WSwap

Smart Contract Audit Report Prepared for Wault Finance

Date Issued:	May 20, 2021
Project ID:	AUDIT2021001
Version:	2.0
Confidentiality Level:	Public

inspex

CYBERSECURITY PROFESSIONAL SERVICE

Table of Contents

1. Executive Summary

As requested by Wault Finance, Inspex team conducted an audit to verify the security posture of the WSwap smart contracts between May 12, 2021 and May 17, 2021. During the audit, Inspex team examined all smart contracts and the overall operation in the scope to understand the overview of WSwap smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found, and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 medium, 1 low, 3 very low, and 2 info-severity vulnerabilities. With the mitigation solutions confirmed by the project team, only 1 low, 2 very low, and 2 info-severity issues are left unresolved. Therefore, Inspex trusts that WSwap smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

This smart contract passes Inspex's security verification standard, and is trustworthy.

Approved by Inspex on: 17 May 2021

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inpex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

WSwap is an Automated Market Maker (AMM) protocol that is forked from Uniswap V2 and launched on the Binance Smart Chain (BSC). On WSwap, users can perform ERC20 token swapping easily with the liquidity pool of the platform. Users can also provide liquidity to the pools and gain a part of the swapping fee and the platform's reward tokens.

WSwap Information:

Project Name	WSwap
Website	https://swap.wault.finance/
Smart Contract Type	Ethereum Smart Contract
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	May 12, 2021 - May 15, 2021
Reassessment Date	May 17, 2021

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit and Reassessment:

Name	Location (URL)
WEX.sol	https://github.com/WaultFinance/WAULT/blob/9f4ab8afc581d74ab881522c14c2a4d23cd0f6eb/contracts/WEX.sol
WexMaster.sol	https://github.com/WaultFinance/WAULT/blob/9f4ab8afc581d74ab881522c14c2a4d23cd0f6eb/contracts/WexMaster.sol
WswapV2Factory.sol	https://www.bscscan.com/address/0xb42e3fe71b7e0673335b3331b3e1053bd9822570#code
WswapV2Router02.sol	https://www.bscscan.com/address/0xd48745e39bbed146eec15b79cbf964884f9877c2#code

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue

3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Upgradable Without Timelock
Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation

Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

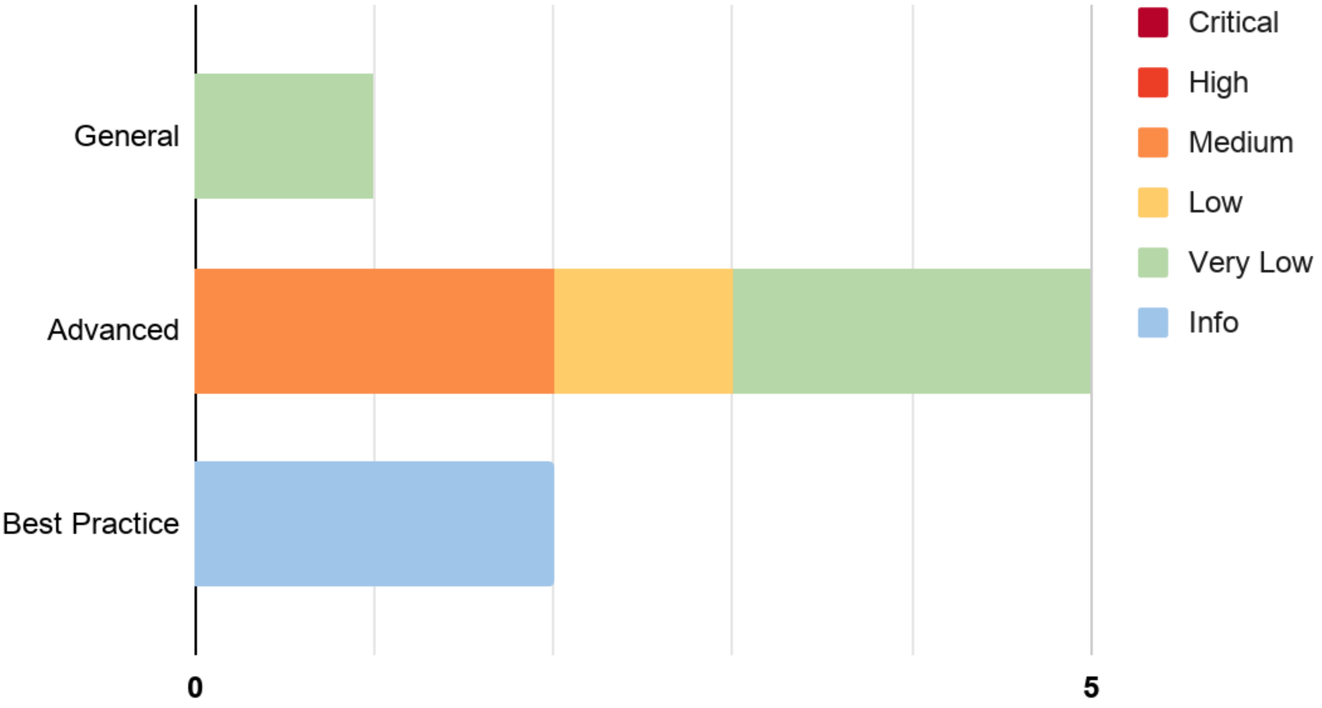
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

