



Smart Contract Security Audit Report





Contents

1. Executive Summary.....	1
2. Audit Methodology.....	2
3. Project Background.....	3
3.1 Project Introduction.....	3
3.2 Project Structure.....	4
3.3 Contract Structure.....	4
4. Code Overview.....	4
4.1 Main Contract address.....	4
4.2 Contracts Description.....	4
4.3 Code Audit.....	11
4.3.1 Medium vulnerabilities.....	11
5. Audit Result.....	12
5.1 Conclusion.....	12
6. Statement.....	12

1. Executive Summary

On Feb. 18, 2021, the SlowMist security team received the Dogeswao team's security audit application for Dogeswap, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

SlowMist Smart Contract DeFi project test method:

Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code module 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.

SlowMist Smart Contract DeFi project risk level:

Critical vulnerabilities	Critical vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High-risk vulnerabilities	High-risk vulnerabilities will affect the normal operation of DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium-risk vulnerabilities	Medium vulnerability will affect the operation of DeFi project. It is recommended to fix medium-risk vulnerabilities.

Low-risk vulnerabilities	Low-risk vulnerabilities may affect the operation of DeFi project in certain scenarios. It is suggested that the project party should evaluate and consider whether these vulnerabilities need to be fixed.
Weaknesses	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Enhancement Suggestions	There are better practices for coding or architecture.

2. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and in-house automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Reentrancy attack and other Race Conditions
- Replay attack
- Reordering attack
- Short address attack
- Denial of service attack
- Transaction Ordering Dependence attack
- Conditional Completion attack
- Authority Control attack
- Integer Overflow and Underflow attack



- TimeStamp Dependence attack
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Explicit visibility of functions state variables
- Logic Flaws
- Uninitialized Storage Pointers
- Floating Points and Numerical Precision
- tx.origin Authentication
- "False top-up" Vulnerability
- Scoping and Declarations

3. Project Background

3.1 Project Introduction

Dogewap is an Ethereum-based protocol designed to facilitate automatic exchange transactions between ETH and ERC20 token digital assets, and automatically provide liquidity on Ethereum/Heco.

Audit version file information

Initial audit files:

SHA256(Contracts.zip)=

c854afeb9f22ac89cfe35a36f5e8ef5e2510e7b534065a70f2db51e3c361e353

Final audit files:

SHA256(V1.0Contracts.zip)=

80b1bc8bfe9756a5d18f1b0c2bedf362f11daa2e066f305a812469335d0e68a0



3.2 Project Structure

├── Factory.sol
└── Router.sol

3.3 Contract Structure

The Dogeswap project is mainly divided into three parts, namely the trading pair contract, the factory contract and the exchange contract. The transaction pair contract is developed based on the ERC2.0 standard, mainly for transaction liquidity management contracts. Factory contract performs transaction rate management, joint construction transaction pair contract, migration configuration, etc. The exchange contract selects the best transaction path based on the information input by the user, performs token exchange, adds liquidity, and so on.

4. Code Overview

4.1 Main Contract address

The contract has been deployed on the mainnet:

Factory: 0x0419082bb45f47Fe5c530Ea489e16478819910F3

Router: 0x539A9Fbb81D1D2DC805c698B55C8DF81cbA6b35

4.2 Contracts Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

UniswapV2Pair			
Function Name	Visibility	Mutability	Modifiers
getReserves	Public view	-	-
initialize	External	Can modify state	-
mint	External	Can modify state	lock
burn	External	Can modify state	lock
swap	External	Can modify state	lock
skim	External	Can modify state	lock
sync	External	Can modify state	lock
transfer	External	Can modify state	-
transferFrom	External	Can modify state	-
approve	External	Can modify state	-
permit	External	Can modify state	-

UniswapV2Factory			
Function Name	Visibility	Mutability	Modifiers
allPairsLength	External view	-	-

