STAYSAFU
AUDIT

FEBRUARY 20TH, 2023

ShibaZilla

TABLE OF CONTENTS

AUDIT SUMMARY

This report was written for ShibaZilla in order to find flaws and vulnerabilities in the ShibaZilla project's source code, as well as any contract dependencies that weren't part of an officially recognized library.

A comprehensive examination has been performed, utilizing Static Analysis, Manual Review, and ShibaZilla Deployment techniques. The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors
- Assessing the codebase to ensure compliance with current best practices and industry standards
- Ensuring contract logic meets the specifications and intentions of the client
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders
- Through line-by-line manual review of the entire codebase by industry expert

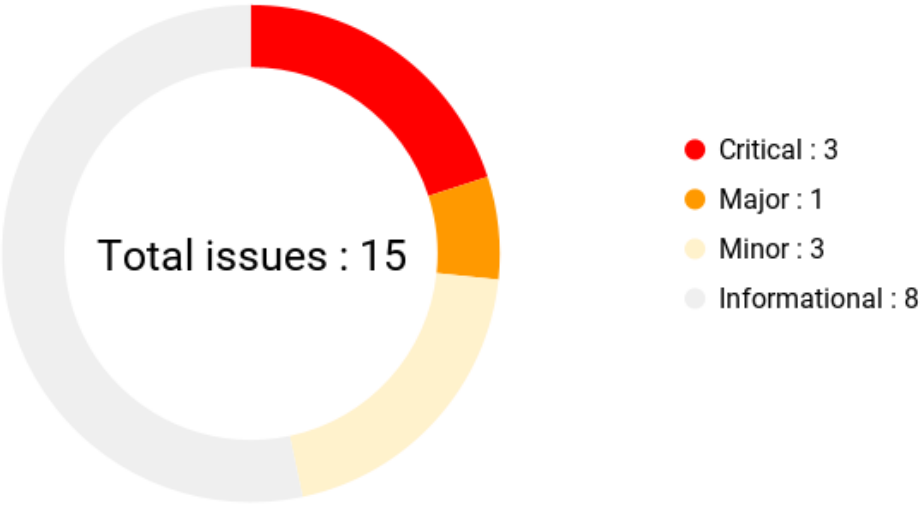
AUDIT OVERVIEW

PROJECT SUMMARY

Project name	ShibaZilla
Description	BZ EXCHANGE belongs to the US team, creating crypto currency applications including Exchange, NFT, payment and P2P platforms.
Platform	BNB Smart Chain
Language	Solidity
Codebase	https://bscscan.com/address/0x6b6689f49336bf8d2ce938c231ad022385c8aa54#code

FINDINGS SUMMARY

Vulnerability	Total
Critical	3
Major	1
Medium	0
Minor	3
Informational	8



AUDIT FINDINGS

Code	Title	Severity
TRSF-1	transferFrom logic incorrectly implemented	Critical
AMNT-1	Amounts subtracted and added to balances in _transferStandard exceed inputs	Critical
TRSF-2	Underflow revert is always thrown by _transferStandard	Critical
BLC-1	balanceOf() returns 0 for large range of balances	Major
ADDR-1	Changing dev address should use an acceptance approach	Minor

TRSF-3	Final logic branch of _transfer() does not do anything	Minor
SUPP-1	_getcurrentSupply returns the same value regardless of conditional statement	Minor
RED-1	If redisFeeonBuy and redisFeeOnSell can never be more than zero make constant and remove from setFee function	Informational
RED-2	Rewriting _redisFee and _taxFee is an expensive approach	Informational
TRSF-4	Use of _tokenTransfer sending to _transferStandard uses gas for no purpose	Informational
ROUT-1	Router and Pair variables can be immutable	Informational
INT-1	Using smaller integers for fees will reduce flexibility of percentages that can be applied	Informational
MATH-1	SafeMath does not need to be used after version 0.8.0	Informational
UINT-1	Uint256 will improve gas efficiency of lockTheSwap()	Informational
MSG-1	Use msg.sender instead of msgSender() unless expecting forwarders	Informational

TRSF-1 | transferFrom logic incorrectly implemented

Description

The best practice approach for transferFrom is to check the allowance is more than the amount being spent, subtract the amount being spent from the allowance, and then transfer the tokens. However, as the contract transfers the amount before subtracting from the allowance, the contract would be open to a re-entrancy attack where transfer can be called indefinitely draining the balance of an account with an allowance of more than 1 unit.

Recommendation

Call the _approve() function first and then call _transfer to protect against re-entrancy attacks.