4. Summary of Findings

From the assessments, Inspex has found 8 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complication.
Resolved *	The issue has been resolved with mitigations and clarifications.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
XID-001	WEX Reward Miscalculation (wexPerBlock)	Advanced	**Medium**	Resolved *
XID-002	WEX Reward Miscalculation (totalAllocPoint)	Advanced	**Medium**	Resolved *
XID-003	Design Flaw in massUpdatePools() Function	Advanced	**Low**	Acknowledged
XID-004	Improper Deduction of User's Pending Reward	Advanced	**Very Low**	Acknowledged
XID-005	Improper Handling of Transfer Amount	Advanced	**Very Low**	Resolved *
XID-006	Outdated Compiler Version	General	**Very Low**	Acknowledged
XID-007	Implicit Visibility Level	Best Practice	**Info**	No Security Impact
XID-008	Unnecessary Function Calling	Best Practice	**Info**	No Security Impact

* The mitigations or clarifications by Wault Finance can be found in section 5.

5. Detailed Findings Information

5.1. WEX Reward Miscalculation (wexPerBlock)

ID	IDX-001
Target	WaxMaster.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The WEX reward miscalculation can lead to unfair WEX token distribution. **Likelihood:** Medium This function can be called by the owner only but would affect most pools that are not updated in the same block.
Status	Resolved * Wault Finance has mitigated this issue by committing to executing the `massUpdatePools()` function before every call of `setWexPerBlock()` from now on.

5.1.1. Description

The `wexPerBlock` variable is used to determine the total number of WEX tokens to be minted as a reward per block, so it is one of the main factors used in the rewards calculation. Therefore, whenever the `wexPerBlock` variable is modified without updating the pending rewards first, the reward of each pool will be incorrectly calculated.

In the `setWexPerBlock()` function shown below, the `wexPerBlock` variable is modified without updating the rewards.

WexMaster.sol

```
function setWexPerBlock(uint256 _wexPerBlock) public onlyOwner {
    require(_wexPerBlock > 0, "!wexPerBlock-0");
    wexPerBlock = _wexPerBlock;
}
```

For example:

Assuming that `wexPerBlock` is originally set to 1500 WEX per block.

Block	Action
1000	All pools’ rewards are updated.
1100	`wexPerBlock` is updated to 2000 WEX per block using `setWexPerBlock()` function.
1200	The pools’ rewards are updated once again.

The total rewards minted during block 1000 to block 1200 is equal to 2000 WEX per block, from block 1000 to block 1200 (2000 x (1200 - 1000) = 400000 WEX).

However, the rewards should be calculated by accounting for the original `wexPerBlock` value during the period when it is not yet updated as follows:

- 1500 WEX per block, from block 1000 to block 1100 (1500 * (1100 - 1000) = 150000 WEX)
- 2000 WEX per block, from block 1100 to block 1200 (2000 * (1200 - 1100) = 200000 WEX)
- Total WEX minted (150000 + 200000 = 350000 WEX)

5.1.2. Recommendation

Inspex suggests adding `massUpdatePools()` function calling before updating `wexPerBlock` variable as shown in the following example:

WexMaster.sol

```
function setWexPerBlock(uint256 _wexPerBlock) public onlyOwner {
    require(_wexPerBlock > 0, "!wexPerBlock-0");
    massUpdatePools();
    wexPerBlock = _wexPerBlock;
}
```

5.2. WEX Reward Miscalculation (totalAllocPoint)

ID	IDX-002
Target	WaxMaster.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The WEX reward miscalculation can lead to unfair WEX token distribution. **Likelihood:** Medium This issue would happen every time the `_withUpdate` parameter is set to false.
Status	Resolved * Wault Finance has mitigated this issue by committing to setting `_withUpdate` to `true` in every call of the `add()` and `set()` functions from now on.

