VAULT FINANCE

Smart Contract Security Audit Report

16.09.2022

Versatile Finance Audit

Helping Businesses Incubate Ideas Into Reality

info@versatile.finance

Summary

Project Name: Vault Finance

Contract Address:

Factory: 0x506CbD9C43B7c00Ed047A1fe7B21f35B332f9e57

Router: 0x80d1fac4833C746a4ad098fFcD2d845200Bbe18A

FeeFactory: 0x890b3835B1e6D806DDE665F6E4B1918dCf6af8AC

Vault: 0x48406726ca48a05dDb331aEf4e0e623B4BF1C8AD

Client contact: Vault Finance Team

Blockchain: Binance smart chain

Language: Solidity

Project website: https://thevaultfinance.com

Buy Tax: 0 - 30%

Sell Tax: 0 - 30%

Token name: Vault Finance

Token supply: 1,000,000,000,000,000

Token ticker: VFX

Decimals: 18

Dividend distributor: 0x046f8e4c1aad90851b75eab856cfdb08bfeeaa43

Contract deployer address: 0x32f1C25148DeCbdBe69E1cc2F87E0237BC34b700

Swap: 0x979a52abcd0C6ef43b3673f34760EB3594a4c583

Contract's current owner address: 0x32f1c25148decbdbe69e1cc2f87e0237bc34b700

Background

Versatile Finance was commissioned by Vault Finance Team to perform an audit of the smart contract.

https://bscscan.com/address/0x32f1c25148decbdbe69e1cc2f87e0237bc34b700

https://bscscan.com/address/0x506CbD9C43B7c00Ed047A1fe7B21f35B332f9e57

https://bscscan.com/address/0x80d1fac4833C746a4ad098fFcD2d845200Bbe18A

https://bscscan.com/address/0x890b3835B1e6D806DDE665F6E4B1918dCf6af8AC

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

What is an audit

A smart contract audit is a comprehensive review process designed to discover logical errors, security vulnerabilities, and optimization opportunities within code. The Versatile Finance manages this a step further by verifying economic logic to ensure the stability of smart contracts and highlighting privileged functionality to create a report that is easy to understand for developers and community members.

Techniques and Methods

- The code quality
- Use of best practices
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.
- Code risk issue analysis and recommendations
- Ownership privileges
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

We analyze the design patterns and structure of smart contracts. A thorough check is done to ensure the smart contract is structured in a way that will not have any issues.

Static Analysis

A static Analysis of Smart Contracts is done to identify contract vulnerabilities. In this step, a series of automated tools and manual testings are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code is done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts is completely manually analyzed line by line, and the logic is checked and compared with what's mentioned in the whitepaper to make sure everything's functioned as intended.

Gas Consumption

We check the behavior of smart contracts in production. Manual testings are done in DEXs to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Issue Categories

Every issue in this report has been assigned a severity level. There are four levels of severity and each of them has been explained below.

High severity issues

NO High severity issues found

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality and we recommend these issues be fixed before moving to a live environment.

Medium-level severity issues

NO Medium severity issues found

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems and they can still be fixed. This can put users' funds at risk and has a medium to the high probability of exploitation.

Low-level severity issues

NO Low severity issues found

Informational

NO informational issues found

These are severity four issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Centralization

3 Centralization issues found & fixed

Owner can block/unblock wallets

```
ftrace | funcSig
function toggleBlacklist(address account⬆) external onlyOwnerOrSentinel {
    _blacklist[account⬆] = !_blacklist[account⬆];
    emit Blacklisted(account⬆, _blacklist[account⬆]);
}
```

Owner can set fees without any maximum limit

```
ftrace | funcSig
function setBuyFees(uint256[] memory _fees⬆) external onlyTokenOwner {
    require(_fees⬆.length == reservedBuyFees.length + 5, "invalid fee set");
    liquidityBuyFee = _fees⬆[0];
    dividendBuyFee = _fees⬆[1];
    reflectBuyFee = _fees⬆[2];
    burnBuyFee = _fees⬆[3];
    marketingBuyFee = _fees⬆[4];
    for (uint256 i = 0; i < reservedBuyFees.length; i++) {
        reservedBuyFees[i] = _fees⬆[i + 5];
    }
}

ftrace | funcSig
function setSellFees(uint256[] memory _fees⬆) external onlyTokenOwner {
    require(_fees⬆.length == reservedSellFees.length + 5, "invalid fee set");
    liquiditySellFee = _fees⬆[0];
    dividendSellFee = _fees⬆[1];
    reflectSellFee = _fees⬆[2];
    burnSellFee = _fees⬆[3];
    marketingSellFee = _fees⬆[4];
    for (uint256 i = 0; i < reservedSellFees.length; i++) {
        reservedSellFees[i] = _fees⬆[i + 5];
    }
}
```

