Zellic

Route Processor

Smart Contract Security Assessment

January 12, 2023

Prepared for:

Sushiswap

Prepared by:

Katerina Belotskaia and Vlad Toie

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 January 4th to January 9th, 2023. During this engagement, Zellic reviewed Route Processor'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:

- Could a malicious attacker leverage the route processor somehow to steal funds?
- Is the memory handled properly in the InputStream library? Could there be any edge cases?
- Could the user interact with the project in a way that causes them to lose funds?

1.2 Non-goals and Limitations

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper generation of the routes that the users are using when interacting with the project - these routes are generated by the Sushiswap team and are not part of the scope of this assessment
- Issues stemming from code or infrastructure outside of the assessment scope, or their interactions with the code in scope

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. Our focus during the assessment was on ensuring the proper functioning of the routes and the security of the code.

However, given the complexity and unique usage of the project, it is important to recognize that there may be unforeseen issues that arise when users interact with it through the Sushiswap web interface. These issues could include locking up of funds or other problems. To mitigate this risk, we recommend ongoing monitoring and testing of the project as it is used by users as well as having a system in place for users to report any issues they encounter and for the team to promptly address them.

Additionally, it is important to note that when the user interacts with the protocol outside of the Sushiswap web interface, we cannot guarantee expected functionality. This is due to the InputStream library being designed to be used through the use of

Sushiswap's API and/or web interface, as arbitrary usage (e.g., passing an 'uint' instead of an 'address', which would still work because of the assembly) could lead to unexpected behavior and potential loss of funds.

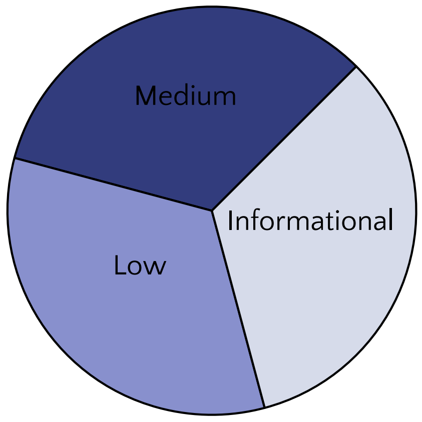
1.3 Results

During our assessment on the scoped Route Processor contracts, we discovered three findings. No critical issues were found. Of the three, one was of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment 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	1
Low	1
Informational	1



2 Introduction

2.1 About Route Processor

Route Processor 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:

Route Processor Contracts

Repository	https://github.com/sushiswap/sushiswap/
Versions	`285f7eb59edf2b65afedff3fca916d7178963f1e`
Programs	• InputStream • RouteProcessor
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

Katerina Belotskaia, Engineer
kate@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

January 4, 2022	Start of primary review period
January 9, 2022	End of primary review period

3 Detailed Findings

3.1 The transferred amount may not reflect `msg.value`

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

Description

The `wrapAndDistributeERC20Amounts` function wraps the native tokens that were supplied by the user and then forwards them to the pools that `RouteProcessor` interacts with. Here, the `msg.value` parameter is not checked against the `amountTotal` variable, leaving room for error.

```
function wrapAndDistributeERC20Amounts(uint256 stream, address token)
    private returns (uint256 amountTotal) {

    wNATIVE.deposit{value: msg.value}();

    uint8 num = stream.readUint8();
    amountTotal = 0;

    for (uint256 i = 0; i < num; ++i) {
      address to = stream.readAddress();
      uint256 amount = stream.readUint();
      amountTotal += amount;
      IERC20(token).safeTransfer(to, amount);
    }
}
```

Impact

This could lead to loss of funds for the end user in the case that they transfer more than the required amount.

Recommendations

We recommend adding a check to ensure that `msg.value == amountTotal` at the end of the function, as shown below:

```
function wrapAndDistributeERC20Amounts(uint256 stream, address token)
    private returns (uint256 amountTotal) {

    wNATIVE.deposit{value: msg.value}();

    uint8 num = stream.readUint8();
    amountTotal = 0;

    for (uint256 i = 0; i < num; ++i) {
      address to = stream.readAddress();
      uint256 amount = stream.readUint();
      amountTotal += amount;
      IERC20(token).safeTransfer(to, amount);
    }

    require(msg.value == amountTotal, “RouteProcessor: invalid amount”);
}
```

Remediation

This issue was fixed by Sushiswap in commit 4aa4bd3.

