WOJAK

Smart Contract Review

Deliverable: Smart Contract Audit Report

Security Report

September 2021

Introduction

Given the opportunity to review WOJAK Project's smart contract source code, we in the report outline our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is ready to launch after resolving the mentioned issues, there are no critical or high issues found related to business logic, security or performance.

About WOJAK: -

Item	Description
Issuer	WOJAK
Website	www.woj.finance
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 17, 2021

The Test Method Information: -

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open-source code, non-open-source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant effect on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project party should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.

The Full List of Check Items:

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	MONEY-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead of Transfer
	Costly Loop
	(Unsafe) Use of Untrusted Libraries
	(Unsafe) Use of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
	Business Logics Review

Advanced DeFi Scrutiny	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Common Weakness Enumeration (CWE) Classifications Used in This Audit:

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.

Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an ex pilotable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

Findings

Summary

Here is a summary of our findings after analyzing the WOJAK's Smart Contract. During the first phase of our audit, we studied the smart contract sourcecode and ran our in-house static code analyzer through the Specific tool. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	No. of Issues
Critical	1(Resolved/Acknowledged)
High	0
Medium	0
Low	0
Total	1

We have so far identified that there are potential issues with severity of **1 Critical, 0 High, 0 Medium, and 0 Low**. Overall, these smart contracts are well- designed and engineered, though the implementation can be improved and bug free by common recommendations given under POCs.

Functional Overview

($) = payable function # = non-constant function	[Pub] public [Ext] external [Prv] private [Int] internal

+ LiquiditySniper (Ownable)
  - [Ext] liquidifyAndBuy ($)
  - [Prv] addLiquidity #
  - [Prv] buyToken #
  - [Prv] getAmountOutMin
  - [Pub] withdraw #
  - [Pub] deposit ($)
  - [Pub] getBalance
  - [Ext] <Fallback> ($)

Detailed Results

Issues Checking Status

1. Unprotected Ether Withdrawal

- SWC ID:105
- Severity: Critical
- Location: LiquiditySniper.sol
- Relationships: CWE-284: Improper Access Control
- Description: Any sender can withdraw the Ether from the contract account. If it's not set on purpose, any arbitrary senders other than the contract creator can profitably extract Ether from the contract account. Verify the business logic carefully and make sure that the appropriate security controls are in place to prevent unexpected loss of funds.

```
113
114       function withdraw() public {
115           payable(msg.sender).transfer(address(this).balance);
116       }
117
```

- Remediations: Implement controls so withdrawals can only be triggered by authorized parties or according to the specs of the smart contract system.
- Resolved: After the first phase of Audit, this issue was discussed with the WOJAK's dev team, and they Resolved it before the deployment of contract at mainnet.

Automated tool Analysis

Slither: -

```
Different versions of Solidity is used:
        - Version used: ['>=0.4.22<0.9.0', '^0.8.0']
        - ^0.8.0 (Context.sol#2)
        - ^0.8.0 (IERC20.sol#2)
        - >=0.4.22<0.9.0 (New.sol#2)
        - ^0.8.0 (Ownable.sol#2)
        - >=0.4.22<0.9.0 (Uniswap.sol#2)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used

Context._msgData() (Context.sol#18-13) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

Pragma version^0.8.0 (Context.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (IERC20.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version>=0.4.22<0.9.0 (New.sol#2) is too complex
Pragma version^0.8.0 (Ownable.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version>=0.4.22<0.9.0 (Uniswap.sol#2) is too complex
solc-0.8.0 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Parameter LiquiditySniper.buyToken(address,uint256,uint256)._amountOutMin (New.sol#81) is not in mixedCase
Parameter LiquiditySniper.getAmountOutMin(address,uint256)._tokenOut (New.sol#96) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions

Redundant expression "this (Context.sol#11)" inContext (Context.sol#4-14)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements

Variable LiquiditySniper.liquidityAndBuy(address,uint256,uint256,uint256)._amountOutMin (New.sol#54) is too similar to LiquiditySniper.getAmountOutMin
(address,uint256).amountOutMins (New.sol#106-109)
Variable LiquiditySniper.buyToken(address,uint256,uint256)._amountOutMin (New.sol#81) is too similar to LiquiditySniper.getAmountOutMin(address,uint25
6).amountOutMins (New.sol#106-109)
Variable IUniswapV2Router.addLiquidity(address,address,uint256,uint256,uint256,uint256,address,uint256).amountADesired (Uniswap.sol#47) is too similar
 to IUniswapV2Router.addLiquidity(address,address,uint256,uint256,uint256,uint256,address,uint256).amountBDesired (Uniswap.sol#48)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-are-too-similar

LiquiditySniper.FACTORY (New.sol#11) is never used in LiquiditySniper (New.sol#9-128)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-state-variable
```

```
LiquiditySniper.FACTORY (New.sol#11) is never used in LiquiditySniper (New.sol#9-128)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-state-variable

withdraw() should be declared external:
        - LiquiditySniper.withdraw() (New.sol#114-116)
deposit(uint256) should be declared external:
        - LiquiditySniper.deposit(uint256) (New.sol#118-121)
getBalance() should be declared external:
        - LiquiditySniper.getBalance() (New.sol#123-125)
renounceOwnership() should be declared external:
        - Ownable.renounceOwnership() (Ownable.sol#42-44)
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address) (Ownable.sol#47-50)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
```

