expelee

Building the Futuristic Blockchain Ecosystem

SECURITY AUDIT REPORT

SonicSwap Router

TOKEN OVERVIEW

Risk Findings

Severity	Found
🔴 High	2
🟠 Medium	0
🟡 Low	6
🔵 Informational	0

Centralization Risks

Owner Privileges	Description
🟢 Can Owner Set Taxes >25% ?	Detected
🟢 Owner needs to enable trading ?	Not Detected
🟢 Can Owner Disable Trades ?	Not Detected
🟢 Can Owner Mint ?	Not Detected
🟢 Can Owner Blacklist ?	Not Detected
🟢 Can Owner set Max Wallet amount ?	Not Detected
🟢 Can Owner Set Max TX amount ?	Not Detected

TABLE OF CONTENTS

OVERVIEW

The Expelee team has performed a line-by-line manual analysis and automated review of the smart contract. The smart contract was analysed mainly for common smart contract vulnerabilities, exploits, and manipulation hacks. According to the smart contract audit:

Audit Result	High Risk Detected
Audit Date	13 April 2025

CONTRACT DETAILS

Contract Address:
0x8885b3cfF909e129d9F8f75b196503F4F8B1A351

Contract Name: SonicxSwapRouter

Blockchain: Sonic

Contract Type: ERC-20

Contract Creator:
0xa4c576e2373282e94ae08ee4212f552d9555b986

Compiler Version: v0.6.6+commit.6c089d02

AUDIT METHODOLOGY

Audit Details

Our comprehensive audit report provides a full overview of the audited system's architecture, smart contract codebase, and details on any vulnerabilities found within the system.

Audit Goals

The audit goal is to ensure that the project is built to protect investors and users, preventing potentially catastrophic vulnerabilities after launch , that lead to scams and rugpulls.

Code Quality

Our analysis includes both automatic tests and manual code analysis for the following aspects:

- Exploits
- Back-doors
- Vulnerability
- Accuracy
- Readability

Tools

- Manual Review: The code has undergone a line-by-line review by the Ace team.
- BSC Test Network: All tests were conducted on the BSC Test network, and each test has a corresponding transaction attached to it. These tests can be found in the "Functional Tests" section of the report.
- Slither: The code has undergone static analysis using Slither.

VULNERABILITY CHECKS

Design Logic	Passed
Compiler warnings	Passed
Private user data leaks	Passed
Timestamps dependence	Passed
Integer overflow and underflow	Passed
Race conditions & reentrancy. Cross-function race conditions	Passed
Possible delays in data delivery	Passed
Oracle calls	Passed
Front Running	Passed
DoS with Revert	Passed
DoS with block gas limit	Passed
Methods execution permissions	Passed
Economy model	Passed
Impact of the exchange rate on the logic	Passed
Malicious event log	Passed
Scoping and declarations	Passed
Uninitialized storage pointers	Passed
Arithmetic accuracy	Passed
Cross-function race conditions	Passed
Safe Zepplin module	Passed

RISK CLASSIFICATION

When performing smart contract audits, our specialists look for known vulnerabilities as well as logical and acces control issues within the code. The exploitation of these issues by malicious actors may cause serious financial damage to projects that failed to get an audit in time. We categorize these vulnerabilities by the following levels:

High Risk

Issues on this level are critical to the smart contract's performance/functionality and should be fixed before moving to a live environment.

Medium Risk

Issues on this level are critical to the smart contract's performance/functionality and should be fixed before moving to a live environment.

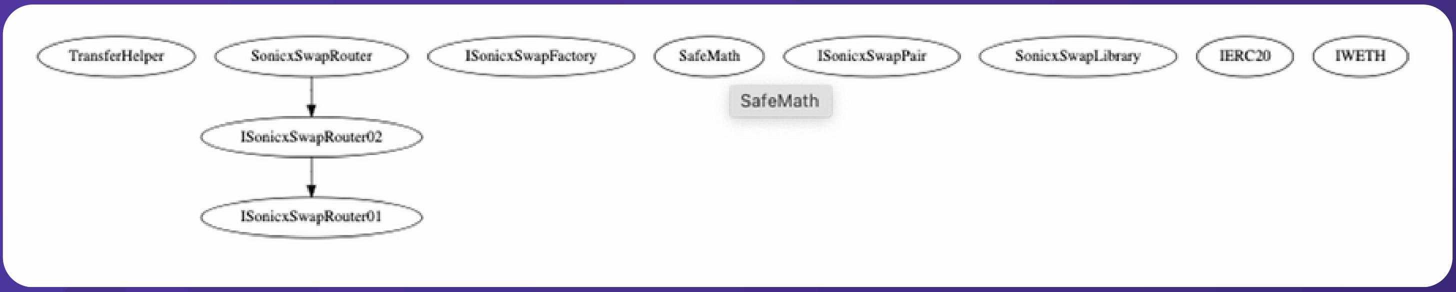
Low Risk

Issues on this level are minor details and warning that can remain unfixed.