3.2 Arbitrary token transfers in `wrapAndDistributeERC20Amounts`

- **Target**: `RouteProcessor`
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The `wrapAndDistributeERC20Amounts` function wraps and then forwards the wrapped tokens from the `RouteProcessor` contract to the pools that it interacts with.

```
function wrapAndDistributeERC20Amounts(uint256 stream, address token)
    private returns (uint256 amountTotal) {

    wNATIVE.deposit{value: msg.value}();

    uint8 num = stream.readUint8();
    amountTotal = 0;

    for (uint256 i = 0; i < num; ++i) {
      address to = stream.readAddress();
      uint256 amount = stream.readUint();
      amountTotal += amount;

      // @audit arbitrary `token` is passed, instead of `wNATIVE`
      IERC20(token).safeTransfer(to, amount);
    }
}
```

Due to the way the `token` parameter is passed to the `safeTransfer` function, it is possible to pass an arbitrary token address to the function. This allows for anyone to send tokens on behalf of the contract.

This is not a highly critical issue, as the `RouteProcessor` contract should, in theory, be interacted with via the Sushiswap front end, which would generate a legitimate `token` address in its route generation process. Moreover, it is not expected of the contract to hold any tokens, as it is designed to be used as a one-time transaction.

Impact

The transaction is reverted, and the tokens are not sent. In some cases, it could lead to tokens up for grabs in the MEV (e.g., via front-running), should any user unknowingly transfer tokens to the `RouteProcessor` contract.

Recommendations

We recommend removing the `token` parameter altogether.

```
function wrapAndDistributeERC20Amounts(uint256 stream)
    private returns (uint256 amountTotal) {

    wNATIVE.deposit{value: msg.value}();

    uint8 num = stream.readUint8();
    amountTotal = 0;

    for (uint256 i = 0; i < num; ++i) {
      address to = stream.readAddress();
      uint256 amount = stream.readUint();
      amountTotal += amount;

      IERC20(wNATIVE).safeTransfer(to, amount);
    }
    require(msg.value == amountTotal, “RouteProcessor: invalid amount”);
}
```

Remediation

This issue was fixed by Sushiswap in commit 4aa4bd3.

3.3 Usage of `transfer()` method for native tokens

- **Target**: `RouteProcessor`
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `unwrapNative` calls Solidity's `transfer` method to send full native tokens balance to `receiver` account.

Impact

The `transfer` method uses a hardcoded amount of gas (2300) and may fail if gas costs increase in the future.

Recommendations

Consider using the `payable(receiver).call.value(value)(“”)` function:

```
(bool success, )
    = payable(receiver).call.value(address(this).balance)(“”);
require(success, “Transfer failed.”);
```

Remediation

The Sushiswap team has decided not to address this particular finding at the time of writing this report and may consider addressing it in the future.

4 Discussion

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

4.1 Disallow `tokenIn` and `tokenOut` to be the same

The `tokenIn` and `tokenOut` parameters of the `processRoute` function are currently not checked to be different. As of the current state of the implementation, there seems to be no security implications of this. The sole issue would be the waste of gas for the transaction, which should revert eventually, during the UniswapV2Pair swap call.

However, it is not clear if this would be the case in the future. Therefore, it is recommended to add a check that `tokenIn` and `tokenOut` are different.

```
function processRoute(
    address tokenIn,
    uint256 amountIn,
    address tokenOut,
    uint256 amountOutMin,
    address to,
    bytes memory route
) external payable returns (uint256 amountOut) {
  require(tokenIn ≠ tokenOut, "TokenIn and TokenOut must be different");
```

4.2 Passing the `tokenIn` in a consistent way

In the `RouteProcessor` contract, the `tokenIn` parameter is passed in two different ways.