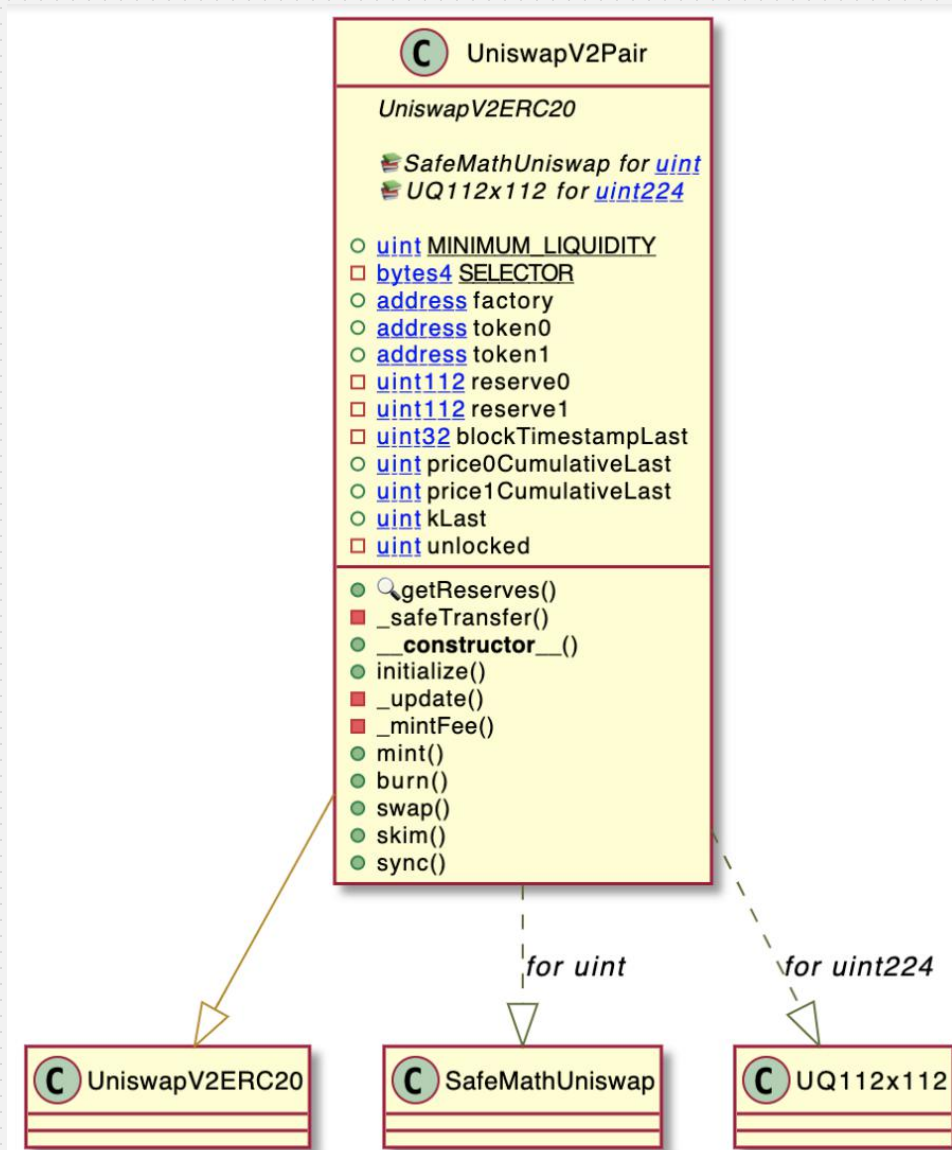
pairCodeHash	External pure	-	-
createPair	External	Can modify state	-
setFeeTo	External	Can modify state	-
setFeeRate	External	Can modify state	-
setMigrator	External	Can modify state	-
setFeeToSetter	External	Can modify state	-

