Vault

Smart Contract Audit Report
Prepared for AleSwap

Date Issued:	Aug 27, 2021
Project ID:	AUDIT2021014
Version:	v1.0
Confidentiality Level:	Public

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021014
Version	v1.0
Client	AleSwap
Project	Vault
Auditor(s)	Suvicha Buakhom Patipon Suwanbol
Author	Suvicha Buakhom
Reviewer	Pongsakorn Sommalai
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Aug 27, 2021	Full report	Suvicha Buakhom

Contact Information

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

Table of Contents

1. Executive Summary **1**
1.1. Audit Result 1
1.2. Disclaimer 1
2. Project Overview **2**
2.1. Project Introduction 2
2.2. Scope 3
3. Methodology **4**
3.1. Test Categories 4
3.2. Audit Items 5
3.3. Risk Rating 6
4. Summary of Findings **7**
5. Detailed Findings Information **9**
5.1. Token Draining Using SetVault() Function 9
5.2. Transaction Ordering Dependence 11
5.3. Use of Upgradable Contract 16
5.4. Centralized Control of State Variable 18
5.5. Liquidity Token Amount Miscalculation (_swapWBNBToLp) 19
5.6. Liquidity Token Amount Miscalculation (addLiquidity) 25
5.7. Improper Usage of SafeERC20.safeApprove() 30
5.8. Insufficient Logging for Privileged Functions 32
5.9. Improper Price Tolerance Design 34
5.10. Improper Function Visibility 37
5.11. Inexplicit Solidity Compiler Version 39
6. Appendix **41**
6.1. About Inspex 41
6.2. References 42

1. Executive Summary

As requested by AleSwap, Inspex team conducted an audit to verify the security posture of the Vault smart contracts between Aug 17, 2021 and Aug 18, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Vault 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 3 high, 2 medium, 2 low, 1 very low, and 3 info-severity issues. With the project team's prompt response, 3 high, 2 medium, 1 low, 1 very low, and 3 info-severity issues were resolved in the reassessment, while 1 low-severity issue was acknowledged by the team. Therefore, Inspex trusts that Vault 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.

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

AleSwap is a decentralized finance platform that relies on Binance Smart Chain (BSC). The main objective is to provide convenient exchange between any cryptocurrency holder and partner shop.

AleSwap Vault is the auto compound decentralized application that will automatically convert the users' tokens, add them into Warden's Masterchef contract, and compound the farming reward.

Scope Information:

Project Name	Vault
Website	https://aleswap.finance/
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Aug 17, 2021 - Aug 18, 2021
Reassessment Date	Aug 26, 2021

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: efd3f777b2e9aaa2c635d3d0f463f1884e998112)

Contract	Location (URL)
AleVault	https://github.com/aleswap-finance/aleswap-vault/blob/efd3f777b2/contracts/AleVault.sol
StrategyWardenLP	https://github.com/aleswap-finance/aleswap-vault/blob/efd3f777b2/contracts/strategies/Warden/StrategyWardenLP.sol
WardenSwapper	https://github.com/aleswap-finance/aleswap-vault/blob/efd3f777b2/contracts/swapper/WardenSwapper.sol

Reassessment: (Commit: 592f5f4a1cc195c144464c2b734a33d9ca182a50)