AMNT-1 | Amounts subtracted and added to balances in _transferStandard exceed inputs

Description

The logic used in _transferStandard will always lead to a scenario where the amount being subtracted from the sender is vastly larger than the amount being sent. In addition, the amount being added to the receiver account would be vastly larger than the amount being sent.

This stems from the number returned from _getRate() which would be the 1678146220830669498892333116067940693525651951 using the values initialized in the construction of the contract. This error is not triggered due to an underflow risk discussed next but the logic is faulty and would need to be revised.

For example: if the amount being transferring was 100 (or 100 / 10 ** 9) then the following outputs would be returned from _getValues dues to _getRates:

- rAmount = 100 * 1678146220830669498892333116067940693525651951 = 167814622083066949889233311606794069352565195100
- rTransferAmount = 167814622083066949889233311606794069352565195100 - 335629244166133899778466623213588138705130390 2 - 0 = 164438319681435600911478745385438257795553851198

Recommendation

Revise the calculation used to add and subtract to balances in _transferStandard(). The value returned from _getRate() is the cause of

this critical issue and the following critical issues. To resolve this along with a series of other issues, the purpose of the calculation will need to be revisited and the logic will need to be changed to prevent such a large value being returned that leads to inflated values and rounding errors (covered in a later point).

TRSF-2 | Underflow revert is always thrown by _transferStandard

Description

Due to the large value being returned from _getRate(), the amount that would be subtracted from the sender in _transferStandard will always overflow. Taking the example above, the sender would being sending an amount of 0.000000001 but the amount being subtracted from their balance would be 167814622083066949889233311606794069352.565195100. This would mean that there is no way to successfully complete the _transferStandard function.

Recommendation

Review the logic used in _getRate() to prevent the number that is being returned from creating an underflow revert.

<code>

BLC-1 | balanceOf() returns 0 for large range of balances

Description

Another knock-on effect from the faulty _getRate() return value is the return that would be received when balanceOf() is called. For example if the balance of the caller is 10,000,000,000.000000 or 10,000,000,000 * 10 **9 then balanceOf would return a value of 0. This is due to the large value returned from _getRate() leading to a rounding down error, as there are no floating point numbers in Solidity this means when the number exceeds the balance in such a large way the value returned will simply return 0.

Recommendation

Review logic in _getRate().

ADDR-1 | Changing dev address should use an acceptance approach

Description

If the transferOwnership transfer uses the wrong address or a zero address then the owner role would be lost forever for the contract. The transfer ownership function should check that the input address is not a zero address and the process should be two steps. The first step would be nominating an address to take ownership and the second would allow the nominated accounts to call the acceptOwnership() function for the transfer to be a success. This will ensure that the account is active and valid that is accepting ownership of the contract.

Recommendation

Change the setNewDevAddress() function into a two step process with a transfer function and a separate accept function.

TRSF-3 | Final logic branch of _transfer() does not do anything

Description

The logic used in the final branch of the _transfer() function rewrites the redisFee and the taxFee to 0 but as they are set to 0 in the beginning of the function this logic is only needed if the values are expected to be non-zero by this point. However, the values are rewritten if from = uniswapV2Pair and to != address(uniswapRouter) or from = address(uniswapRouter) and to!= uniswapV2Pair.

As the final branch checks that the from and to do not equal uniswapV2Pair, this will mean that there are no instances where the redisFee and taxFee would be any value other than zero in this branch of the logic. In effect, the logic that is used to rewrite these values would always be rewriting values that are zero.

Recommendation

Remove the final branch of logic rewriting the values for redisFee and taxFee.

SUPP-1 | _getcurrentSupply returns the same value regardless of conditional statement

Description

The logic in _getCurrentSupply() returns the same values regardless of the condition statement that is checked. The combinations of returns are:

(rSupply, tSupply)

(_rTotal, _tTotal)

As rSupply equals _rTotal and tSupply equals _tTotal, there is no need to use the condition statement as it does not do anything.

Recommendation

Remove the condition if statement and return _rTotal and _tTotal.

```
Function _getCurrentSupply() private view returns (uint256, uint256) {
return (_rTotal, _tTotal)
}
```

RED-1 | If redisFeeonBuy and redisFeeOnSell can never be more than zero make constant and remove from setFee function

Description

In the setFee() function there are two checks in place to ensure that the values for redisFeeOnBuy and redisFeeOnSell are less than 0. If these values are always expected to be set to zero then there is no need to pass them into the function or run the check, as it will use gas that is not needed.

In addition, if these values will always be set to zero then the state variables can be set to constants.

