SALUS SECURITY

NOV 2022

CODE SECURITY ASSESSMENT

SHIBNOBI

Overview

Project Summary

- Name: Shibnobi
- Platform: Ethereum, BSC
- Language: Solidity

Project Dashboard

Application Summary

Name	Shibnobi
Version	v2
Type	Solidity
Dates	Nov 03 2022
Logs	Nov 03 2022, Nov 04 2022

Vulnerability Summary

Total High-Severity issues	0
Total Medium-Severity issues	0
Total Low-Severity issues	0
Total informational issues	5
Total	5

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Questionable token acquisition method	Informational	Coding Practice	Acknowledged
2	Uncoordinated variable modification method	Informational	Coding Practice	Resolved
3	Floating compiler version	Informational	Coding Practice	Acknowledged
4	Redundant functions and variables	Informational	Coding Practice	Resolved
5	Optimization	Informational	Coding Practice	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

No significant issues are found.

2.3 Informational Findings

1. Questionable token acquisition method

Severity: Informational	Category: Coding Practice
Target: - Shibnobi_v2.sol	

Description

```
function checkLiquidity() internal {
(uint256 r1, uint256 r2, ) = uniswapV2Pair(uniswapPair).getReserves();
lpTokens = balanceOf(uniswapPair);
hasLiquidity = r1 > 0 && r2 > 0 ? true : false;
}
```

It is questionable of how the “Shibnobi V2” token is acquired from the “uniswapV2Pair”. The function “(contractTokenBalance >= lpTokens * minLpBeforeSwapping / 1000)” indicates that the “lpTokens” is derived from “uniswapPair.getReserves._reserve0”. However, if other clients transfer “ShibnobiV2 token” to “uniswapPair”, it will influence the value of “lpTokens = balanceOf(uniswapPair)”.

Recommendation

Change to:

```
function checkLiquidity() internal {
(uint256 r1, uint256 r2, ) = uniswapV2Pair(uniswapPair).getReserves();
lpTokens = r1;
hasLiquidity = r1 > 0 && r2 > 0 ? true : false;
}
```

2. Uncoordinated variable modification method

Severity: Informational	Category: Coding Practice
Target: - Shibnobi_v2.sol	

Description

```
function setBridgeAddress(address a) external onlyOwner {
require(a != address(0), "Can't set 0");
bridgeAddress = a;
}.
function migrateBridge(address newAddress) external onlyOwner {
require(newAddress != address(0) && !automatedMarketMakerPairs[newAddress],
"Can't set this address");
bridgeAddress = newAddress;
isExcludedFromFee[newAddress] = true;
_limits[newAddress].isExcluded = true;
}
```

There are two different functions that can modify the value of “bridgeAddress” but they have been implemented in two different ways.

Recommendation

Consider removing the “setBridgeAddress(address a)” function and only keep the “migrateBridge(address newAddress)” function.

3. Floating compiler version

Severity: Informational	Category: Coding Practice
Target: - Shibnobi_v2.sol	

Description

```
pragma solidity ^0.8.9
```

Recommendation

It is recommended to use the up-to-date stable compiler version.

4. Redundant functions and variables

Severity: Informational	Category: Coding Practice

Target:
- Shibnobi_v2.sol

Description

```
uint256 private _burnFee;
uint256 private _previousBurnFee;
uint256 private _liquidityFee;
uint256 private _previousLiquidityFee;
uint256 private _marketingFee;
uint256 private _previousMarketingFee;
uint256 private _stakingFee;
uint256 private _previousStakingFee;
function removeAllFee() private {
if (
_burnFee == 0 &&
_liquidityFee == 0 &&
_marketingFee == 0 &&
_stakingFee == 0
) return;
_previousBurnFee = _burnFee;
_previousLiquidityFee = _liquidityFee;
_previousMarketingFee = _marketingFee;
_previousStakingFee = _stakingFee;
_burnFee = 0;
_liquidityFee = 0;
_marketingFee = 0;
_stakingFee = 0;
}
function _transfer(
address from,
address to,
uint256 amount
) internal override {
console.log("ShibnobiV2 _transfer call");
if (!tradingActive) {
console.log("tradingActive false");
require(
isExcludedFromFee[from] || isExcludedFromFee[to],
"Trading is not active yet."
);
}
checkLiquidity();
if (hasLiquidity && !inSwapAndLiquify && automatedMarketMakerPairs[to])
{
uint256 contractTokenBalance = balanceOf(address(this));
if (contractTokenBalance >= lpTokens * minLpBeforeSwapping / 1000)
takeFee(contractTokenBalance);
```

```
}
removeAllFee();
if (!isExcludedFromFee[from] && !isExcludedFromFee[to]) {
if (automatedMarketMakerPairs[from]) {
_burnFee = (amount * buyBurnFee) / feeDenominator;
_liquidityFee = (amount * buyLiquidityFee) / feeDenominator;
_marketingFee = (amount * buyMarketingFee) / feeDenominator;
_stakingFee = (amount * buyStakingFee) / feeDenominator;
}
else if (automatedMarketMakerPairs[to]) {
_burnFee = (amount * sellBurnFee) / feeDenominator;
_liquidityFee = (amount * sellLiquidityFee) / feeDenominator;
_marketingFee = (amount * sellMarketingFee) / feeDenominator;
_stakingFee = (amount * sellStakingFee) / feeDenominator;
} else {
_burnFee = (amount * transferBurnFee) / feeDenominator;
_liquidityFee =
(amount * transferLiquidityFee) /
feeDenominator;
_marketingFee =
(amount * transferMarketingFee) /
feeDenominator;
_stakingFee = (amount * transferStakingFee) / feeDenominator;
}
_handleLimited(
from,
to,
amount - _burnFee - _liquidityFee - _marketingFee - _stakingFee
);
}
uint256 _transferAmount = amount -
_burnFee -
_liquidityFee -
_marketingFee -
_stakingFee;
super._transfer(from, to, _transferAmount);
uint256 _feeTotal = _burnFee +
_liquidityFee +
_marketingFee +
_stakingFee;
if (_feeTotal > 0) {
super._transfer(from, address(this), _feeTotal);
_liquidityTokensToSwap += _liquidityFee;
_marketingFeeTokensToSwap += _marketingFee;
_burnFeeTokens += _burnFee;
_stakingFeeTokens += _stakingFee;
}
restoreAllFee();
}
function restoreAllFee() private {
_burnFee = _previousBurnFee;
_liquidityFee = _previousLiquidityFee;
_marketingFee = _previousMarketingFee;
_stakingFee = _previousStakingFee;
}
```

Recommendation

It is recommended to remove the Variables: “_burnFee, _liquidityFee, _marketingFee, _stakingFee” and functions “function restoreAllFee() private” and “function removeAllFee() private” The parts that are to be removed are labeled in red below. These are also for gas optimization purposes.

5. Optimization

Severity: Informational	Category: Coding Practice
Target: - Shibnobi_v2.sol	

Description

```
ShibnobiV2.swapTokensForETH._approve(address(this), address(uniswapRouter),
tokenAmount);

ShibnobiV2.addLiquidity._approve(address(this), address(uniswapRouter),
tokenAmount);
```

The approved procedure for “uniswapRouter” has been carried out redundantly twice in both “swapTokensForETH(uint256)” and “addLiquidity(uint256, uint256)” functions.

Recommendation

It is recommended to remove the below two original functions and add the solution function into the constructor.

```
_approve(address(this), address(uniswapRouter), type(uint256).max);
```