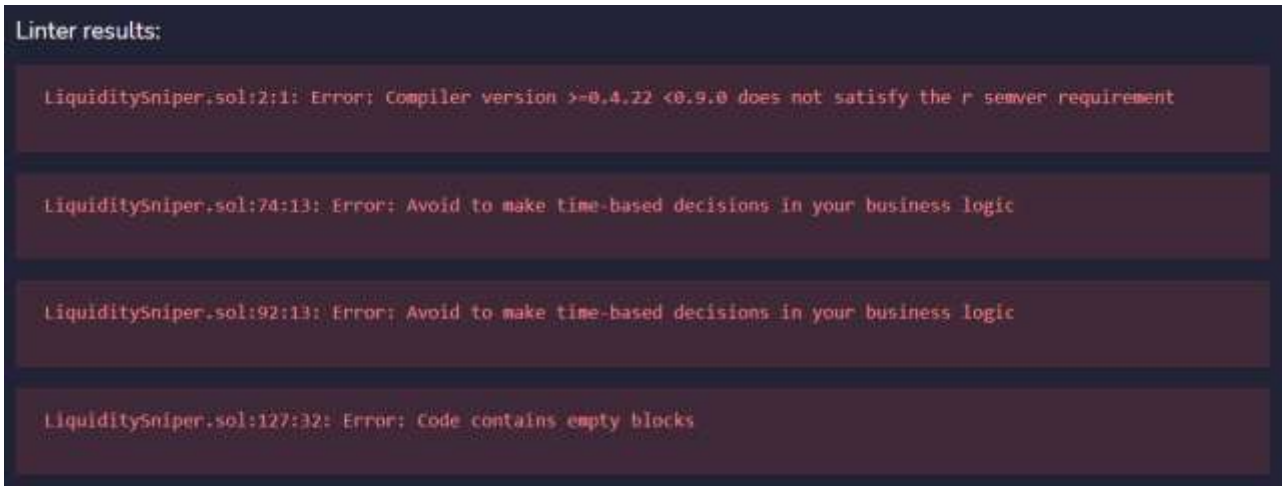
MythX: -

Line	SWC Title	Severity	Short Description
2	(SWC-103) Floating Pragma	Low	A floating pragma is set.
9	(SWC-123) Requirement Violation	Low	Requirement violation.
31	(SWC-123) Requirement Violation	Low	Requirement violation.

Mythril: -

```
root@sv-VirtualBox:/home/sv/Wojak/New1# myth analyze New.sol
The analysis was completed successfully. No issues were detected.
```

SolHint: -



Basic Coding Bugs

1. **Constructor Mismatch**
   - Description: Whether the contract name and its constructor are not identical to each other.
   - Result: PASSED
   - Severity: Critical
2. **Ownership Takeover**
   - Description: Whether the set owner function is not protected.
   - Result: PASSED
   - Severity: Critical
3. **Redundant Fallback Function**
   - Description: Whether the contract has a redundant fallback function.
   - Result: PASSED
   - Severity: Critical
4. **Overflows & Underflows**
   - Description: Whether the contract has general overflow or underflow vulnerabilities
   - Result: PASSED
   - Severity: Critical
5. **Reentrancy**
   - Description: Reentrancy is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
   - Result: PASSED
   - Severity: Critical
6. **MONEY-Giving Bug**
   - Description: Whether the contract returns funds to an arbitrary address.
   - Result: PASSED
   - Severity: Critical

7. **Blackhole**

   - Description: Whether the contract locks ETH indefinitely: merely in without out.
   - Result: PASSED
   - Severity: High

8. **Unauthorized Self-Destruct**

   - Description: Whether the contract can be killed by any arbitrary address.
   - Result: PASSED
   - Severity: Medium

9. **Revert DoS**

   - Description: Whether the contract is vulnerable to DoS attack because of unexpected revert.
   - Result: PASSED
   - Severity: Medium

10. **Unchecked External Call**

    - Description: Whether the contract has any external call without checking the return value.
    - Result: PASSED
    - Severity: Medium

11. **Gasless Send**

    - Description: Whether the contract is vulnerable to gasless send.
    - Result: PASSED
    - Severity: Medium

12. **Send Instead of Transfer**

    - Description: Whether the contract uses send instead of transfer.
    - Result: PASSED
    - Severity: Medium

13. **Costly Loop**

- Description: Whether the contract has any costly loop which may lead to Out-Of-Gas exception.
- Result: PASSED
- Severity: Medium

14. **(Unsafe) Use of Untrusted Libraries**

- Description: Whether the contract use any suspicious libraries.
- Result: PASSED
- Severity: Medium

15. **(Unsafe) Use of Predictable Variables**

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: PASSED
- Severity: Medium

16. **Transaction Ordering Dependence**

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: PASSED
- Severity: Medium

17. **Deprecated Uses**

- Description: Whether the contract use the deprecated tx.origin to perform the authorization.
- Result: PASSED
- Severity: Medium

Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: PASSED
- Severity: Critical

Conclusion

In this audit, we thoroughly analyzed WOJAK's Smart Contract. The current code base is well organized but there are promptly there is Critical Type issue found in the first phase of Smart Contract Audit, which is Resolve by the WOJAK's dev team before deploying the contract at mainnet.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.