5.2.1. Description

The `totalAllocPoint` variable is used to determine the portion that each pool would get from the total rewards minted, so it is one of the main factors used in the rewards calculation. Therefore, whenever the `totalAllocPoint` variable is modified without updating the pending rewards first, the reward of each pool will be incorrectly calculated.

In the `add()` and `set()` functions shown below, the `totalAllocPoint` variable is modified without updating the rewards.

WexMaster.sol

```
function add(
    uint256 _allocPoint,
    IERC20 _lpToken,
    bool _withUpdate
) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock
        ? block.number
        : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
        PoolInfo({
```

```
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accWexPerShare: 0
        })
    );
}

function set(
    uint256 _pid,
    uint256 _allocPoint,
    bool _withUpdate
) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
        _allocPoint
    );
    poolInfo[_pid].allocPoint = _allocPoint;
}
```

For example:

Assuming that `wexPerBlock` is originally set to 2000 WEX per block.

Block	Action
1000	All pools' rewards are updated.
1100	A new pool is added using the **add()** function, causing the **totalAllocPoint** to be changed to 120.
1200	The pools' rewards are updated once again.

The total rewards allocated to pool 0 during block 1000 to block 1200 is equal to 2000 WEX per block (2000 * (1200 - 1000) * (10 / 120) = 33333.33 WEX).

However, the rewards should be calculated by accounting for the original `totalAllocPoint` value during the period when it is not yet updated as follows:

- 2000 WEX per block, from block 1000 to block 1100, with a proportion of 10/100 (2000 * (1100 - 1000) * (10 / 100) = 20000 WEX)
- 2000 WEX per block, from block 1100 to block 1200, with a proportion of 10/120 (2000 * (1100 - 1000) * (10 / 120) = 16666.67 WEX)
- Total WEX allocated to pool 0 (20000 + 16666.67 = 36666.67 WEX)

5.2.2. Recommendation

Inspex suggests removing `_withUpdate` variable in the `set()` and `add()` functions and always calling the `massUpdatePools()` function before updating `totalAllocPoint` variable as shown in the following example:

WexMaster.sol

```
function add(
    uint256 _allocPoint,
    IERC20 _lpToken
) public onlyOwner {
    massUpdatePools();
    uint256 lastRewardBlock = block.number > startBlock
        ? block.number
        : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
        PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accWexPerShare: 0
        })
    );
}

function set(
    uint256 _pid,
    uint256 _allocPoint
) public onlyOwner {
    massUpdatePools();
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
        _allocPoint
    );
    poolInfo[_pid].allocPoint = _allocPoint;
}
```

5.3. Design Flaw in massUpdatePools() Function

ID	IDX-003
Target	WaxMaster.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `massUpdatePools()` will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the `poolInfo` size will be raised until the `massUpdatePools()` is eventually unusable.
Status	Acknowledged At the time of assessment, the number of pools is small, so the gas fee is very unlikely to reach the gas limit. This function is mainly used by the contract owner and Wault Finance is willing to pay for the increasing gas fee.

5.3.1. Description

The `massUpdatePools()` function executes the `updatePool()` function, which is a state modifying function for all added pools as shown below:

WexMaster.sol

```
function massUpdatePools() public {
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
        updatePool(pid);
    }
}
```

With the current design, the added pools cannot be removed. They can only be disabled by setting the `pool.allocPoint` to 0. Even if a pool is disabled, the `updatePool()` function for this pool is still called. Therefore, if new pools continue to be added to this contract, the `poolInfo.length` will continue to grow and this function will eventually be unusable due to excessive gas usage.

5.3.2. Recommendation

Inspex suggests making the contract capable of removing unnecessary/ended pools to reduce the loop round in the `massUpdatePools()` function as follows:

```
require(_pid < poolInfo.length);
poolInfo[_pid] = poolInfo[poolInfo.length-1];
poolInfo.length--;
```

5.4. Improper Deduction of User's Pending Reward

ID	IDX-004
Target	WexMaster.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low The user will lose only the pending WEX token rewards. **Likelihood:** Low It is very unlikely that a user would need to call the `emergencyWithdraw()` function.
Status	Acknowledged Currently, there is no need to use the `emergencyWithdraw()` function. However, in the case that it is required, Wault Finance will suggest the users to execute the `claim()` function first, and Wault Finance has confirmed that they will return all the pending rewards to the investors. Wault Finance has acknowledged this issue and will fix it in the next release.

5.4.1. Description

The `emergencyWithdraw()` function can be used to withdraw all the LP tokens from a specific pool that the user has deposited into. This function is designed to be used in an emergency case only, so no reward would be withdrawn from the pool. However, the amount of pending rewards that should be claimable by the user is also reset to 0, causing a loss of accumulated pending rewards to the user when this function is used.

WexMaster.sol

```
function emergencyWithdraw(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    pool.lpToken.safeTransfer(address(msg.sender), user.amount);
    emit EmergencyWithdraw(msg.sender, _pid, user.amount);
    user.amount = 0;
    user.rewardDebt = 0;
    user.pendingRewards = 0;
}
```

5.4.2. Recommendation

Inspex recommends not to set the user's accumulated pending reward to 0 when the `emergencyWithdraw()` function is called. The example can be seen in the following source code:

WexMaster.sol

```
function emergencyWithdraw(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    pool.lpToken.safeTransfer(address(msg.sender), user.amount);
    emit EmergencyWithdraw(msg.sender, _pid, user.amount);
    user.amount = 0;
    user.rewardDebt = 0;
}
```

5.5. Improper Handling of Transfer Amount

ID	IDX-005
Target	WexMaster.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low The user will receive fewer WEX tokens than the actual amount claimable. **Likelihood:** Low During this audit activity, Inspex has yet to find a scenario that can cause the amount of WEX token to be insufficient.
Status	Resolved * Wault Finance has responded that there is no scenario in the present that the amount of WEX would be insufficient, so there is no risk. However, Inspex suggests fixing this issue in preparation for new functions or contracts that may be integrated with this smart contract in the future.

5.5.1. Description

Please be noted that only the `claim()` function will be used as an example of this issue.

In the `deposit()`, `withdraw()`, and `claim()` functions, to transfer WEX token to a user, the `safeWexTransfer()` function is called.

WexMaster.sol

```
function claim(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    uint256 pending =
user.amount.mul(pool.accWexPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0 || user.pendingRewards > 0) {
        user.pendingRewards = user.pendingRewards.add(pending);
        safeWexTransfer(msg.sender, user.pendingRewards);
        emit Claim(msg.sender, _pid, user.pendingRewards);
        user.pendingRewards = 0;
    }
    user.rewardDebt = user.amount.mul(pool.accWexPerShare).div(1e12);
}
```

In the `safeWexTransfer()` function, if the claim amount exceeds the current WEX balance of the `WexMaster` contract, all WEX tokens in the contract will be transferred to the user.

WexMaster.sol

```
function safeWexTransfer(address _to, uint256 _amount) internal {
    uint256 wexBal = wex.balanceOf(address(this));
    if (_amount > wexBal) {
        wex.transfer(_to, wexBal);
    } else {
        wex.transfer(_to, _amount);
    }
}
```

Then, the `user.pendingRewards` will be set to 0 as shown below:

WexMaster.sol

```
function claim(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    uint256 pending =
user.amount.mul(pool.accWexPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0 || user.pendingRewards > 0) {
        user.pendingRewards = user.pendingRewards.add(pending);
        safeWexTransfer(msg.sender, user.pendingRewards);
        emit Claim(msg.sender, _pid, user.pendingRewards);
        user.pendingRewards = 0;
    }
    user.rewardDebt = user.amount.mul(pool.accWexPerShare).div(1e12);
}
```

As a result, the user will receive less WEX token than the actual amount claimable.

5.5.2. Recommendation

Inspex suggests reducing the user pending reward equal to the actual amount of WEX token transferred as shown below:

WexMaster.sol

```
function claim(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    uint256 pending =
user.amount.mul(pool.accWexPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0 || user.pendingRewards > 0) {
        user.pendingRewards = user.pendingRewards.add(pending);
        uint256 claimedReward = safeWexTransfer(msg.sender,
user.pendingRewards);
        emit Claim(msg.sender, _pid, claimedReward);
        user.pendingRewards -= claimedReward;
    }
    user.rewardDebt = user.amount.mul(pool.accWexPerShare).div(1e12);
}
```

WexMaster.sol

```
function safeWexTransfer(address _to, uint256 _amount) internal returns
(uint32) {
    uint256 wexBal = wex.balanceOf(address(this));
    if (_amount > wexBal) {
        wex.transfer(_to, wexBal);
        return wexBal;
    } else {
        wex.transfer(_to, _amount);
        return _amount;
    }
}
```

5.6. Outdated Compiler Version

ID	IDX-006
Target	WswapV2Router02.sol, WswapV2Factory.sol
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Very Low **Impact:** Low From the list of known Solidity bugs, direct impact cannot be caused from those bugs themselves. **Likelihood:** Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	Acknowledged Wault Finance has acknowledged this issue and will fix it in the next release.

5.6.1. Description

The Solidity compiler versions specified in the smart contracts were outdated. These versions have publicly known inherent bugs[2][3] that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

WswapV2Router02.sol

```
// SPDX-License-Identifier: MIT

pragma solidity =0.6.6;
```

WswapV2Factory.sol

```
pragma solidity =0.5.16;
```

5.6.2. Recommendation

Inspex suggests upgrading the Solidity compiler to the latest stable version[4].

As of May 2021, the latest stable versions of Solidity compiler in each major are as follows:

- Major 0.5: v0.5.17
- Major 0.6: v0.6.12

5.7. Improper Function Visibility

ID	IDX-007
Target	WEX.sol, WexMaster.sol
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	**No Security Impact** Wault Finance has acknowledged this issue and will fix it in the next release.

5.7.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

The following source code shows that the `setBurnrate()` function of the WEX token is set to `public` and it is never called from any internal function.

WEX.sol

```
909 function setBurnrate(uint8 burnrate_) public onlyOwner {
910     require(0 <= burnrate_ && burnrate_ <= 20, "burnrate must be in valid range");
911     _setupBurnrate(burnrate_);
912 }
```

The following table contains all functions that have `public` visibility and are never called from any internal function.

Target	Function
WEX.sol (L: 905)	`mint()`
WEX.sol (L: 909)	`setBurnrate()`
WEX.sol (L: 914)	`addWhitelistedAddress()`
WEX.sol (L: 919)	`removeWhitelistedAddress()`
WexMaster.sol (L: 1289)	`add()`

WexMaster.sol (L: 1311)	`set()`
WexMaster.sol (L: 1380)	`deposit()`
WexMaster.sol (L: 1409)	`withdraw()`
WexMaster.sol (L: 1436)	`emergencyWithdraw()`
WexMaster.sol (L: 1442)	`claim()`
WexMaster.sol (L: 1465)	`setWexPerBlock()`

5.7.2. Recommendation

Inspex suggests changing all functions' visibility to `external` if they are not called from any internal function as shown in the following example:

WEX.sol

```
function setBurnrate(uint8 burnrate_) external onlyOwner {
    require(0 <= burnrate_ && burnrate_ <= 20, "burnrate must be in valid range");
    _setupBurnrate(burnrate_);
}
```

5.8. Unnecessary Function Calling

ID	IDX-008
Target	WswapV2Router02.sol
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	No Security Impact Wault Finance has acknowledged this issue and will fix it in the next release.

5.8.1. Description

The `pairFor()` function returns the address of an LP pair corresponding to the factory and the addresses of the token pair. However, it is called inside the `getReserves()` function without any purpose as shown below, resulting in unnecessarily wasted gas.

WswapV2Router02.sol

```
function getReserves(address factory, address tokenA, address tokenB) internal view returns (uint reserveA, uint reserveB) {
    (address token0,) = sortTokens(tokenA, tokenB);
    pairFor(factory, tokenA, tokenB);
    (uint reserve0, uint reserve1,) = IWaultSwapPair(pairFor(factory, tokenA, tokenB)).getReserves();
    (reserveA, reserveB) = tokenA == token0 ? (reserve0, reserve1) : (reserve1, reserve0);
}
```

5.8.2. Recommendation

Inspex suggests removing the unused `pairFor()` function calling in the `getReserves()` function as shown in the following example:

WswapV2Router02.sol

```
function getReserves(address factory, address tokenA, address tokenB) internal view returns (uint reserveA, uint reserveB) {
    (address token0,) = sortTokens(tokenA, tokenB);
    (uint reserve0, uint reserve1,) = IWaultSwapPair(pairFor(factory, tokenA, tokenB)).getReserves();
    (reserveA, reserveB) = tokenA == token0 ? (reserve0, reserve1) : (reserve1, reserve0);
}
```

6. Appendix

6.1. About Inspex

CYBERSECURITY
PROFESSIONAL
SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Contact Information:

Website	https://inspex.co
Twitter	https://twitter.com/InspexCo
Telegram	t.me/inspexco
Email	contact@inspex.co

6.2. References

[1] "OWASP Risk Rating Methodology." [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

[2] "List of Known Bugs — Solidity 0.5.16 documentation." [Online]. Available: https://docs.soliditylang.org/en/v0.5.16/bugs.html. [Accessed: 15-May-2021]

[3] "List of Known Bugs — Solidity 0.6.6 documentation." [Online]. Available: https://docs.soliditylang.org/en/v0.6.6/bugs.html. [Accessed: 15-May-2021]

[4] ethereum, "Releases · ethereum/solidity." [Online]. Available: https://github.com/ethereum/solidity/releases. [Accessed: 15-May-2021]

inspex

CYBERSECURITY PROFESSIONAL SERVICE