Recommendation

Remove redisFeeonBuy and redisFeeonSell inputs from setFee(), remove the require statements, and set the state variables to constants.

```
Function setFee(
uint256 taxFeeOnBuy,
uint256 taxFeeOnSell
) public onlyDev {
require(taxFeeOnBuy < 4, “string”);
require(taxFeeOnSell < 4, “string”);
                _taxFeeOnBuy = taxFeeOnBuy;
_taxFeeOnSell = taxFeeOnSell;     }
```

RED-2 | Rewriting _redisFee and _taxFee is an expensive approach

Description

The logic in _transfer() rewrites the state variables of _redisFee and _taxFee so that they can be used later in the function when _getValues is called. Although this allows both variables to be used later in the flow without passing them between the functions, it is a much more expensive approach as storage will be written to multiple times rather than initializing a memory variable.

Instead of using the current approach, it would be recommended to remove _redisFee and _taxFee from state variables and to initialize two memory variables of the same name in the _transfer() function. These variables can then be passed down into the following functions to be used as expected.

Recommendation

Remove _redisFee and _taxFee from state variables. Initialized both variables as memory in the _transfer() function and pass the variables down into the following functions.

```
uint256 redisFee;
uint256 taxFee;

// Following the branch logic in _transfer()

_tokenTransfer(from, to, amount, redisFee, taxFee);

// _tokenTransfer function
_transferStandard(sender, recipient, amount);
```

TRSF-4 | Use of _tokenTransfer sending to _transferStandard uses gas for no purpose

Description

The logic in _tokenTransfer sends directly to _transferStandard and as _tokenTransfer() is only called in the _transfer function; there is no need for two functions to be used. Instead, the _transfer() function can call _transferStandard directly rather than adding an additional function call and this will save some gas in the process.

Recommendation

Remove the _tokenTransfer() function and directly call _transferStandard() to save the unused step and gas.

ROUT-1 | Router and Pair variables can be immutable

Description

If the values set in the Router and Pair have no logic to be overwritten they can be saved as immutable values to reduce the gas when they are used in functions.

Recommendation

Store the Router and Pair values as immutable variables to save gas in the contract.

INT-1 | Using smaller integers for fees will reduce flexibility of percentages that can be applied

Description

The fee logic throughout the contract is denominated in 10s i.e. to calculate a fee of 20% the value of 2 is used along with multiplying by 10 to get the right outcome. To improve the flexibility of fees in the contract, we would encourage the use of 100s or 1000s depending on preference to calculate fees.

For example: by using 100s the fee could be set to 21% or 22% and this could not be achieved with the current methodology that can only move in steps of 10%. Using 1000s would increase the accuracy by another decimal place and depending on preference could be adopted.

Recommendation

Use 100s or 1000s for the fee calculation logic to improve the flexibility in the range of fees that can be charged. If this is implemented then fee checks throughout the contract would also need to be scaled up accordingly i.e. setFee() would need to check if the taxFeeOnBy < 40 if 100s were used or < 400 if 1000s were used.

MATH-1 | SafeMath does not need to be used after version 0.8.0

Description

The use of SafeMath for calculations on integers is no longer required in Solidity after version 0.8.0 as underflow and overflow checks are now completed by default. This means calculates can be done without the use of this library and it can be removed from imports.

Recommendation

Remove SafeMath from imports and calculate equations without the use of SafeMath.

UINT-1 | Uint256 will improve gas efficiency of lockTheSwap()

Description

The logic used in the modifier lockTheSwap() changes the state variables for inSwap from false to true. However, as the value being changed is a bool there will be extra gas costs to pad the value to the 32 bytes slot. To improve the gas efficiency of this function, change inSwap to a uint256 and toggle the value from a starting point of 1 to 2 - the starting point of 1 is more gas efficient as it costs significantly less in Solidity to change a value from a non-zero to non-zero value than from a zero value to non-zero

Recommendation

Change inSwap to a uint256 to save gas in the modifier and set the starting point for inSwap to 1:

```
uint256 private inSwap = 1;

Modifier lockTheSwap() {
        inSwap = 2;
        _;
        inSwap = 1
};
```

MSG-1 | Use msg.sender instead of msgSender() unless expecting forwarders

Description

The msgSender() approach from Context should only be used if the msg.sender is expected to be a forwarder account and the real msg.sender has to be retrieved through other means. If this is not expected to be the case then the simple approach of using msg.sender should be adopted throughout the contract.

Recommendation

Replace the use of msgSender() with msg.sender throughout the contract unless the sender is expected to be a forwarder account,

DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement.

This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without StaySAFU's prior written consent.This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts StaySAFU to perform a security assessment.

This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance. This report should not be used in any way

to make decisions around investment or involvement with any particular project.

This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk.

StaySAFU's position is that each company and individual are responsible for their own due diligence and continuous security. StaySAFU's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or fun.