SushiSwap Security Review

Pashov Audit Group

Conducted by: Peakbolt, ast3ros, 0xbepresent

August 15th 2024 - August 17th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **sushi-labs/tron-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About SushiSwap

SushiSwap on Tron is an automated market maker (AMM) and decentralized exchange (DEX) that enables users to trade assets directly, eliminating the need for a traditional order book to connect buyers and sellers.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - f4863c95490b87229579bdc5354b0c558eb13fbc

fixes review commit hash - 86530e9592c1605e5c51e4c18291d1d68473a197

Scope

The following smart contracts were in scope of the audit:

- `TransferHelper`
- `Math`
- `SushiSwapV2Library`
- `Ownable`
- `SushiSwapV2ERC20`
- `SushiSwapV2Factory`
- `SushiSwapV2Migrator`
- `SushiSwapV2Pair`
- `SushiSwapV2Router02`

7. Executive Summary

Over the course of the security review, Peakbolt, ast3ros, 0xbepresent engaged with SushiSwap to review SushiSwap. In this period of time a total of **5** issues were uncovered.

Protocol Summary

Protocol Name	SushiSwap
Repository	https://github.com/sushi-labs/tron-contracts
Date	August 15th 2024 - August 17th 2024
Protocol Type	DEX

Findings Count

Severity	**Amount**
Medium	2
Low	3
Total Findings	**5**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Incorrect chainId used for permit EIP712 domain separator	Medium	Resolved
[M-02]	Abuse of permit functionality to block legitimate transactions	Medium	Resolved
[L-01]	Un-used SELECTOR constant in SushiSwapV2Pair	Low	Resolved
[L-02]	Incorrect symbol name for pair token	Low	Resolved
[L-03]	Incompatibility with Solidity 0.7.0 and Above	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] Incorrect `chainId` used for permit EIP712 domain separator

Severity

Impact: Medium

Likelihood: Medium

Description

`SushiSwapV2ERC20` implements the EIP-712 `DOMAIN_SEPARATOR` that is used for `permit()`.

```
constructor() public {
        uint chainId;
        assembly {
        //@audit this should not be hardcoded
>>>         chainId := 1
        }
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256(
                    "EIP712Domain(
                      stringname,
                      stringversion,
                      uint256chainId,
                      addressverifyingContract
                    )"
                ),
                keccak256(bytes(name)),
                keccak256(bytes("1")),
                chainId,
                address(this)
            )
        );
    }
```

However, it incorrectly set the `chainId := 1`, which is for Ethereum mainnet and not TRON network.

Based on EIP-712, when the chainId does not match the active chain, the user-agent (browser/wallet) should not perform signing. This will prevent `permit()` from working as certain user agents will follow the guidelines and refuse the signing.

> uint256 chainId the EIP-155 chain id. The user-agent should refuse signing if it does not match the currently active chain.

Recommendations

```
constructor() public {
        uint chainId;
        assembly {
-            chainId := 1
+            chainId := chainid
        }
```

[M-02] Abuse of permit functionality to block legitimate transactions

Severity

Impact: Medium

Likelihood: Medium

Description

The `removeLiquidityWithPermit` function in the `SushiSwapV2Router02` contract allows users to remove liquidity from a liquidity pool by leveraging the `permit` function to approve the transfer of liquidity tokens.

However, this mechanism introduces a potential front-running vulnerability. An attacker can observe the transaction containing the `permit` call in the mempool and issue their own transaction with the same `permit` parameters but a different function call before the legitimate transaction is mined. This could result in the legitimate transaction failing due to the altered state. Similarly, the user who created the signature will end up approving tokens to `SushiSwapV2Router02` without them being used, requiring the signer to execute the transaction manually.

```
function removeLiquidityWithPermit(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline,
        bool approveMax,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external virtual override returns (uint amountA, uint amountB) {
        address pair = SushiSwapV2Library.pairFor(factory, tokenA, tokenB);
        uint value = approveMax ? uint(-1) : liquidity;
>>>     ISushiSwapV2ERC20(pair).permit(
            msg.sender,
            address(this),
            value,
            deadline,
            v,
            r,
            s
        );
>>>     (amountA, amountB) = removeLiquidity(
            tokenA,
            tokenB,
            liquidity,
            amountAMin,
            amountBMin,
            to,
            deadline
        );
    }
```