In the `distributeBentoShares` function, the `token` parameter is passed as a parameter, called with `token = tokenIn` in the `processRoute` parent function.

```
function processRoute(
    address tokenIn,
    uint256 amountIn,
    address tokenOut,
    uint256 amountOutMin,
    address to,
    bytes memory route
) external payable returns (uint256 amountOut) {
    // ...

    if (commandCode == 24) amountInAcc += distributeBentoShares(stream,
    tokenIn);

    // ...
}

function distributeBentoShares(uint256 stream, address token)
    private returns (uint256 amountTotal) {
    // ...
}
```

In the `bentoWithdrawAllFromRP`, however, the `tokenIn` from the parent function is disregarded, and the `token` is read from the `stream`, despite the fact that it is theoretically the same with the `tokenIn` parameter.

```
function processRoute(
    address tokenIn,
    uint256 amountIn,
    address tokenOut,
    uint256 amountOutMin,
    address to,
    bytes memory route
) external payable returns (uint256 amountOut) {
    // ...
    else if (commandCode == 27) bentoWithdrawAllFromRP(stream);
    // ..
}

function bentoWithdrawAllFromRP(uint256 stream) private {
    // ...
    address token = stream.readAddress();
    // ...
}
```

We recommend opting for the first approach and passing the `tokenIn` as a parameter to all the functions that do not do so already.

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 File: `InputStream`

Function: `createStream(bytes memory data)`

Intended behavior

- Internal function
- Allows creating a data stream for a route
- Stores the actual data and its length; presumably there's no limit to the length of the data
- Stores everything in memory, and nothing in `storage`

Branches and code coverage

Intended branches

- Should temporarily store the `data` in memory starting from pointer `0x40`
  - ☐ Test coverage

Negative behavior

- Memory shouldn't be overwritable
  - ☐ Negative test?

Preconditions

- Assumes the memory is empty at this point

Inputs

- `data`
  - **Control**: Full control; it's passed from a parameter of the `external` function.
  - **Authorization**: No checks whatsoever into what the stream is supposed to

do or what it's doing.
    - **Impact**: Could have high impact since you're telling the `routeprocessor` what route to process for you and what commands to execute.

Function call analysis

There aren't external calls.

Function: `isNotEmpty(uint256 stream)`

Intended behavior

- Internal function
- Supposed to tell whether the current `stream` has ended or not. The `stream` basically refers to the memory pointer of the current stream.

Branches and code coverage

Intended branches

- Check whether the current position of the stream is less than the final stored position of the stream
  - ☐ Test coverage

Preconditions

- Assumes that the stream cannot be arbitrarily accessed, other than through the specific internal functions.

Inputs

- `stream`:
  - **Control**: Partial control: it's created through `createStream`
  - **Authorization**: N/A
  - **Impact**: N/A

Function call analysis

There aren't external calls.

Function: `readUint8(uint256 stream)`

Intended behavior

- Internal function
- Read the next value from the stream; assumes that it's an `uint8`

Branches and code coverage

Intended branches

- Move `1` byte to the right (the amount of bytes necessary for an `uint8`).
  - ☐ Test coverage
- Should `return` from memory the `uint8` value stored at that particular pointer in the stream.
  - ☐ Test coverage

Negative behavior

- Shouldn't allow reading something else other than an `uint8`.
  - ☐ Negative test?

Preconditions

- Assumes that the next value is indeed an `uint8`.

Inputs

- `stream`
  - **Control**: Partial control; it's created through `createStream`
  - **Authorization**: N/A
  - **Impact**: N/A

Function call analysis

There aren't external calls.

Function: `readUint(uint256 stream)`

Intended behavior

- Internal function
- Read the next value from the stream, assuming that it's an `uint`.
- The thing here is that it will never fail, even if an address is read instead of an `uint` apparently; it will get `casted`. There's also no input validation so everything is really tricky.

Branches and code coverage

Intended branches

- Return the `uint` stored at that memory pointer.
  - ☐ Test coverage
- Move `32` bytes to the right (amount necessary for `uint256` AKA `uint`)
  - ☐ Test coverage

Negative behavior

- Shouldn't allow reading a value if it's not `uint256`; this doesn't appear to be enforced.
  - ☐ Negative test?