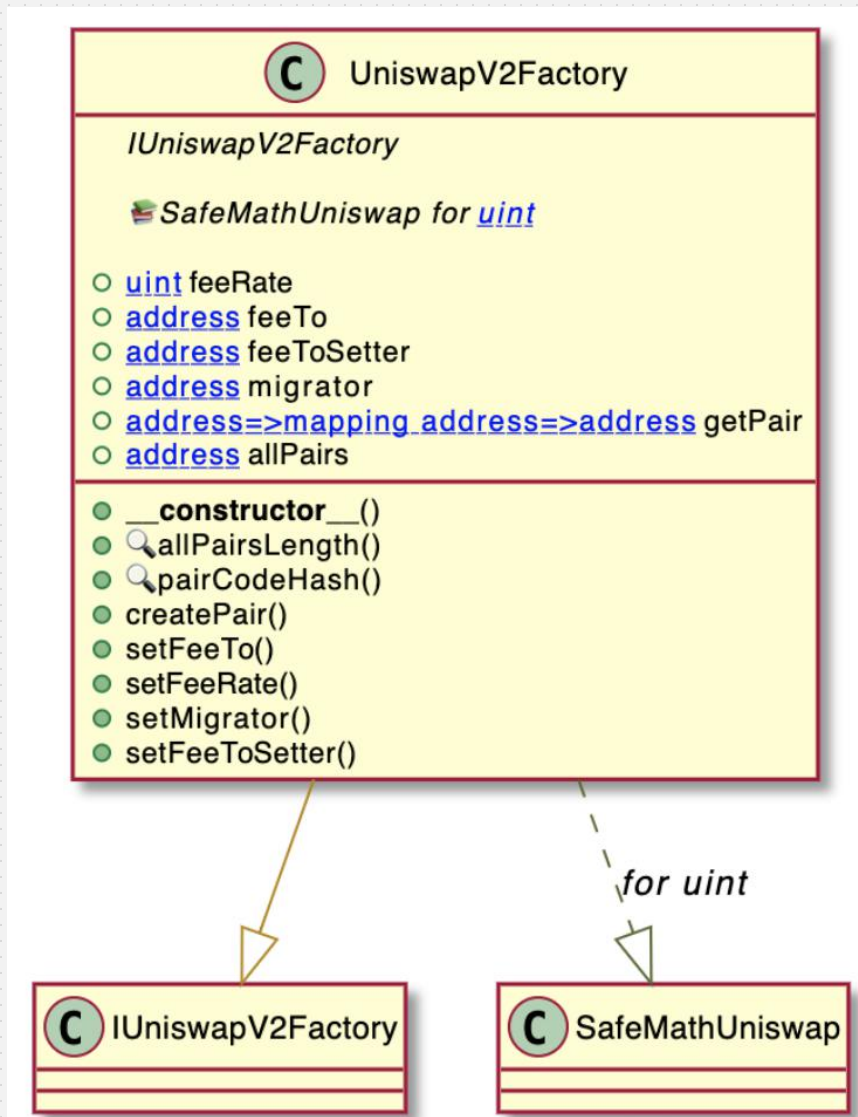
UniswapV2Router02			
Function Name	Visibility	Mutability	Modifiers
addLiquidity	external virtual	Can modify state	ensure
addLiquidityETH	External pure	payable	ensure
removeLiquidity	public virtual	Can modify state	ensure
removeLiquidityETH	public virtual	payable	ensure
removeLiquidityWithPermit	external virtual	Can modify state	-
removeLiquidityETHWithPermit	external virtual	Can modify state	-
removeLiquidityETHSupportingFeeOnTransferTokens	public virtual	Can modify state	ensure
removeLiquidityETHWithPermitSupportingFeeOnTransferTokens	external virtual	Can modify state	-

swapExactTokensForTokens	external virtual	Can modify state	ensure
swapTokensForExactTokens	external virtual	payable	ensure
swapTokensForExactETH	external virtual	Can modify state	ensure
swapExactTokensForETH	external virtual	Can modify state	ensure
swapETHForExactTokens	external virtual	Can modify state	ensure
swapExactTokensForTokensSupportingFeeOnTransferTokens	external virtual	Can modify state	ensure
swapExactETHForTokensSupportingFeeOnTransferTokens	external virtual	payable	ensure
swapExactTokensForETHSupportingFeeOnTransferTokens	external virtual	Can modify state	ensure
quote	public pure	-	
getAmountOut	public view	-	
getAmountIn	public view	-	
getAmountsOut	public view	-	
getAmountsIn	public view	-	