Once the `permit` is executed by an attacker, the nonce for the user will be incremented, causing the legitimate `permit` call to fail with a "INVALID_SIGNATURE" revert error. Consider the next scenario:

1. `UserA` intends to call `SushiSwapV2Router02::removeLiquidityWithPermit` with specific parameters.
2. Attacker observes the pending transaction from `UserA` and he sends a transaction to call the `permit` function with the same parameters as `UserA` transaction.
3. The `nonce` for the permit is now used.
4. When `UserA` transaction is executed, it reverts due to the invalid nonce.

Additionally, the same issue occurs in `SushiSwapV2Router02::removeLiquidityETHWithPermit` and `SushiSwapV2Router02::removeLiquidityETHWithPermitSupportingFeeOnTransferTokens`.

Recommendations

It is recommended to handle the failure in the execution of `permit` as advised by OpenZeppelin:

```
function doThingWithPermit
  (..., uint256 value, uint256 deadline, uint8 v, bytes32 r, bytes32 s) public {
    try token.permit(msg.sender, address
      (this), value, deadline, v, r, s) {} catch {}
    doThing(..., value);
}
```

8.2. Low Findings

[L-01] Un-used `SELECTOR` constant in `SushiSwapV2Pair`

`SushiSwapV2Pair` has a `SELECTOR` constant that is no longer in use as it is relying on `TransferHelper` for safe transfer. This can be resolved by removing the `SELECTOR`.

Note that this will change the bytecode and requires an update to the creation code hash in `SushiSwapV2Library.pairFor()`.

```
contract SushiSwapV2Pair is ISushiSwapV2Pair, SushiSwapV2ERC20 {
    using SafeMath for uint;
    using UQ112x112 for uint224;

    uint public constant override MINIMUM_LIQUIDITY = 10 ** 3;

-     bytes4 private constant SELECTOR =
-         bytes4(keccak256(bytes("transfer(address,uint256)")));
```

[L-02] Incorrect `symbol` name for pair token

The symbol name for the ERC20 token of V2 Pair is incorrect. It can confuse users when the symbol name is used for display on the frontend or explorer.

```
contract SushiSwapV2ERC20 is ISushiSwapV2ERC20 {
    using SafeMath for uint;

    string public constant override name = "SushiSwap V2";
-     string public constant override symbol = "SUHSI-V2";
+     string public constant override symbol = "SUSHI-V2";
    uint8 public constant override decimals = 18;
    uint public override totalSupply;
```

[L-03] Incompatibility with Solidity 0.7.0 and Above

The TransferHelper library and Sushiswap contract specify a Solidity version range from 0.6.12 to less than 0.8.0:

```
pragma solidity >=0.6.12 <0.8.0;
```

However, the implementation of the safeTransferETH function uses syntax that is incompatible with Solidity versions 0.7.0 and above. Specifically, the method for sending ETH in external calls changed in Solidity 0.7.0. Current implementation:

```
function safeTransferETH(address to, uint value) internal {
        (bool success, ) = to.call.value(value)(new bytes
        //(0)); // @audit cannot compile with solc >= 0.7
        require(success, "TransferHelper: ETH_TRANSFER_FAILED");
    }
```

This syntax for specifying the value in ETH transfers (call.value()) was deprecated in Solidity 0.7.0 in favor of a new curly brace syntax.

From solidity docs:

https://docs.soliditylang.org/en/v0.7.2/070-breaking-changes.html#changes-to-the-syntax

It's recommended to Update the safeTransferETH function to use the new syntax compatible with Solidity 0.7.0 and above:

```
function safeTransferETH(address to, uint256 value) internal {
-       (bool success, ) = to.call.value(value)(new bytes(0));
+       (bool success, ) = to.call{value: value}(new bytes(0));
        require
          (success, 'TransferHelper::safeTransferETH: ETH transfer failed');
    }
```