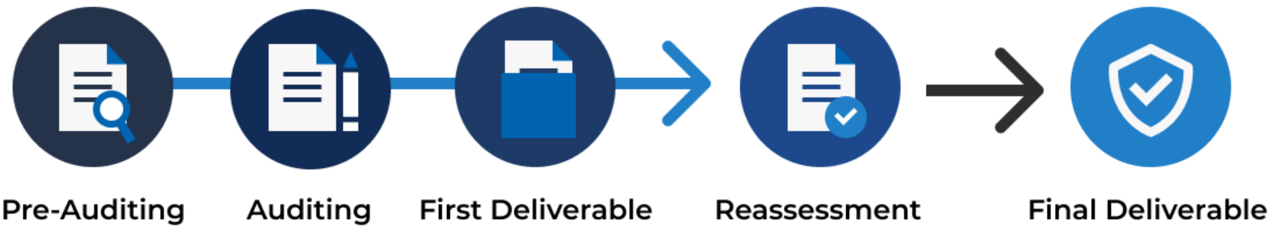
Contract	Location (URL)
AleVault	https://github.com/aleswap-finance/aleswap-vault/blob/592f5f4a1c/contracts/AleVault.sol
StrategyWardenLP	https://github.com/aleswap-finance/aleswap-vault/blob/592f5f4a1c/contracts/strategies/Warden/StrategyWardenLP.sol
WardenSwapper	https://github.com/aleswap-finance/aleswap-vault/blob/592f5f4a1c/contracts/swapper/WardenSwapper.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they are inherited 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 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

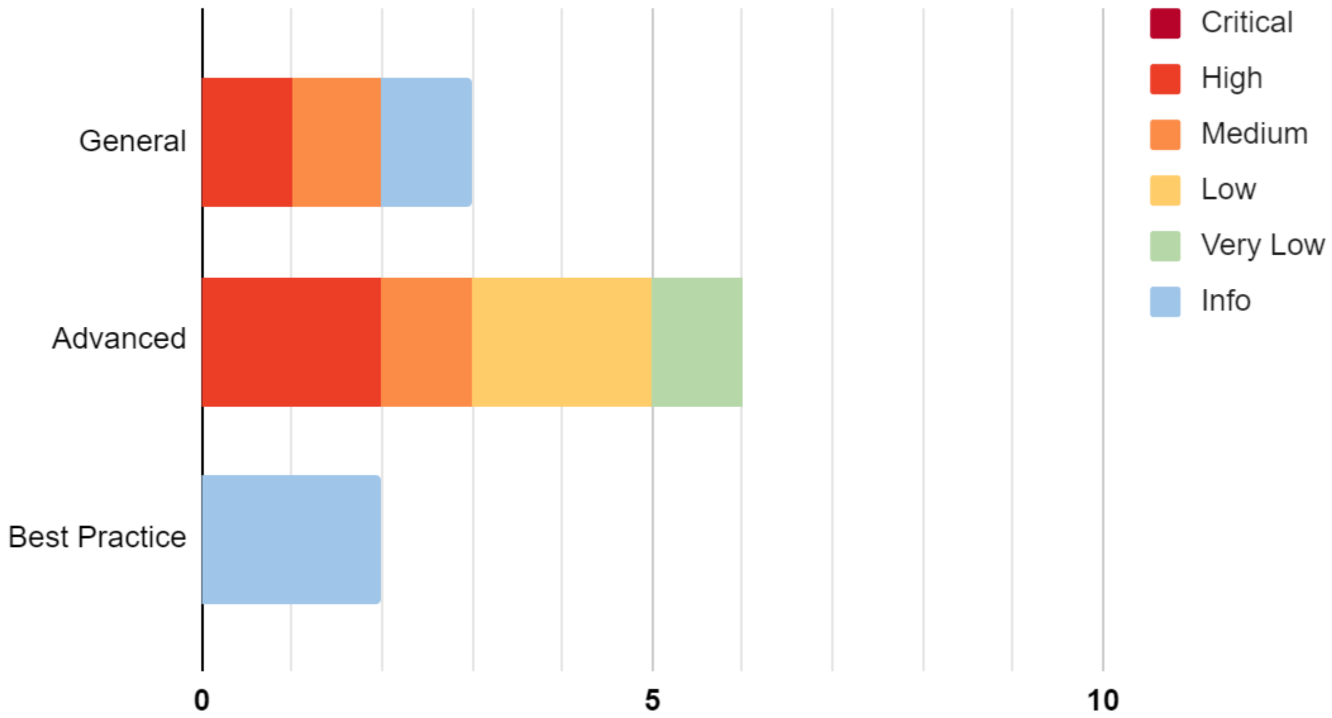
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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 11 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 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 Draining Using SetVault() Function	Advanced	High	Resolved
IDX-002	Transaction Ordering Dependence	General	High	Resolved
IDX-003	Use of Upgradable Contract	Advanced	High	Resolved *
IDX-004	Centralized Control of State Variable	General	Medium	Resolved
IDX-005	Liquidity Token Amount Miscalculation (_swapWBNBToLp)	Advanced	Medium	Resolved
IDX-006	Liquidity Token Amount Miscalculation (addliquidity)	Advanced	Low	Acknowledged
IDX-007	Improper Usage of SafeERC20.safeApprove()	Advanced	Low	Resolved
IDX-008	Insufficient Logging for Privileged Functions	Advanced	Very Low	Resolved
IDX-009	Improper Price Tolerance Design	General	Info	Resolved
IDX-010	Improper Function Visibility	Best Practice	Info	Resolved
IDX-011	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved

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

5. Detailed Findings Information

5.1. Token Draining Using SetVault() Function

ID	IDX-001
Target	StrategyWardenLP
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The owner can steal all the staked LP tokens from the `Masterchef` contract. **Likelihood:** Medium Only the contract owner can use this function; however, there is no restriction to prevent the owner from performing this attack.
Status	Resolved AleSwap team has resolved this issue as suggested in commit `2fc94869a72b74549bcc85645ec102ebca41613e`.

5.1.1. Description

In the `StrategyWardenLP` contract, the `withdraw()` function allows the `AleVault` contract to withdraw the LP token from the `Masterchef` contract for the users. These LP token will then be transferred to the `AleVault` contract to perform further actions.

StrategyWardenLP.sol

```
function withdraw(uint256 _amount) external returns (uint256) {
    require(msg.sender == vault, "!vault");

    uint256 wantBal = IERC20(want).balanceOf(address(this));

    if (wantBal < _amount) {
        IMasterChef(masterchef).withdraw(poolId, _amount.sub(wantBal));
        wantBal = IERC20(want).balanceOf(address(this));
    }

    if (wantBal > _amount) {
        wantBal = _amount;
    }

    if (tx.origin == owner() || paused()) {
        IERC20(want).safeTransfer(vault, wantBal);
```

```
        return wantBal;
    } else {
        uint256 withdrawalFee =
wantBal.mul(WITHDRAWAL_FEE).div(WITHDRAWAL_MAX);
        IERC20(want).safeTransfer(vault, wantBal.sub(withdrawalFee));
        return wantBal.sub(withdrawalFee);
    }
}
```

However, the owner can set the `AleVault` contract to be any address with the privilege from the `onlyOwner` modifier.

StrategyWardenLP.sol

```
function setVault(address _vault) external onlyOwner {
    vault = _vault;
}
```

This allows the newly set `vault` address to be able to execute the `withdraw()` function to drain all the staking LP token.

5.1.2. Remediation

Inspex suggests removing the `setVault()` function from the `StrategyWardenLP` contract, and setting the `vault` address within the `initialize()` function.

5.2. Transaction Ordering Dependence

ID	IDX-002
Target	StrategyWardenLP
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** High **Impact:** Medium The users and the platform will lose a portion of tokens from the front-running attack when compounding a reward. **Likelihood:** High It is likely that this issue will occur since there is no restriction and it is profitable for the attacker, so there is high motivation for the attack.
Status	Resolved AleSwap team has resolved this issue by setting minimum fee and liquidity token output on `harvest()` and adding `onlyHarvester` modifier in commit `2fc94869a72b74549bcc85645ec102ebca41613e`. The `harvest()` function will be called by the owner only to prevent malicious actions, e.g. setting minimum fee and liquidity token output to 0 by attackers.

5.2.1. Description

The `harvest()` function will collect the reward from `masterchef` and consume the reward through the `chargeFees()`, the `addLiquidity()`, and the `deposit()` functions.