Preconditions

- Assumes that the value that's read really is an `uint256`.

Inputs

- `stream`
  - **Control**: Partial control; it's created through `createStream`
  - **Authorization**: N/A
  - **Impact**: N/A

Function call analysis

There aren't external calls.

Function: `readAddress()`

Intended behavior

- Internal function
- Read the next value from the stream, assuming it's an `address`.

Branches and code coverage

Intended branches

- Move `20` bytes to the right, the amount necessary for an address.
  - ☐ Test coverage

Negative behavior

- Shouldn't allow reading a value if it's not an `address`; this doesn't appear to be enforced.
  - ☐ Negative test?

Preconditions

- Assumes that the value that's read really is an `address`.

Inputs

- `stream`:
  - **Control**: Partial control; it's created through `createStream`
  - **Authorization**: N/A
  - **Impact**: N/A

Function call analysis

There aren't external calls.

Function: `readBytes()`

Intended behavior

- Internal function
- Read the next value from the stream, assuming it's `bytes`.

Branches and code coverage

Intended branches

- Read 32 bytes and then move all the way to the length of the bytes object.
  - ☐ Test coverage

Negative behavior

- Shouldn't allow reading a value if it's not a `bytes`; this doesn't appear to be enforced.
  - ☐ Negative test?

Preconditions

- Assumes that the value that's ready really is of `bytes` type.

Inputs

- `stream`
  - **Control**: Partial control; it's created through `createStream`
  - **Authorization**: N/A
  - **Impact**: N/A

Function call analysis

There aren't external calls.

5.2 File: `RouteProcessor`

Function: `bentoDepositAmountFromBento(uint256 stream, address token)`

Intended behavior

- Transfers `tokens` from `BentoBox` to a pool.

Branches and code coverage

Intended branches

- Should transfer from `BentoBox` to a `pool`. There's nothing enforcing that `to` is a pool though! It can be anything!
  - ☐ Test coverage

Negative behavior

- Shouldn't allow depositing own balance of `token`.
  - ☐ Negative test?

Preconditions

- "Expected to be launched for initial liquidity distribution from user to Bento, so we know exact amount" - Sushiswap team.
- Native tokens or ERC20 tokens must be present to the bentoBox address.
- Assumes that the `address.this` has been whitelisted by the `BentoBox` to perform actions on its behalf.

Inputs

- `token`
  - **Control**: Full control; can be arbitrary token

    - **Authorization**: No checks
    - **Impact**: The address of ERC20 `token` contract can be zero address or the `bentoBox` contract should have non-zero balance of this `token`
- `to`
    - **Control**: Full control
    - **Authorization**: No checks whether this is an authorized pool or not
    - **Impact**: The receiver of tokens
- `stream`
    - **Control**: Assumed it's correct
    - **Authorization**: N/A
    - **Impact**: In case of incorrect data, the `amount` value can be parsed incorrectly

Function call analysis

- `bentoBox.deposit()`
    - **What is controllable?** `token`, `to`, `amount`
    - **If return value controllable, how is it used and how can it go wrong?** There isn't a return value here.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Can revert if `bentoBox` balance of tokens less then `amount` value.

Function: `bentodepositAllFromBento()`

Intended behavior

- Similar to `bentoDepositAmountFromBento`, but the `amount` value is calculated inside the function.

Function: `bentoWithdrawShareFromRP()`

Intended behavior

- Withdraw all Bento tokens from Bento to an address.

Branches and code coverage

Intended branches

- Withdraw a `share` of `bentoTokens` from the `bentoBox`.
    - ☐ Test coverage

Negative behavior

- No tokens should be left in there after the entire route is executed.
  - ☐ Negative test?

Preconditions

- Assumes token have been deposited on behalf of `address(this)` beforehand but in the same transaction (since no funds can be leftover).
- This contract is allowed to use funds belonging to the `from` address.