Informational

Issues on this level are minor details and warning that can remain unfixed.

INHERITANCE TREE



POINTS TO NOTE

- The owner can change the global fee rate
- The owner can change stable pair fees
- The owner and operator can control fee changes

MANUAL REVIEW

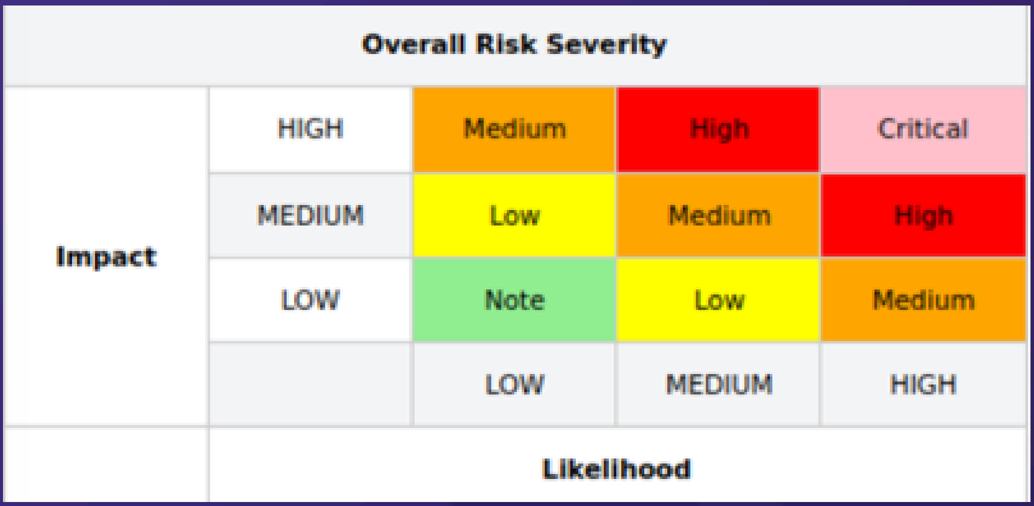
Severity Criteria

Expelee assesses the severity of disclosed vulnerabilities according to methodology based on OWASP standarts.

Vulnerabilities are dividend into three primary risk categroies:
High
Medium
Low

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious input handling
- Escalation of privileges
- Arithmetic
- Gas use



Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

HIGH RISK FINDING

Centralization – Potential Fee Calculation Logic Error

Severity: HIGH

Description:
The logic for calculating amountToSendToUser has a potential issue. When amountsOut is less than _fee, it calculates _fee - amountsOut to send to the user. This could happen if exeFee is set very high relative to the denominator (10^20). In such cases, users would unexpectedly receive tokens calculated as _fee - amountsOut rather than zero or some other appropriate amount

```
uint256 _fee = (exeFee * amountsOut) / 100000000000000000000; uint256
amountToSendToUser = amountsOut > _fee ? amountsOut - _fee : _fee - amountsOut;
```

Recommendation: Modify the fee calculation logic to handle situations where the fee exceeds the output amount.

HIGH RISK FINDING

Centralization – Uncapped Fee Setting

Severity: HIGH

Description:
There are no upper bounds on the fees that can be set by the owner or operator. This allows them to potentially set extremely high fees (even > 100%) which could effectively confiscate user funds. This represents a significant centralization risk.

```
function changeFee(uint256 _fee) external controlOrder { Fee = _fee; } function
changeStablefee(address[2] memory _address, uint256 setFee) public controlOrder {
stableFee[_address[0]][_address[1]].fee = setFee; }
```

Recommendation: Implement a maximum fee cap.

LOW RISK FINDING

Centralization – Use of assert Instead of require for External Calls

Severity: Low

Suggestion:

The contract uses assert() instead of require() to check the result of external WETH transfers. This is problematic because.

assert(IWETH(WETH).transfer(pair, amountETH));

Suggestion: Replace assert with require() statements.

LOW RISK FINDING

Centralization – Missing Zero Address Validation

Severity: Low

Description:
The contract does not validate that critical addresses (_factory, _WETH, _owneraddress, _operator) are not zero addresses. If any of these are accidentally set to the zero address during deployment, it could permanently break core contract functionality

_owneraddress, address _operator) public { factory = _factory; WETH = _WETH; Fee = _fee; owneraddress = _owneraddress; operator = _operator; }

Recommendation: Add zero address validation.

