: Must encode an address.
  - **Impact**: Determines the address of the token to be sent to the pool (this data is supposed to be originally provided to Uniswap by RP3).

Branches and code coverage (including function calls)

Intended branches

- The requested amount is sent to the pool.
  ☑ Test Coverage

Negative behavior

- The caller is not the expected Uniswap pool.
  ☐ Test Coverage
- The requested amount is negative and causes a revert.
  ☐ Test Coverage

Function call analysis

- `uniswapV3SwapCallback → IERC20(tokenIn).safeTransfer(msg.sender, uint256 (amount))`
  - **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The revert is bubbled up, as intended.

6 Audit Results

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

During our audit, we discovered four findings. No critical issues were found. The four findings were informational in nature. Sushiswap acknowledged all findings and implemented fixes where noted.

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 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 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.

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.