Inputs

- `token`
  - **Control**: Full control
  - **Authorization**: No checks
  - **Impact**: The address of the ERC20 tokens contract can be zero address, or this contract should have a non-zero balance of these tokens inside the `bentoBox` contract
- `stream`
  - **Control**: Full control
  - **Authorization**: No checks
  - **Impact**: in case of incorrect data, the `amount` value can be parsed incorrectly

Function call analysis

- `bentoBox.withdraw()`
  - **What is controllable?** `to` - if the address is wrong but exists, the funds may be lost, `amount`. This is prone to leeching, since if there's a case where someone leaves some tokens, they can be taken away from the contract by front-running for example. Important to keep in mind. Presumably, these types of errors are accounted for in the front end.
  - **If return value controllable, how is it used and how can it go wrong?** This function returns `amountOut` and `shareOut` values, but `caller` function ignores them.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can revert in some cases: if `to` - zero or if transfer ends with an error.

Function: `bentoWithdrawAllFromRP()`

Intended behavior

- Allows to withdraw all user's tokens from `bentoBox`.

- Same happens for this function as the `bentoWithdrawShareFromRP` function. The main difference is that `share = 0` there and `amount = 0` here.

Function: `swapTrident()`

Intended behavior

Allows to perform swap over trident pool.

Branches and code coverage

Intended branches

- The transaction will be successful if the `pool` address implements the `swap` function.
  - ☐ Test coverage
- Make sure that the result of the swap is correct.
  - ☐ Test coverage

Negative behavior

- Will revert if the tokens were not previously transferred.
  - ☐ Test coverage

Preconditions

- The `token0` and `token1` `bento` balance of the called `pool` must be replenished beforehand.

Inputs

- `swapData`
- **Control**: Controlled
  - **Authorization**: No checks
  - **Impact**: Contains the necessary information for performing the swap
- `pool`
  - **Control**: Controlled
  - **Authorization**: No checks
  - **Impact**: The address of `pool`, which implemented the IPool `swap` logic - should be trusted `pool` for user

Function call analysis

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

  - **What is controllable?** `pool`, `swapData`
  - **If return value controllable, how is it used and how can it go wrong?** There isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can be reverted as a result of an error in the transfer of funds over bentobox contract

Function: `swapUniswapPool`

Intended behavior

- Allows to perform swap over `sushi/uniswap` pool.

Branches and code coverage

Intended branches

- Only allow swapping own funds (as in, same transaction). This is the intended behavior.
  - ☑ Test coverage
- Whitelist the `pools` that can be interacted with.
  - ☐ Test coverage

Negative behavior

- Shouldn't use just any pools
  - ☐ Negative test?

Preconditions

- Assumes the `pool` is a legitimate one. Anything can be supplied really.
- Assumes that the `contract` has previously supplied collateral to the pool such that the `swap` can happen.
- Assumes that it's a total swap - `0 t0 and all t1` or `all t0 and 0 t1`.

Inputs

- `pool`
  - **Control**: Full control
  - **Authorization**: N/A
  - **Impact**: The address of `pool`, which implemented the `IUniswapV2Pair` `swap` logic - should be trusted `pool` for user
- `tokenIn`

  - **Control**: Full control
  - **Authorization**: Limited; perhaps there is some on the pool side, but not trustworthy
  - **Impact**: The address of the token to swap must match one of the tokens' addresses in the pool

Function call analysis

- `IUniswapV2Pair(pool).swap()`
  - **What is controllable?** The `pool` - can be an arbitrary one
  - **If return value controllable, how is it used and how can it go wrong?** There isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can be reverted as a result of an error in the transfer of funds over bentobox contract

Function: `distributeERC20Amounts`

Intended behavior

- Distributes input ERC20 tokens from msg.sender to addresses. Tokens should be approved.

Branches and code coverage

Intended branches

- Decrease the balance of the `msg.sender` .
  - ☐ Test coverage
- Increase the balances of all the `to` addresses by their subsequent `amount`
  - ☐ Test coverage

Negative behavior

- Shouldn't allow `msg.sender` not to send enough tokens or to trick the system by sending illegitimate ones.
  - ☐ Negative test?

Preconditions

- Assumes `msg.sender` has approved the transfer.