Analysis of the main contract structure:

Factory.sol

















Router.sol


UniswapV2Router02

IUniswapV2Router02

 *SafeMathUniswap* for *uint*

- [address](#) factory
- [address](#) WETH
-  **__constructor__()**
- ◆ **_addLiquidity()**
- **addLiquidity()**
-  **addLiquidityETH()**
- **removeLiquidity()**
- **removeLiquidityETH()**
- **removeLiquidityWithPermit()**
- **removeLiquidityETHWithPermit()**
- **removeLiquidityETHSupportingFeeOnTransferTokens()**
- **removeLiquidityETHWithPermitSupportingFeeOnTransferTokens()**
- ◆ **_swap()**
- **swapExactTokensForTokens()**
- **swapTokensForExactTokens()**
-  **swapExactETHForTokens()**
- **swapTokensForExactETH()**
- **swapExactTokensForETH()**
-  **swapETHForExactTokens()**
- ◆ **_swapSupportingFeeOnTransferTokens()**
- **swapExactTokensForTokensSupportingFeeOnTransferTokens()**
-  **swapExactETHForTokensSupportingFeeOnTransferTokens()**
- **swapExactTokensForETHSupportingFeeOnTransferTokens()**
-  **quote()**
-  **getAmountOut()**
-  **getAmountIn()**
-  **getAmountsOut()**
-  **getAmountsIn()**

 *IUniswapV2Router02*

 *SafeMathUniswap*

for uint

4.3 Code Audit

4.3.1 Medium vulnerabilities

4.3.1.1 Risk of Excessive Authority

The migrator can migrate the liquidity in the contract to other contracts to obtain the assets of the liquidity provider.

Code location: Factory.sol

```
function mint(address to) external lock returns (uint liquidity) {
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
    uint balance0 = IERC20Uniswap(token0).balanceOf(address(this));
    uint balance1 = IERC20Uniswap(token1).balanceOf(address(this));
    uint amount0 = balance0.sub(_reserve0);
    uint amount1 = balance1.sub(_reserve1);

    bool fee0n = _mintFee(_reserve0, _reserve1);
    uint _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update in
    _mintFee
    if (_totalSupply == 0) {
        address migrator = IUniswapV2Factory(factory).migrator();
        if (msg.sender == migrator) {
            liquidity = IMigrator(migrator).desiredLiquidity();
            require(liquidity > 0 && liquidity != uint256(-1), "Bad desired liquidity");
        } else {
            require(migrator == address(0), "Must not have migrator");
            liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
            _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
        }
    } else {
        liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul(_totalSupply) / _reserve1);
    }
    require(liquidity > 0, 'UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED');
```

```
_mint(to, liquidity);

_update(balance0, balance1, _reserve0, _reserve1);
if (fee0n) kLast = uint(reserve0).mul(reserve1); // reserve0 and reserve1 are up-to-date
emit Mint(msg.sender, amount0, amount1);
}
```

Fix status: The project team removed the liquidity migration logic.

5. Audit Result

5.1 Conclusion

Audit Result : PASSED

Audit Number : 0X002102240001

Audit Date : Feb. 24, 2021

Audit Team : SlowMist Security Team

Summary conclusion: The SlowMist security team use a manual and SlowMist Team analysis tool audit of the codes for security issues. There are eleven security issues found during the audit. There are one medium-risk vulnerabilities. After communicating with the project party and confirming that the risks found in the audit process have been fixed or are within the acceptable range.

6. Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility base on these.



For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the issuance of this report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.



SLOWMIST

Official Website

www.slowmist.com



E-mail

team@slowmist.com



Twitter

[@SlowMist_Team](https://twitter.com/SlowMist_Team)



Github

<https://github.com/slowmist>