LOW RISK FINDING

Centralization – Old pragma version

Severity: Low

Description:
Standardise all pragma versions to a single fixed version (e.g., =0.6.6) to prevent compatibility issues. Floating pragmas allows contracts to be compiled with any compiler in the specified range, potentially introducing unknown bugs.

```
pragma solidity =0.6.6;
```

LOW RISK FINDING

Centralization – Missing Events for Critical State Changes

Severity: Low

Description:

The contract does not emit events when critical parameters like fees are changed. This makes it difficult to track important changes off-chain.

function changeFee(uint256 _fee) external controlOrder { Fee = _fee; } function changeStablefee(address[2] memory _address, uint256 setFee) public controlOrder { stableFee[_address[0]][_address[1]].fee = setFee; }

Recommendation: Add event emissions for fee changes.

LOW RISK FINDING

Centralization – Missing Reentrancy modifier

Severity: Low

Description:

The contract makes external calls to tokens and other contracts without implementing a reentrancy guard. While most ERC20 transfers are unlikely to allow reentrancy, some functions make calls to external contracts that could potentially be malicious

IWETH(WETH).withdraw(amountETH); TransferHelper.safeTransferETH(to, amountETH);

Recommendation: Implement a reentrancy guard for functions that make external calls, especially those that handle ETH.

LOW RISK FINDING

Centralization – Missing Reentrancy modifier

Severity: Low

Description:

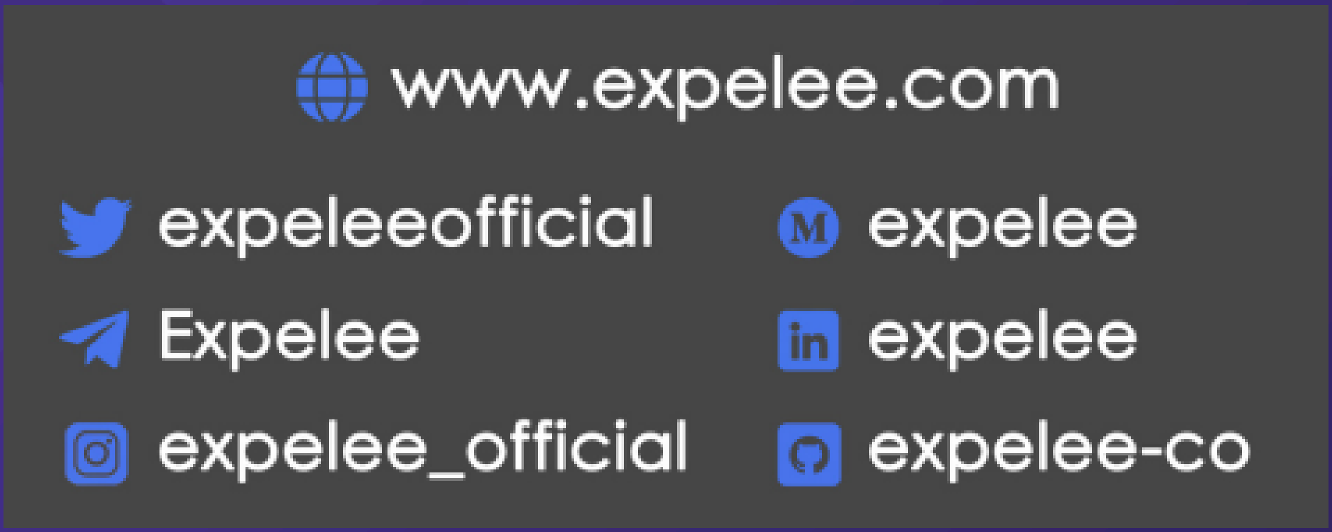
The contract makes external calls to tokens and other contracts without implementing a reentrancy guard. While most ERC20 transfers are unlikely to allow reentrancy, some functions make calls to external contracts that could potentially be malicious.

IWETH(WETH).withdraw(amountETH); TransferHelper.safeTransferETH(to, amountETH);

Recommendation: Implement a reentrancy guard for functions that make external calls, especially those that handle ETH.

ABOUT EXPELEE

Expelee is a product-based aspirational Web3 start-up. Coping up with numerous solutions for blockchain security and constructing a Web3 ecosystem from deal making platform to developer hosting open platform, while also developing our own commercial and sustainable blockchain.



www.expelee.com

expeleeofficial

Expelee

expelee_official

expelee

expelee

expelee-co

expelee

Building the Futuristic Blockchain Ecosystem

DISCLAIMER

All the content provided in this document is for general information only and should not be used as financial advice or a reason to buy any investment. Team provides no guarantess against the sale of team tokens or the removal of liquidity by the project audited in this document.

Always do your own research and project yourselves from being scammed. The Expelee team has audited this project for general information and only expresses their opinion based on similar projects and checks from popular diagnostic tools.

Under no circumstances did Expelee receive a payment to manipulate those results or change the awarding badge that we will be adding in our website. Alway do your own research and protect yourselves from scams.

This document should not be presented as a reason to buy or not buy any particular token. The Expelee team disclaims any liability for the resulting losses.

expelee

Building the Futuristic Blockchain Ecosystem