Inputs

- `amount`
  - **Control**: Full control
  - **Authorization**: N/A; it's assumed that `msg.sender` has approved the `contract` to transfer
  - **Impact**: N/A
- `to`
  - **Control**: Full control
  - **Authorization**: No checks
  - **Impact**: In case of wrong address, tokens can be lost
- `token`
  - **Control**: Full control
  - **Authorization**: No checks - tokens transfer must be approved for `this` contract
  - **Impact**: N/A

Function call analysis

- `IERC20(token).safeTransferFrom(msg.sender, to, amount)`
  - **What is controllable?** `token`, `amount`, `to`
  - **If return value controllable, how is it used and how can it go wrong?** There isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A

Function: `wrapAndDistributeERC20Amounts()`

Intended behavior

- Should allow `wrapping` all native tokens and distributing the `Wrapped` ERC20 tokens from `RP` to addresses.

Branches and code coverage

Intended branches

- Balance of contract should decrease by `amount`.
  - ☐ Test coverage
- Balance of other addresses should increase by `amount`.
  - ☐ Test coverage

Negative behavior

- Shouldn't allow transferring more than `msg.value`. Otherwise, it could revert or transfer leftover funds.
  - ☐ Negative test?
- Shouldn't allow transferring just any tokens, and should only use `wNATIVE`.
  - ☐ Negative test?
- Shouldn't allow anyone to front-run/leech off remaining `wnative` if they didn't supply enough `msg.value` in the first place. This should in theory be handled, since the entire swap happens in one long transaction (constructed by the Sushiswap back end).
  - ☐ Negative test?

Preconditions

- Assumes that `msg.value` is enough to cover for all the `amountTotal`. Currently there's no check put in place for that.

Inputs

- `to`
  - **Control**: Full control
  - **Authorization**: Presumably it's sent to a pool, so it should be trusted.
  - **Impact**: In case of wrong address, tokens can be lost
- `token`
  - **Control**: Full control
  - **Authorization**: There's no checks; it is necessary to check that `token == wNative`
  - **Impact**: N/A
- `amount`
  - **Control**: Full control
  - **Authorization**: There's no checks; it is necessary to check that `sharesTotal == msg.value`
  - **Impact**: N/A

Function call analysis

- `IERC20(token).safeTransfer(to, amount);`
  - **What is controllable?** `token`, `amount`, `to`
  - **If return value controllable, how is it used and how can it go wrong?** There isn't a return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A

Function: `distributeBentoShares()`

Intended behavior

- Allow distributing `input` Bento tokens from `msg.sender` to addresses.

Branches and code coverage

Intended branches

- Balances of the `to` addresses should increase.
  - ☐ Test coverage
- Balances of the `msg.sender` should decrease.
  - ☐ Test coverage
- Should be transferred to pools, but there's no enforcing this for the `to` parameters.
  - ☐ Test coverage
- Assumes that the `tokens` that are about to be transferred are legitimate.
  - ☐ Test coverage

Negative behavior

- Shouldn't allow transferring `unwanted` token to trick the system by any means.
  - ☐ Negative test?

Preconditions

- Requires that `msg.sender` has approved the amounts that are about to be transferred.

Inputs

- `share`
  - **Control**: Full control
  - **Authorization**: No checks; this will fail if `msg.sender` doesn't have enough/has not approved enough anyway
  - **Impact**: The number of shares that will be transferred
- `token`
  - **Control**: Full control
  - **Authorization**: No checks
  - **Impact**: Shares on the balance inside the `bentoBox` should be enough to transfer
- `to`
  - **Control**: Full control

    - **Authorization**: No checks
    - **Impact**: The receiver of shares

Function call analysis

- `bentoBox.transfer(token, msg.sender, to, share);`
    - **What is controllable?** `token`, `to`, `share`
    - **If return value controllable, how is it used and how can it go wrong?** There isn't return value
    - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if `to` is zero or balance of `msg.sender` is less than `share` amount

Function: `distributeERC20Shares()`

Intended behavior

- Distributes ERC20 tokens from RP to addresses.
- Shares are distributed instead of amounts to minimize slippage during swaps.