Owner can set max sell amount without minimum limit

```
ftrace | funcSig
function setMaxSellAmount(uint256 _amount⬆) external onlyTokenOwner {
    maxSellAmount = _amount⬆;
}
```

Owner privileges

Token

The owner can blacklist/unblock wallets from the contract

```
ftrace | funcSig
function toggleBlacklist(address account⬆) external onlyOwnerOrSentinel {
    _blacklist[account⬆] = !_blacklist[account⬆];
    emit Blacklisted(account⬆, _blacklist[account⬆]);
}
```

The owner can add/remove sentinel users

```
ftrace | funcSig
function toggleSentinel(address account⬆) external onlyOwner {
    _sentinels[account⬆] = !_sentinels[account⬆];
    emit Sentinel(account⬆, _sentinels[account⬆]);
}
```

The owner can enable trading, once enabled can not disable again

```
ftrace | funcSig
function enableTrading() external onlyOwner {
    require(!tradingEnabled, "Trading has already been enabled");

    tradingEnabled = true;
    emit TradingEnabled();
}
```

The owner can add/remove wallets, who can do transfers before start trading

```
ftrace | funcSig
function setCanTransferBeforeTrading(address account⬆, bool status⬆)
    external
    onlyOwner
{
    _canTransferBeforeTradingIsEnabled[account⬆] = status⬆;
}
```

The owner can change minimum distribution period and minimum distribution amount

```
ftrace | funcSig
function setDistributionCriteria(
    uint256 _minPeriod⬆,
    uint256 _minDistribution⬆
) external onlyOwner {
    dividendDistributor.setDistributionCriteria(
        _minPeriod⬆,
        _minDistribution⬆
    );
}
```

The owner can change distribution gas limit maximum up to 750000

```
ftrace | funcSig
function setDistributorSettings(uint256 gas⬆) external onlyOwner {
    require(gas⬆ < 750000, "Gas must be lower than 750000");
    distributorGas = gas⬆;
}
```

The owner can include/exclude wallets from rewards

```
ftrace | funcSig
function setIsDividendExempt(address holder⬆, bool exempt⬆)
    external
    onlyOwner
{
    require(holder⬆ != address(this), "Holder can't be token");
    isDividendExempt[holder⬆] = exempt⬆;

    if (exempt⬆) {
        dividendDistributor.setShare(holder⬆, 0);
    } else {
        dividendDistributor.setShare(holder⬆, balanceOf(holder⬆));
    }
}
```

Router

The owner can whitelist tokens from the router, only whitelisted tokens can use the router

```
ftrace | funcSig
function setWhitelist(address _addr, bool _flag) external onlyOwner {
    whitelist[_addr] = _flag;
}
```

The owner can enable router to public trading

```
ftrace | funcSig
function enablePublicTrading(bool _flag) external onlyOwner {
    isPublicTrading = _flag;
}
```

Factory

The owner can change fees setter address

```
ftrace | funcSig
function setFeeToSetter(address _feeToSetter⬆) external {
    require(msg.sender == feeToSetter, "Novation: FORBIDDEN");
    feeToSetter = _feeToSetter⬆;
}
```

The owner can add/remove tokens

```
ftrace | funcSig
function addToken(address _token⬆) external {
    require(msg.sender == feeToSetter, "Novation: FORBIDDEN");
    require(!tokens.contains(_token⬆), "Novation: FORBIDDEN");
    tokens.add(_token⬆);
}

ftrace | funcSig
function removeToken(address _token⬆) external {
    require(msg.sender == feeToSetter, "Novation: FORBIDDEN");
    require(tokens.contains(_token⬆), "Novation: FORBIDDEN");
    tokens.remove(_token⬆);
}
```

Fee Factory

The owner can distribute collected fees

```
ftrace | funcSig
function distribute() external onlyTokenOwner {
    _distribute();
}
```

The owner can change all buy and sell fees

```
ftrace | funcSig
function setBuyFees(uint256[] memory _fees⬆) external onlyTokenOwner {
    require(_fees⬆.length == reservedBuyFees.length + 5, "invalid fee set");
    liquidityBuyFee = _fees⬆[0];
    dividendBuyFee = _fees⬆[1];
    reflectBuyFee = _fees⬆[2];
    burnBuyFee = _fees⬆[3];
    marketingBuyFee = _fees⬆[4];
    for (uint256 i = 0; i < reservedBuyFees.length; i++) {
        reservedBuyFees[i] = _fees⬆[i + 5];
    }
}

ftrace | funcSig
function setSellFees(uint256[] memory _fees⬆) external onlyTokenOwner {
    require(_fees⬆.length == reservedSellFees.length + 5, "invalid fee set");
    liquiditySellFee = _fees⬆[0];
    dividendSellFee = _fees⬆[1];
    reflectSellFee = _fees⬆[2];
    burnSellFee = _fees⬆[3];
    marketingSellFee = _fees⬆[4];
    for (uint256 i = 0; i < reservedSellFees.length; i++) {
        reservedSellFees[i] = _fees⬆[i + 5];
    }
}
```

