Farm

Smart Contract Audit Report
Prepared for Powaa

Date Issued: Sep 6, 2022
Project ID: AUDIT2022046
Version: v3.0
Confidentiality Level: Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022046
Version	v3.0
Client	Powaa
Project	Farm
Auditor(s)	Darunphop Pengkumta Fungkiat Phadejtaku
Author(s)	Darunphop Pengkumta
Reviewer	Patipon Suwanbol
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
3.0	Sep 6, 2022	Public version report	Darunphop Pengkumta
2.0	Sep 5, 2022	Update the issues' severity and status	Darunphop Pengkumta
1.0	Sep 5, 2022	Full report	Darunphop Pengkumta

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by Powaa, Inspex team conducted an audit to verify the security posture of the Farm smart contracts between Aug 30, 2022 and Aug 31, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of the Farm 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 high, 2 medium, 1 low, 1 very low, and 1 info-severity issues. With the project team's prompt response, 2 high, 1 medium, 1 very low and 1 info-severity issues were resolved in the reassessment, while 1 medium and 1 low-severity issues were acknowledged by the team. Therefore, in the long run, Inspex suggests resolving all issues found in this report.

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

Powaa farm is a farming protocol that helps traders make the most out of the upcoming Ethereum Merge towards PoS. The project offers the users the vaults to deposit their assets to gain rewards from the protocol. Moreover, once the Merge is complete, the protocol will swap all the deposited assets on PoW Ethereum for ETHW and re-distribute them back to the depositors.

Scope Information:

Project Name	Farm
Website	https://powaa.xyz/
Smart Contract Type	Ethereum Smart Contract
Chain	Ethereum
Programming Language	Solidity
Category	Token, Yield Farming

Audit Information:

Audit Method	Whitebox
Audit Date	Aug 30, 2022 - Aug 31, 2022
Reassessment Date	Sep 5, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

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

Initial Audit: (Commit: 977b2c3267fff6d9760c486f6754d25629cb345f)

Contract	Location (URL)
TokenVault	https://github.com/powaa-protocol/powaa-contract/blob/977b2c3267/contracts/v0.8.16/TokenVault.sol
GovLPVault	https://github.com/powaa-protocol/powaa-contract/blob/977b2c3267/contracts/v0.8.16/GovLPVault.sol
BaseTokenVault	https://github.com/powaa-protocol/powaa-contract/blob/977b2c3267/contracts/v0.8.16/BaseTokenVault.sol
UniswapV2GovLPVaultMigrator	https://github.com/powaa-protocol/powaa-contract/blob/977b2c3267/contracts/v0.8.16/migrators/gov-lp-vaults/UniswapV2GovLPVaultMigrator.sol
UniswapV3TokenVaultMigrator	https://github.com/powaa-protocol/powaa-contract/blob/977b2c3267/contracts/v0.8.16/migrators/token-vaults/UniswapV3TokenVaultMigrator.sol
SushiSwapLPVaultMigrator	https://github.com/powaa-protocol/powaa-contract/blob/977b2c3267/contracts/v0.8.16/migrators/token-vaults/SushiSwapLPVaultMigrator.sol
Controller	https://github.com/powaa-protocol/powaa-contract/blob/977b2c3267/contracts/v0.8.16/Controller.sol
LinearFeeModel	https://github.com/powaa-protocol/powaa-contract/blob/977b2c3267/contracts/v0.8.16/fee-model/LinearFeeModel.sol
POWAA	https://github.com/powaa-protocol/powaa-contract/blob/977b2c3267/contracts/v0.8.16/POWAA.sol

Reassessment: (Commit: a86504357bfc53629c4d20024463a712fabc9f84)

Contract	Location (URL)
TokenVault	https://github.com/powaa-protocol/powaa-contract/blob/a86504357b/contracts/v0.8.16/TokenVault.sol
GovLPVault	https://github.com/powaa-protocol/powaa-contract/blob/a86504357b/contracts/v0.8.16/GovLPVault.sol
BaseTokenVault	https://github.com/powaa-protocol/powaa-contract/blob/a86504357b/contracts/v0.8.16/BaseTokenVault.sol
UniswapV2GovLPVaultMigrator	https://github.com/powaa-protocol/powaa-contract/blob/a86504357b/contracts/v0.8.16/migrators/gov-lp-vaults/UniswapV2GovLPVaultMigrator.sol

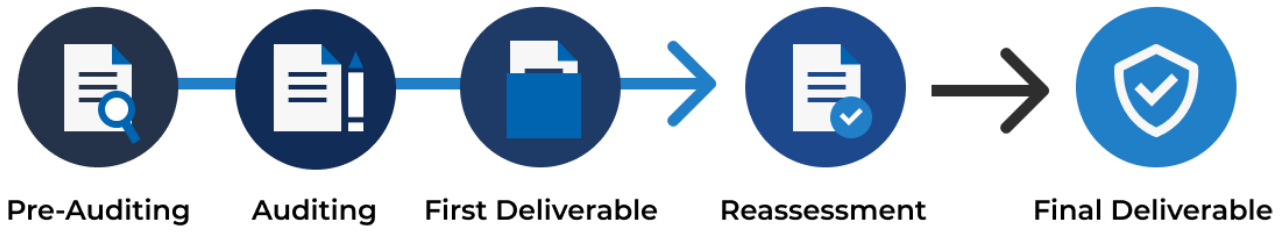
UniswapV3TokenVaultMigrator	https://github.com/powaa-protocol/powaa-contract/blob/a86504357b/contracts/v0.8.16/migrators/token-vaults/UniswapV3TokenVaultMigrator.sol
SushiSwapLPVaultMigrator	https://github.com/powaa-protocol/powaa-contract/blob/a86504357b/contracts/v0.8.16/migrators/token-vaults/SushiSwapLPVaultMigrator.sol
Controller	https://github.com/powaa-protocol/powaa-contract/blob/a86504357b/contracts/v0.8.16/Controller.sol
LinearFeeModel	https://github.com/powaa-protocol/powaa-contract/blob/a86504357b/contracts/v0.8.16/fee-model/LinearFeeModel.sol
POWAA	https://github.com/powaa-protocol/powaa-contract/blob/a86504357b/contracts/v0.8.16/POWAA.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

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 testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) 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**.

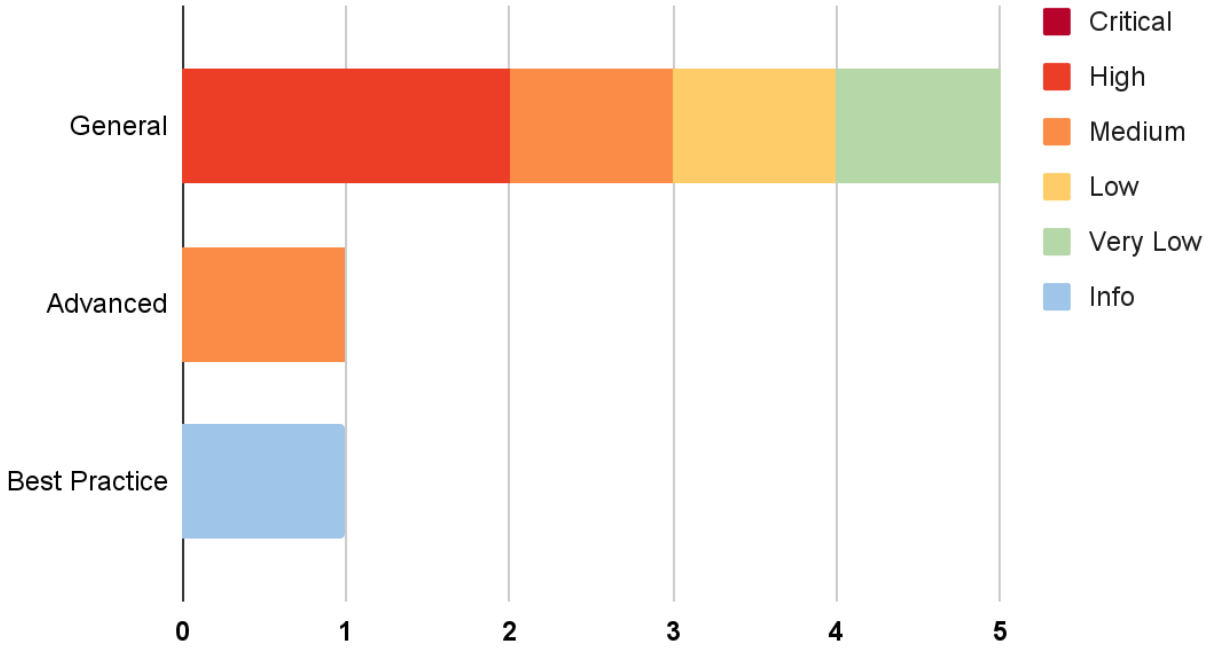
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

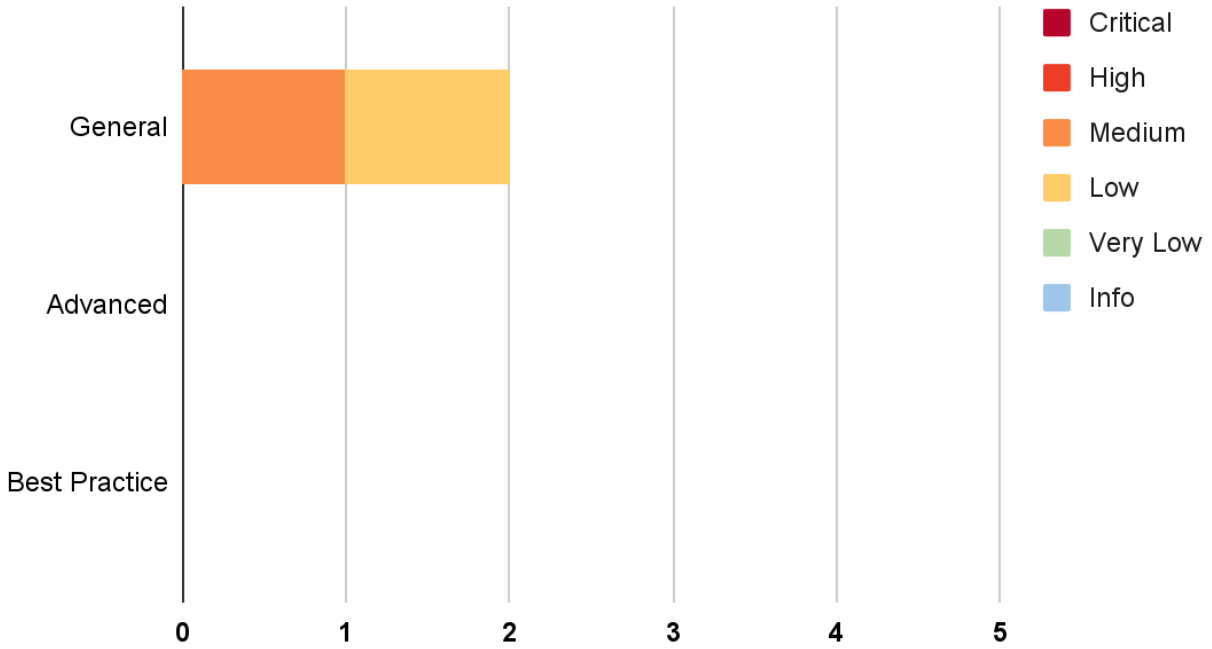
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
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	Token Manual Minting by Contract Owner	General	High	Resolved *
IDX-002	Centralized Control of State Variable	General	High	Resolved *
IDX-003	Transaction Ordering Dependence	General	Medium	Acknowledged
IDX-004	Improper Reward Update in Claim Functions	Advanced	Medium	Resolved
IDX-005	Loop Over Unbounded Data Structure	General	Low	Acknowledged
IDX-006	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-007	Improper Function Visibility	Best Practice	Info	Resolved

* The mitigations or clarifications by Powaa can be found in Chapter 5.

5. Detailed Findings Information

5.1. Token Manual Minting by Contract Owner

ID	IDX-001
Target	POWAA
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** High **Impact:** High The contract owner can arbitrarily mint the tokens. They can mint any amount of the token to anyone until the total minted value reaches the `_maxTotalSupply` state. **Likelihood:** Medium Only the owner can call the `mint()` function. The owner can mint the token to the owner's wallet and dump the token to the available market.
Status	Resolved * The Powaa team has confirmed that the `mint()` function will be called through a timelock contract. This allows the platform users to be notified that the owner will mint the `$POWAA` tokens.

5.1.1. Description

In the `POWAA` contract, the `mint()` function has `onlyOwner` as a modifier, so only the owner of the contract can execute this function. This function can be used to arbitrary mint `$POWAA` as shown below.

POWAA.sol

```
24 function mint(address to, uint256 amount) external onlyOwner returns (bool) {
25   if (ERC20.totalSupply() + amount > _maxTotalSupply) {
26     revert POWAA_MaxTotalSupplyExceeded();
27   }
28   _mint(to, amount);
29   return true;
30 }
```

As a result, the owner can call the `mint()` function in order to mint the tokens.

5.1.2. Remediation

Inspex suggests removing the `mint()` function from the token contract to prevent manually minting tokens by the owner and implementing a contract to control the `$POWAA` minting amount and distribute them to the vaults systematically. However, if the `mint()` function is needed, Inspex suggests implementing a

community-run smart contract governance to control the use of these functions.

If removing the functions or implementing the smart contract governance does not suit the business design, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the execution of the `mint` function for a reasonable amount of time e.g., 24 hours.

5.2. Centralized Control of State Variable

ID	IDX-002
Target	BaseTokenVault TokenVault GovLPVault
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood: Medium** There is nothing to restrict the changes from being done; however, this action can only be done by the privileged roles.
Status	Resolved * The Powaa team has confirmed that the privileged functions will be called through a timelock contract. The critical state-changing function will be handled with a time delay before taking any effect. So, the users can strategize their actions based on the upcoming effects of the timelock contract. **The user should verify that the privilege execution rights have been transferred to the timelock contract before using the platform.**

5.2.1. Description

Critical state variables can be updated at 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:

File	Contract	Function	Modifier/Role
BaseTokenVault.sol (L:183)	BaseTokenVault	setRewardsDistribution()	onlyMasterContractOwner
BaseTokenVault.sol (L:243)	BaseTokenVault	notifyRewardAmount()	onlyRewardsDistribution
TokenVault.sol (L:74)	TokenVault	setMigrationOption()	onlyMasterContractOwner
TokenVault.sol (L:197)	TokenVault	notifyRewardAmount()	onlyRewardsDistribution

GovLPVault.sol (L:57)	GovLPVault	setMigrationOption()	onlyMasterContractOwner
GovLPVault.sol (L:68)	GovLPVault	migrate()	masterContract.owner()

5.2.2. Remediation

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 implementing a community-run smart contract governance to control the use of these functions.

If removing the functions or implementing the smart contract governance is not possible, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the changes for a reasonable amount of time e.g., 24 hours. For example, the functions that have the `onlyMasterContractOwner` modifier require the `msg.sender` state to be the owner of the master contract. By changing the owner of the master contract into a timelock contract, this can mitigate the impact that is caused by a sudden change of the critical states. This is an example of a timelock contract implementation: https://blog.openzeppelin.com/protect-your-users-with-smart-contract-timelocks/.

5.3. Transaction Ordering Dependence

ID	IDX-003
Target	UniswapV2GovLPVaultMigrator UniswapV3TokenVaultMigrator SushiSwapLPVaultMigrator
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Medium **Impact:** High The front-running attack can be performed, resulting in a bad swapping rate and a lower bounty. **Likelihood:** Low It is easy to perform the attack. When trading is enabled, any user can swap the token in the pool, resulting in the attacker can front-run the migrator's transaction. The migrator can only swap once per vault, so, would not happen again in the same vault. Furthermore, the migration is designed to be executed in the Proof-of-Work chain after The Merge, so the Proof-of-Work `$ETH` could be considered as another asset from the Proof-of-Stake `$ETH`. The value on the Proof-of-Work chain should be naturally lower than the main chain, which disincentivizes the attacker from leveraging this issue. Likewise, with the staking token Proof-of-Work chain, they are a forked asset that does not inherently have a value, which does not have an effect on the asset on the main chain.
Status	**Acknowledged** The Powaa team has acknowledged and accepted the impact of the issue.

5.3.1. Description

In the `UniswapV3TokenVaultMigrator` contract, the `exactInputSingle()` function is called with the `IV3SwapRouter.ExactInputSingleParams` struct, which the `amountOutMinimum` parameter is set to 0 at line 122, into the `router` state at line 126 as shown below:

UniswapV3TokenVaultMigrator.sol

```
104 function execute(bytes calldata _data)
105   external
106   onlyWhitelistedTokenVault(msg.sender)
107   nonReentrant
108 {
109   (address token, uint24 poolFee) = abi.decode(_data, (address, uint24));
110
```

```
111   uint256 swapAmount = IERC20(token).balanceOf(address(this));
112
113   IERC20(token).safeApprove(address(router), swapAmount);
114
115   IV3SwapRouter.ExactInputSingleParams memory params = IV3SwapRouter
116     .ExactInputSingleParams({
117       tokenIn: token,
118       tokenOut: WETH9,
119       fee: poolFee,
120       recipient: address(this),
121       amountIn: swapAmount,
122       amountOutMinimum: 0,
123       sqrtPriceLimitX96: 0
124     });
125
126   router.exactInputSingle(params);
127   _unwrapWETH(address(this));
128
129   uint256 govLPTokenVaultFee = govLPTokenVaultFeeRate.mulWadDown(
130     address(this).balance
131   );
132   uint256 treasuryFee = treasuryFeeRate.mulWadDown(address(this).balance);
133   uint256 controllerFee = controllerFeeRate.mulWadDown(address(this).balance);
134   uint256 vaultReward = address(this).balance -
135     govLPTokenVaultFee -
136     treasuryFee -
137     controllerFee;
138   treasury.safeTransferETH(treasuryFee);
139   govLPTokenVault.safeTransferETH(govLPTokenVaultFee);
140   controller.safeTransferETH(controllerFee);
141   msg.sender.safeTransferETH(vaultReward);
142
143   emit Execute(vaultReward, treasuryFee, controllerFee, govLPTokenVaultFee);
144 }
```

Therefore, the front-running attack can be performed, resulting in a bad swapping rate.

The line that the values have been set to 0 in the `amountOutMin` parameter or the `amountOutMinimum` parameters are as follows:

File	Contract	Function
UniswapV2GovLPVaultMigrator.sol (L:73)	UniswapV2GovLPVaultMigrator	execute()
UniswapV3TokenVaultMigrator.sol (L:122)	UniswapV3TokenVaultMigrator	execute()
SushiSwapLPVaultMigrator.sol (L:113)	SushiSwapLPVaultMigrator	execute()

SushiSwapLPVaultMigrator.sol (L:128)	SushiSwapLPVaultMigrator	execute()

5.3.2. Remediation

The tolerance values (the `amountOutMin` and `amountOutMinimum`) should not be set to 0. Inspex suggests calculating the expected amount out with the token price fetched from price oracles.

In case the price of the needed tokens is not available from other trustable sources, Inspex suggests using the time-weight average price (TWAP Oracle) instead of directly quoting from the reserves (https://docs.uniswap.org/protocol/V2/concepts/core-concepts/oracles).

5.4. Improper Reward Update in Claim Functions

ID	IDX-004
Target	GovLPVault TokenVault
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The user will lose the reward that has not been updated when claiming $ETH before claiming the reward token. **Likelihood:** Medium The reward token will be lost only when the user claims $ETH before claiming the reward token. The lost amount is only the amount that has not been updated. The updated amount is still claimable.
Status	Resolved The Powaa team has resolved this issue by adding the `claimGov()` function into `claimETH()` and `claimETHPOWAA()` functions at line 182 and 101, respectively. Hence, the rewards would be updated when the user claimed $ETH. Fixed in commit `a86504357bfc53629c4d20024463a712fabc9f84`.

5.4.1. Description

The vault offers a reward to the users who stake assets in the vault. After the Merge has happened, the vault will swap the staked asset into `$ETH` and distribute it back proportionally to the staking user. The vault also provides the staking users with the governance token as a reward.

The users can freely claim their reward through the `claimGov()` function.

BaseTokenVault.sol

```
363 function claimGov() public nonReentrant updateReward(msg.sender) {
364   uint256 reward = rewards[msg.sender];
365   if (reward > 0) {
366     rewards[msg.sender] = 0;
367     IERC20(rewardsToken).safeTransfer(msg.sender, reward);
368     emit RewardPaid(msg.sender, reward);
369   }
370 }
```

The users can claim their `$ETH` after the vault has finished the execution of their `migrate()` functions. When

the users claim their $ETH the _balances state will be set to 0. The user can claim $ETH through the claimETH() function in the TokenVault contract and through the claimETHPOWAA() function in the GovLPVault contract.

The problem will arise when the users have claimed $ETH before claiming their reward. Since the _balances state has been set to 0, the following execution of the updateReward() function will not have any _balances to calculate the new reward.

TokenVault.sol

```
180 function claimETH() external whenMigrated {
181   uint256 claimable = _balances[msg.sender].mulDivDown(
182     ethSupply,
183     _totalSupply
184   );
185
186   if (claimable == 0) {
187     return;
188   }
189
190   _balances[msg.sender] = 0;
191
192   msg.sender.safeTransferETH(claimable);
193
194   emit ClaimETH(msg.sender, claimable);
195 }
```

5.4.2. Remediation

Inspex suggests updating the users' rewards before setting the caller's balance to 0 by adding the updateReward() modifier to the claimETH() and claimETHPOWAA() functions in GovLPVault and TokenVault contracts, respectively.

GovLPVault.sol

```
 99 function claimETHPOWAA() external whenMigrated updateReward(msg.sender){
100   uint256 claimableETH = _balances[msg.sender].mulDivDown(
101     ethSupply,
102     _totalSupply
103   );
104   uint256 claimablePOWAA = _balances[msg.sender].mulDivDown(
105     powaaSupply,
106     _totalSupply
107   );
108
109   if (claimableETH == 0 && claimablePOWAA == 0) {
110     return;
111   }
```

```
112
113   _balances[msg.sender] = 0;
114
115   msg.sender.safeTransferETH(claimableETH);
116   IERC20(rewardsToken).safeTransfer(msg.sender, claimablePOWAA);
117
118   emit ClaimETHPOWAA(msg.sender, claimableETH, claimablePOWAA);
119 }
```

TokenVault.sol

```
180 function claimETH() external whenMigrated updateReward(msg.sender){
181   uint256 claimable = _balances[msg.sender].mulDivDown(
182     ethSupply,
183     _totalSupply
184   );
185
186   if (claimable == 0) {
187     return;
188   }
189
190   _balances[msg.sender] = 0;
191
192   msg.sender.safeTransferETH(claimable);
193
194   emit ClaimETH(msg.sender, claimable);
195 }
```

5.5. Loop Over Unbounded Data Structure

ID	IDX-005
Target	Controller
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `migrate()` function will eventually be unusable due to excessive gas usage. Also, a failure in the execution of a vault will prevent other vaults from successfully executing the `migrate()` function. **Likelihood:** Low It is very unlikely that the `tokenVaults` size will be raised until the `migrate()` function is eventually unusable due to the maximum gas limit of the blockchain could handle. An error from a vault could be occurred by some condition in the vaults that are not ready to be migrated.
Status	**Acknowledged** The Powaa team has clarified that they have conducted an experiment to make sure that the execution could fit the block's gas limit.

5.5.1. Description

The `migrate()` function in the `Controller` contract is a core function of the vaults migration process, this function will loop in the `tokenVaults` list to call the `migrate()` function on every single vault in the list including the `govLPVault` state.

However, gas consumption will increase as the size of the `tokenVaults` increases, if the size of the `tokenVaults` is large enough to consume gas higher than the maximum gas limit of the blockchain could handle while executing the `migrate()` function at lines 111 and 112, the transaction will be reverted and never success.

Moreover, in the current design, the `migrate()` functions in every vault will be executed within a single loop, continuously. If an error or a revert has occurred in some vaults, the other vaults will be affected and cannot be migrated. So it could be more practical if the executor could decide the range of vaults to be migrated to avoid errors.

Controller.sol

```
102 function migrate() external {
103   if (tokenVaults.length == 0) revert Controller_NoVaults();
```

```
104   if (govLPVault == address(0)) revert Controller_NoGovLPVault();
105
106   uint256 vaultLength = tokenVaults.length;
107   address[] memory _vaults = new address[](vaultLength + 1);
108
109   for (uint256 index = 0; index < vaultLength; index++) {
110     _vaults[index] = tokenVaults[index];
111     ITokenVault(tokenVaults[index]).reduceReserve();
112     ITokenVault(tokenVaults[index]).migrate();
113   }
114
115   _vaults[vaultLength] = govLPVault;
116   ITokenVault(govLPVault).migrate();
117
118   uint256 executionFee = address(this).balance;
119   if (executionFee > 0) {
120     msg.sender.safeTransferETH(executionFee);
121
122     emit TransferFee(msg.sender, executionFee);
123   }
124   emit Migrate(_vaults);
125 }
```

5.5.2. Remediation

Inspex suggests implementing the pagination mechanism in the `migrate()` function, so the authorized executor should provide the desired offset of the `tokenVaults` list to execute the `migrate()` function to avoid maximum gas limitation.

Controller.sol

```
102 function migrate(uint256 startIndex, uint256 endIndex, bool
    isMigrateGovLPVault) external {
103   if (tokenVaults.length == 0) revert Controller_NoVaults();
104   if (govLPVault == address(0)) revert Controller_NoGovLPVault();
105
106   uint256 vaultLength = tokenVaults.length;
107   address[] memory _vaults = new address[](vaultLength + 1);
108
109   uint256 vaultLength = tokenVaults.length;
110   if (endIndex > vaultLength) endIndex = vaultLength;
111   if (isMigrateGovLPVault) endIndex = endIndex + 1;
112   address[] memory _vaults = new address[](endIndex - startIndex);
113
114   for (uint256 index = startIndex; index < endIndex; index++) {
115     _vaults[index] = tokenVaults[index];
116     ITokenVault(tokenVaults[index]).reduceReserve();
117     ITokenVault(tokenVaults[index]).migrate();
```

```
118     }
119
120     if (isMigrateGovLPVault){
121       _vaults[vaultLength] = govLPVault;
122       ITokenVault(govLPVault).migrate();
123     }
124
125     uint256 executionFee = address(this).balance;
126     if (executionFee > 0) {
127       msg.sender.safeTransferETH(executionFee);
128
129       emit TransferFee(msg.sender, executionFee);
130     }
131     emit Migrate(_vaults);
132   }
```

5.6. Insufficient Logging for Privileged Functions

ID	IDX-006
Target	SushiSwapLPVaultMigrator UniswapV2GovLPVaultMigrator
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged function's executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved The Powaa team has resolved this issue by emitting events for the execution of privileged functions in the commit `a86504357bfc53629c4d20024463a712fabc9f84`.

5.6.1. Description

A privileged function that is executable by the controlling parties is not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of the privileged function, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the owner can update the `tokenVaultOK` state by executing the `whitelistTokenVault()` function in the `SushiSwapLPVaultMigrator` contract, and no events are emitted, which results in the users not getting notified that the `tokenVaultOK` address has changed.

SushiSwapLPVaultMigrator.sol

```
90  function whitelistTokenVault(address tokenVault, bool isOk)
91    external
92    onlyOwner
93  {
94    tokenVaultOK[tokenVault] = isOk;
95  }
```

The privileged functions without sufficient logging are as follows:

File	Contract	Function
SushiSwapLPVaultMigrator.sol (L:90)	SushiSwapLPVaultMigrator	whitelistTokenVault()

UniswapV2GovLPVaultMigrator.sol (L:50)	UniswapV2GovLPVaultMigrator	whitelistTokenVault()

5.6.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

SushiSwapLPVaultMigrator.sol

```
89  event WhitelistTokenVault(address tokenVault, bool isOk);
90  function whitelistTokenVault(address tokenVault, bool isOk)
91    external
92    onlyOwner
93  {
94    tokenVaultOK[tokenVault] = isOk;
95    emit WhitelistTokenVault(tokenVault, isOk);
96  }
```

5.7. Improper Function Visibility

ID	IDX-007
Target	GovLPVault TokenVault POWAA
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The Powaa team has resolved this issue by changing the functions visibility into public in the commit `a86504357bfc53629c4d20024463a712fabc9f84`.

5.7.1. Description

Public functions that are never called internally by the contract itself should have external visibility. This improves the readability of the contract, allowing clear distinction between functions that are externally used and functions that are also called internally.

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

POWAA.sol

```
20 function maxTotalSupply() public view virtual returns (uint256) {
21   return _maxTotalSupply;
22 }
```

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

File	Contract	Function
GovLPVault.sol (L:38)	GovLPVault	initialize()
TokenVault.sol (L:55)	TokenVault	initialize()
POWAA.sol (L:20)	POWAA	maxTotalSupply()

5.7.2. Remediation

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

POWAA.sol

```
20 function maxTotalSupply() external view virtual returns (uint256) {
21   return _maxTotalSupply;
22 }
```

6. Appendix

6.1. About Inspex

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	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspeX

CYBERSECURITY PROFESSIONAL SERVICE