Sushiswap
Security Review

Review by:
Milo truck, Security Researcher

August 26, 2024

Contents

1 Introduction

1.1 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.2 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.2.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

SushiSwap is a decentralized exchange which, similar to platforms like Uniswap and Balancer, uses a collection of liquidity pools to achieve this goal. Users first lock up assets into smart contracts, and traders then buy and sell cryptocurrencies from those pools, swapping out one token for another.

From Jul 19th to Jul 20th the security researchers conducted a review of mode-lock/src on commit hash e7d9e63b. A total of **4** issues in the following risk categories were identified:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 1
- Gas Optimizations: 0
- Informational: 2

3 Findings

3.1 Medium Risk

3.1.1 Routes with `applyPermit()` can be forced to revert through front-running

Severity: Medium Risk

Context: RouteProcessor5.sol#L240

Description: Including `applyPermit()` in a route calls `safePermit()` from OpenZeppelin's `SafeERC20` library:

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

`safePermit()` checks that the owner's nonce has increased after `permit()` was called, and reverts if it does not:

```
uint256 nonceBefore = token.nonces(owner);
token.permit(owner, spender, value, deadline, v, r, s);
uint256 nonceAfter = token.nonces(owner);
require(nonceAfter == nonceBefore + 1, "SafeERC20: permit did not succeed");
```

However, this allows an attacker to forcefully cause all swaps that use `applyPermit()` to revert by front-running. For example:

- User calls `processRoute()` with a route that includes `applyPermit()`.
- Attacker front-runs the user's call and calls `permit()` with the same signature.
- Afterwards, when the attacker's call is executed, `safePermit()` reverts as the signature was already used.

Although an attacker has no incentive to do this for regular swaps through the router, it could cause issues for external integrations that do not expect `processRoute()` to revert.

Recommendation: In `applyPermit()`, consider checking if the caller's allowance for the router is greater than `value` before calling `safePermit()`:

```
if (IERC20(tokenIn).allowance(msg.sender, address(this)) < value) {
    IERC20Permit(tokenIn).safePermit(msg.sender, address(this), value, deadline, v, r, s);
}
```

This ensures that `safePermit()` is only called when the permit signature has not been used.

3.2 Low Risk

3.2.1 Slippage protection can be bypassed through cross-contract reentrancy

Severity: Low Risk

Context: RouteProcessor5.sol#L222-L224

Description: In `processRouteInternal()`, slippage protection is enforced by checking that the amount of tokens received by the `to` address (i.e. `balanceOutFinal - balanceOutInitial`) is not less than `amountOutMin`:

```
uint256 balanceOutFinal = tokenOut.anyBalanceOf(to);
if (balanceOutFinal < balanceOutInitial + amountOutMin)
  revert MinimalOutputBalanceViolation(balanceOutFinal - balanceOutInitial);
```

However, if the `to` address is a contract with functionality, it might be possible to bypass slippage protection in the router. For example, consider the following naive contract with deposit and swapping functionality:

```
contract DepositAndSwap {
  address constant NATIVE_ADDRESS = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;

  function deposit(address token, uint256 amount) external payable {
    if (token == NATIVE_ADDRESS) {
      require(msg.value == amount, "Wrong amount");
    } else {
      require(msg.value == 0, "Wrong amount");
      IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
    }

    // Deposit accounting is here...
  }

  function withdraw(address token, uint256 amount) external {
    // Functionality to withdraw tokens here...
  }

  function swapEthForTokens(
    uint256 amountIn,
    address tokenOut,
    uint256 amountOutMin,
    bytes memory route
  ) external {
    routeProcessor5.processRoute(
      NATIVE_ADDRESS,
      amountIn,
      tokenOut,
      amountOutMin,
      address(this),
      route
    );
  }
}
```

Slippage protection can be bypassed as follows:

- Assume the contract has no USDC.
- Attacker calls `swapEthForTokens()` for USDC with a certain `amountOutMin`.
- In `processRoute()`:
  - `balanceOutInitial` is `0`.
  - Attacker receives an external call, either from the router or during swapping.
  - Attacker calls `deposit()` to deposit 1000 USDC, which he can withdraw later.
  - In the check, `balanceOutFinal` is the sum of 1000 USDC from the attacker's deposit and USDC received from the swap.

Essentially, by depositing to the `to` address in an external call before the `balanceOutFinal < balanceOutInitial + amountOutMin` check, an attacker can inflate `balanceOutFinal`.

Recommendation: Consider documenting that contracts integrating with the router should ensure users cannot increase the balance of the `to` address during swaps.

3.3 Informational

3.3.1 ETH balance in the router can be drained

Severity: Informational

Context: RouteProcessor5.sol#L129, RouteProcessor5.sol#L218-L220

Description: In `transferValueAndprocessRoute()`, there is no check that `amountValueTransfer` is less than `msg.value`. As such, the caller can specify `amountValueTransfer` as any value to transfer all ETH out from the router:

```
transferValueTo.transferNative(amountValueTransfer);
```

Additionally, when `tokenIn` is ETH, the check that ensures the amount of `tokenIn` used is less than `amountIn` is not performed:

```
uint256 balanceInFinal = tokenIn.anyBalanceOf(msg.sender);
if (tokenIn != Utils.NATIVE_ADDRESS)
  require(balanceInFinal + amountIn + 10 >= balanceInInitial, 'RouteProcessor: Minimal input balance
↪  violation');
```

As such, a user can also drain all ETH from the router by swapping all ETH in the contract for other tokens. Another side-effect of this change is that if a user performs a swap with `tokenIn` as ETH, he might end up receiving more `tokenOut` than expected. For example:

- Assume the router holds 1 ETH.
- User performs a swap with 10 ETH.
- In `processNative()`, `amountTotal` is 11 ETH.
- Therefore, the user receives 11 ETH worth of `tokenOut`.

If the check on `amountIn` was performed, the swap would have reverted as 11 ETH was used, instead of 10 ETH specified in `amountIn`. However, since this check has been removed, the swap no longer reverts.

Recommendation: Since the router is not expected to hold any ETH balance, not having a `amountValueTransfer <= msg.value` check in `transferValueAndprocessRoute()` is not a security risk.

However, consider documenting that swaps with `tokenIn` as ETH might use ETH stored in the router.

3.3.2 Natspec is missing `to` and `route` parameters

Severity: Informational

Context: RouteProcessor5.sol#L94-L107, RouteProcessor5.sol#L111-L128, RouteProcessor5.sol#L133-L150, RouteProcessor5.sol#L155-L172, RouteProcessor5.sol#L178-L191

Description/Recommendation: The Natspec for multiple functions are missing the `to` and `route` parameters, for example:

```
/// @notice Processes the route generated off-chain. Has a lock
/// @param tokenIn Address of the input token
/// @param amountIn Amount of the input token
/// @param tokenOut Address of the output token
/// @param amountOutMin Minimum amount of the output token
/// @return amountOut Actual amount of the output token
function processRoute(
  address tokenIn,
  uint256 amountIn,
  address tokenOut,
  uint256 amountOutMin,
  address to,
  bytes memory route
) external payable lock returns (uint256 amountOut) {
```

The natspec for the functions listed above should be amended to include both parameters.