The owner can change token owner

```
ftrace | funcSig
function updateTokenOwner(address _owner⬆) external onlySwapper {
    require(_owner⬆ != address(0), "invalid owner");
    tokenOwner = _owner⬆;
}
```

The owner can change the fees distribution method

```
ftrace | funcSig
function setIsManual(bool _flag⬆) external onlyTokenOwner {
    isManual = _flag⬆;
}
```

The owner can set minimum token amount to perform auto distribution

```
ftrace | funcSig
function setMinForAutoDistribution(uint256 _amount⬆)
    external
    onlyTokenOwner
{
    minForAutoDistribution = _amount⬆;
}
```

The owner can change liquidity receiver and marketing wallet address

```
ftrace | funcSig
function setLiquidityReceiver(address _wallet⬆) external onlyTokenOwner {
    liquidityReceiver = _wallet⬆;
}

ftrace | funcSig
function setMarketingWallet(address _wallet⬆) external onlyTokenOwner {
    marketingWallet = _wallet⬆;
}
```

The owner can update dividend tracker address

```
ftrace | funcSig
function setDividendTracker(address _tracker⬆) external onlyTokenOwner {
    dividendTracker = _tracker⬆;
}
```

The owner can add reserved wallets

```
ftrace | funcSig
function setReservedWallet(uint256 index, address _wallet)
    external
    onlyTokenOwner
{
    require(index < reservedBuyFees.length, "invalid index");
    reservedWallets[index] = _wallet;
}
```

The owner can change fees to reserved wallets

```
function addReservedFee(
    uint256 _buyFee,
    uint256 _sellFee,
    address _wallet
) external onlyTokenOwner {
    reservedBuyFees.push(_buyFee);
    reservedSellFees.push(_sellFee);
    reservedWallets.push(_wallet);
}
```

The owner can change max sell amount

```
ftrace | funcSig
function setMaxSellAmount(uint256 _amount) external onlyTokenOwner {
    maxSellAmount = _amount;
}
```

The owner can include/exclude wallets from fees

```
ftrace | funcSig
function excludeFee(address _addr⬆, bool _flag⬆) external onlyTokenOwner {
    isFeeExempt[_addr⬆] = _flag⬆;
}
```

The owner can get stuck BNB balance in the contract

```
ftrace | funcSig
function getInStuck() external onlyTokenOwner {
    require(
        address(this).balance > collectedBuyFees.add(collectedSellFees),
        "no stucked"
    );
    (bool success, ) = payable(msg.sender).call{
        value: address(this).balance.sub(collectedBuyFees).sub(
            collectedSellFees
        ),
        gas: 30000
    }("");
}
```

Audit Results

Vulnerability Category	Status
Arbitrary Jump/Storage Write	pass
BRC20 Token standards	pass
Compiler errors	pass
Latest compiler version	pass
Authorization of function call to untrusted contract	pass
Dependence on Predictable Variables	pass
Ether/Token Theft	pass
Gas consumption	pass
Safemath features	pass
Fallback usage	pass
Deprecated items	pass
Redundant code	pass
Overriding variables	pass
Flash Loans	pass
Front Running	pass
Improper Events	pass
Improper Authorization Scheme	pass
Integer Over/Underflow	pass
Business logic issues	pass

Orcle issues	pass
Race Conditions	pass
Reentrancy	pass
Signature Issues	pass
Unbounded Loops	pass
Unused Code	pass
Pseudo random number generator (PRNG)	pass
Fake deposit	pass

Audit conclusion

Versatile Finance team has performed in-depth testing, line by line manual code review, and automated audit of the smart contract. The smart contract was analysed mainly for common smart contract vulnerabilities, exploits, manipulations, and hacks. According to the smart contract audit.

Smart contract functional Status: **PASS**

Number of risk issues: **0**

Solidity code functional issue level: **PASS**

Number of owner privileges: **23**

Centralization risk correlated to the active owner: **LOW**

Smart contract active ownership: **YES**

Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice as of the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Versatile Finance and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Versatile Finance) owe no duty of care towards you or any other person, nor does Versatile Finance make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and Versatile Finance hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Versatile Finance hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Versatile Finance, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed.