

# WexMaster Phase 2

## Smart Contract Audit Report Prepared for Wault Finance



---

<b>Date Issued:</b>	Jun 22, 2021
<b>Project ID:</b>	AUDIT2021004
<b>Version:</b>	v2.0
<b>Confidentiality Level:</b>	Public



## Report Information

Project ID	AUDIT2021004
Version	v2.0
Client	Wault Finance
Project	WexMaster Phase 2
Auditor(s)	Weerawat Pawanawiwat Pongsakorn Sommalai Suvicha Buakhom
Author	Pongsakorn Sommalai
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

## Version History

Version	Date	Description	Author(s)
2.0	Jun 22, 2021	Add Improper Reward Calculation issue	Weerawat Pawanawiwat
1.0	Jun 13, 2021	Full report	Pongsakorn Sommalai

## Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	<a href="https://t.me/inspexco">t.me/inspexco</a>
Email	<a href="mailto:audit@inspex.co">audit@inspex.co</a>

# Table of Contents

<b>1. Executive Summary</b>	<b>1</b>
1.1. Audit Result	1
1.2. Disclaimer	1
<b>2. Project Overview</b>	<b>2</b>
2.1. Project Introduction	2
2.2. Scope	2
<b>3. Methodology</b>	<b>3</b>
3.1. Test Categories	3
3.2. Audit Items	4
3.3. Risk Rating	5
<b>4. Summary of Findings</b>	<b>6</b>
<b>5. Detailed Findings Information</b>	<b>7</b>
5.1. Potential Centralized Control of State Variable	7
5.2. Improper Reward Calculation	9
<b>6. Appendix</b>	<b>12</b>
6.1. About Inspex	12
6.2. References	13

---

# 1. Executive Summary

As requested by Wault Finance, Inspex team conducted an audit to verify the security posture of the WexMaster Phase 2 smart contracts on Jun 12, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of WexMaster Phase 2 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 1 medium and 1 low-severity issues, and the issues were acknowledged by the Wault Finance team. Inspex suggests resolving all issues found in this report as soon as possible to reduce the risk and improve the security level of the smart contracts.

## 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), Inspex 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

WexMaster is a smart contract made to distribute Vault Finance platform's governance token. The users can stake predefined tokens into the pools to gain the governance token as a reward. For this version, Vault Finance has resolved the issues found from the previous version which has been deployed on BSC in preparation for the new deployment on Polygon and also the remediation on BSC.

#### Scope Information:

Project Name	WexMaster Phase 2
Website	<a href="https://wault.finance/">https://wault.finance/</a>
Smart Contract Type	Ethereum Smart Contract
Programming Language	Solidity

#### Audit Information:

Audit Method	Whitebox
Audit Date	Jun 12, 2021

### 2.2. Scope

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

#### Initial Audit:

Name	Location (URL)
WexMaster.sol	<a href="https://github.com/WaultFinance/WAULT/blob/b995392a67c7ffb5cbcd36297a11f9ac7c98ed41/contracts/WexMaster.sol">https://github.com/WaultFinance/WAULT/blob/b995392a67c7ffb5cbcd36297a11f9ac7c98ed41/contracts/WexMaster.sol</a>

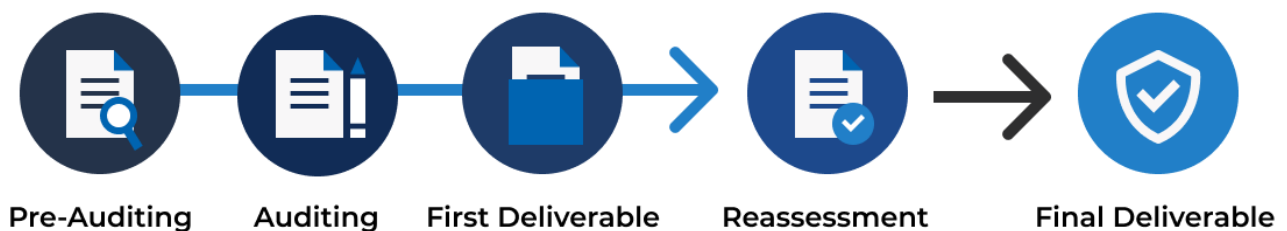
#### Reassessment:

There is no issue that needed the reassessment activity.

## 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
<b>Best Practice</b>
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

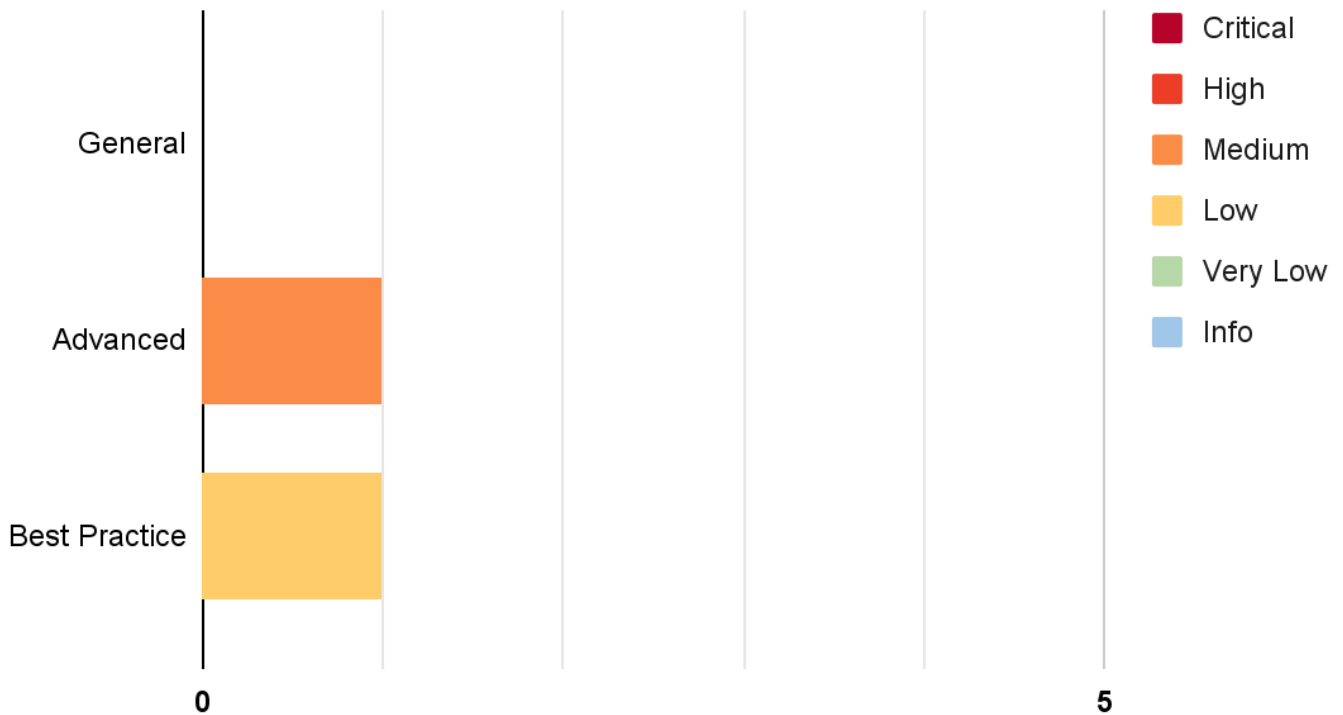
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

**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 2 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. For the clarification or mitigation detail, please refer to Chapter 5.
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
IDX-001	Improper Update of State Variable	General	Low	Acknowledged
IDX-002	Improper Reward Calculation	Advanced	Medium	Acknowledged

## 5. Detailed Findings Information

### 5.1. Potential Centralized Control of State Variable

ID	IDX-001
Target	WexMaster.sol
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	<p><b>Severity:</b> Low</p> <p><b>Impact:</b> Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users.</p> <p><b>Likelihood:</b> Low There is nothing to restrict the changes from being done; however, the changes are limited by fixed values in the smart contracts.</p>
Status	<p><b>Acknowledged</b></p> <p>The issue's risk has been acknowledged and accepted.</p>

#### 5.1.1. Description

Critical state variables can be updated any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

Targets	Function	Modifier
WexMaster.sol (L:1297)	add()	onlyOwner
WexMaster.sol (L:1318)	remove()	onlyOwner
WexMaster.sol (L:1328)	set()	onlyOwner
WexMaster.sol (L:1481)	setWexPerBlock()	onlyOwner

### 5.1.2. Recommendation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a Timelock contract to delay the changes for a reasonable amount of time

## 5.2. Improper Reward Calculation

ID	IDX-002
Target	WexMaster.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<b>Severity: Medium</b> <b>Impact: Medium</b> The reward of the pool that has the same staking token as the reward token will be slightly lower than what it should be. <b>Likelihood: Medium</b> It is likely that there is a pool that has the same staking token as the reward token.
Status	<b>Acknowledged</b> Vault Finance team has acknowledged and planned to fix this issue by using the wrapped tokens instead of \$WEX and \$WEXPoly directly.

### 5.2.1. Description

In the `WexMaster` contract, a new staking pool can be added using the `add()` function. The staking token for the new pool is defined using the `_lpToken` variable; however, there is no additional checking whether the `_lpToken` is the same as the reward token (`wex`) or not.

#### WexMaster.sol

```
1297 function add(  
1298     uint256 _allocPoint,  
1299     IERC20 _lpToken  
1300 ) external onlyOwner {  
1301     require(!addedPools[address(_lpToken)], '_lpToken already added');  
1302     massUpdatePools();  
1303     uint256 lastRewardBlock = block.number > startBlock  
1304         ? block.number  
1305         : startBlock;  
1306     totalAllocPoint = totalAllocPoint.add(_allocPoint);  
1307     poolInfo.push(  
1308         PoolInfo({  
1309             lpToken: _lpToken,  
1310             allocPoint: _allocPoint,  
1311             lastRewardBlock: lastRewardBlock,  
1312             accWexPerShare: 0  
1313         })  
1314     );
```

```
1315     addedPools[address(_lpToken)] = true;
1316 }
```

When the `_lpToken` is the same token as `wex`, reward calculation for that pool in the `updatePool()` function can be incorrect. This is because the current balance of the `_lpToken` in the contract is used in the calculation of the reward. Since the `_lpToken` is the same token as the reward, the reward minted to the contract will inflate the value of `lpSupply`, causing the reward of that pool to be less than what it should be.

#### WexMaster.sol

```
1376 function updatePool(uint256 _pid) public validatePoolByPid(_pid) {
1377     PoolInfo storage pool = poolInfo[_pid];
1378     if (block.number <= pool.lastRewardBlock) {
1379         return;
1380     }
1381     uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1382     if (lpSupply == 0) {
1383         pool.lastRewardBlock = block.number;
1384         return;
1385     }
1386     uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1387     uint256 wexReward = multiplier
1388         .mul(wexPerBlock)
1389         .mul(pool.allocPoint)
1390         .div(totalAllocPoint);
1391     wex.mint(address(this), wexReward);
1392     pool.accWexPerShare = pool.accWexPerShare.add(
1393         wexReward.mul(1e12).div(lpSupply)
1394     );
1395     pool.lastRewardBlock = block.number;
1396 }
```

### 5.2.2. Recommendation

Inspex suggests checking the value of the `_lpToken` in the `add()` function to prevent the pool with the same staking token as the reward token from being added , for example:

#### WexMaster.sol

```
1297 function add(  
1298     uint256 _allocPoint,  
1299     IERC20 _lpToken  
1300 ) external onlyOwner {  
1301     require(!addedPools[address(_lpToken)], '_lpToken already added');  
1302     require(_lpToken != wex, '_lpToken is wex');  
1303     massUpdatePools();  
1304     uint256 lastRewardBlock = block.number > startBlock  
1305         ? block.number  
1306         : startBlock;  
1307     totalAllocPoint = totalAllocPoint.add(_allocPoint);  
1308     poolInfo.push(  
1309         PoolInfo({  
1310             lpToken: _lpToken,  
1311             allocPoint: _allocPoint,  
1312             lastRewardBlock: lastRewardBlock,  
1313             accWexPerShare: 0  
1314         })  
1315     );  
1316     addedPools[address(_lpToken)] = true;  
1317 }
```

However, if the pool with the same staking token as the reward token is required, Inspex suggests minting the reward token to another contract to prevent the amount of the staked token from being mixed up with the reward token.

In this case, the WexMaster contract has already been deployed. Therefore, Inspex suggests performing the following actions:

- Deploy the wrapped tokens of \$WEX and \$WEXPoly
- Set the `allocPoints` of the affected pool to 0
- Create new pool with the deployed wrapped tokens
- Migrate all staked tokens from the affected pool to the new pool

## 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.

#### Follow Us On:

Website	<a href="https://inspex.co">https://inspex.co</a>
Twitter	<a href="https://twitter.com/InspexCo">@InspexCo</a>
Facebook	<a href="https://www.facebook.com/InspexCo">https://www.facebook.com/InspexCo</a>
Telegram	<a href="https://t.me/inspex_announcement">@inspex_announcement</a>

---

## 6.2. References

- [1] “OWASP Risk Rating Methodology.” [Online]. Available:  
[https://owasp.org/www-community/OWASP\\_Risk\\_Rating\\_Methodology](https://owasp.org/www-community/OWASP_Risk_Rating_Methodology). [Accessed: 08-May-2021]



**inspex**  
CYBERSECURITY PROFESSIONAL SERVICE