Branches and code coverage

Intended branches

- Allow transferring from this contract to `to` the amount of tokens, based on the shares that are calculated above.
    - ☐ Test coverage

Negative behavior

- Shouldn't allow the transfer of just any `ERC20`.
    - ☐ Negative test?
- Shouldn't allow transferring `unwanted` token to trick the system by any means.
    - ☐ Negative test?

Preconditions

- Assumes that the contract has enough shares to transfer.

Inputs

- `token`
    - **Control**: Full control (func param)
    - **Authorization**: No checks here; any `token` can be used
    - **Impact**: N/A

- `shares`
  - **Control**: Full control; user can transfer however much they want
  - **Authorization**: No authorization checks; anyone can call this
  - **Impact**: The amount of `shares` that uses for `amount` of tokens calculations

Function call analysis

- `IERC20(token).safeTransfer(to, amount);`
  - **What is controllable?** `token`, `to`, `amount`
  - **If return value controllable, how is it used and how can it go wrong?** There isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A

Function: `distributeBentoPortions()`

Intended behavior

- Distributes the `routeProcessor` tokens that the `Bento` holds to addresses.

Branches and code coverage

Intended branches

- Shouldn't allow leeching off these portions. It's assumed that they will be stored beforehand so technically anyone could make this call. This shouldn't, however, be possible, since no fund will be left over after a chain of commands (the entire route) is executed. This should happen if that route was generated by the `back end`.
  - ☐ Test coverage

Negative behavior

- Shouldn't allow transferring `unwanted` token to trick the system by any means.
  - ☐ Negative test?

Preconditions

- Assumes that `address.this` has previously deposited some tokens in that Bento and that there's a way to do so anyway.

Inputs

- `share / amount`

    - **Control**: Full control
    - **Authorization**: No checks here; basically anyone can transfer any value if there's enough balance
    - **Impact**: N/A
- `token`
    - **Control**: Full control to any token that can be transferred from `bentoBox`
    - **Authorization**: No checks on the token whatsoever; it's assumed it exists in `bentoBox`, should fail otherwise
    - **Impact**: The balance of tokens inside `bentoBox` should be non-zero

Function call analysis

- `bentoBox.transfer(token, address(this), to, amount);`
    - **What is controllable?** `token`, `to`, `share`
    - **If return value controllable, how is it used and how can it go wrong?** There isn't a return value
    - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if `to` is zero or balance of `address(this)` is less than `amount`

Function: `unwrapNative()`

Intended behavior

- Allow unwrapping the native token and transferring it to the `receiver`.

Branches and code coverage

Intended branches:

- Should `withdraw` the entire `wnative` balance `-1` of this contract. This means receiving the `native`.
  ☑ Test coverage
- Transfer the `native` after it has been received from the `wnative` to the `receiver` - thus, `receiver.balance += address(this).balance`.
  ☐ Test coverage

Negative behavior:

- Shouldn't allow transferring funds that have been deposited by someone else. This is covered by the "intended" usage of the code, since after a route has been executed, no funds should be left in the contract.
  ☑ Negative test?

Preconditions

- Assumes no funds will exist in either the `wNATIVE` or the contract before or after the route execution.

Inputs

- `receiver`
  - **Control**: Full control
  - **Authorization**: There's no checks; however, this is the intended functionality
  - **Impact**: Arbitrary recipient of native tokens

Function call analysis

- `payable(receiver).transfer(address(this).balance);`
  - **What is controllable?** `receiver`
  - **If return value controllable, how is it used and how can it go wrong?** N/A
  - **What happens if it reverts, reenters, or does other unusual control flow?** There should be no balance left, so reentering is not an issue. Moreover, reverting leads to the entire route execution being reverted.
- `wNATIVE.withdraw(IERC20(address(wNATIVE)).balanceOf(address(this))`
  - **What is controllable?** - `balanceOf(address(this))`
  - **If return value controllable, how is it used and how can it go wrong?** - n/a
  - **What happens if it reverts, reenters, or does other unusual control flow?** It means contract doesn't have the required funds for some reason; the entire chain of the route's commands reverts.

6 Audit Results

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

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

6.1 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.