StrategyWardenLP.sol

```
function harvest() external whenNotPaused onlyEOA {
    IMasterChef(masterchef).deposit(poolId, 0);
    chargeFees();
    addLiquidity();
    deposit();

    emit StratHarvest(msg.sender);
}
```

The reward will be swapped into $WBNB and a pair of LP tokens for adding liquidity to the pool.

During swapping, the `swapExactTokensForTokens()` function in the `chargeFees()` function and the `addLiquidity()` function are called by setting the `amountOutMin` as 0 as shown below:

StrategyWardenLP.sol

```
function chargeFees() internal {
    uint256 toWbnb = IERC20(wad).balanceOf(address(this)).mul(40).div(1000); // 4%
    WARDEN_ROUTER.swapExactTokensForTokens(toWbnb, 0, wadToWbnbRoute, address(this), now);

    uint256 wbnbBal = IERC20(wbnb).balanceOf(address(this));

    uint256 callFeeAmount = wbnbBal.mul(CALL_FEE).div(MAX_FEE); // 0.5%
    IERC20(wbnb).safeTransfer(msg.sender, callFeeAmount);

    uint256 aleswapFeeAmount = wbnbBal.mul(aleswapFee).div(MAX_FEE); // 3.5%
    IERC20(wbnb).safeTransfer(aleswapFeeRecipient, aleswapFeeAmount);

}
```

StrategyWardenLP.sol

```
function addLiquidity() internal {
    uint256 wadHalf = IERC20(wad).balanceOf(address(this)).div(2);

    if (lpToken0 != wad)
        WARDEN_ROUTER.swapExactTokensForTokens(wadHalf, 0, wadToLp0Route, address(this), now);

    if (lpToken1 != wad)
        WARDEN_ROUTER.swapExactTokensForTokens(wadHalf, 0, wadToLp1Route, address(this), now);

    uint256 lp0Bal = IERC20(lpToken0).balanceOf(address(this));
    uint256 lp1Bal = IERC20(lpToken1).balanceOf(address(this));
    WARDEN_ROUTER.addLiquidity(lpToken0, lpToken1, lp0Bal, lp1Bal, 1, 1, address(this), now);
}
```

This means the platform accepts all possible amounts of token received from swapping, including 0 amount. Therefore, the front running attack can be performed, resulting in a bad swapping rate and lower bounty.

For example, when the compounding is happening, the `swapExactTokensForTokens()` function was executed with 0 `amountOutMin` to swap claimed reward ($WAD) to $WBNB in `chargeFee()`, and swap remaining reward to `lpToken0` and `lpToken1` to make liquidity token for farming. The attacker can wait for the compounding transaction of the `StrategyWardenLP` contract, then submit a swapping transaction with the same token pair with a higher gas price to make the attacker's transaction complete before the compounding transaction.

The formula to calculate the price of tokens is as follows (swapping fee is ignored):

```
output = amountIn * reserveOut / (reserveIn + amountIn)
```

Currently, the token amount of the liquidity pool is in the table below:

Pool	reserve token 0	reserve token 1
$WBNB - $WAD	50 $WBNB	50 $WAD

The platform swaps 5 $WAD to $WBNB.

```
output = 5 * 50 / (50 + 5) = 4.54
```

As a result, swapping 5 $WAD will get 4.54 $ WBNB.

However, if this transaction is being front-run with the same input (5 $WAD), the platform will get less $WBNB (worse price).

The price in the liquidity pool is updated as below:

Pool	reserve token 0	reserve token 1
$WBNB - $WAD	45.43 $WBNB	55 $WAD

The platform then swaps 5 $WAD to $WBNB.

```
output = 5 * 45.43/ (55 + 5) = 3.78
```

As a result, swapping 5 $WAD after being a front-run attack will get 3.78 $WBNB.

Hence, the amount of received tokens from swapping is affected by the transaction ordering dependence.

5.2.2. Remediation

Inspex suggests calculating the expected amount out with the token price fetched from the price oracles and setting it to the `amountOutMin` parameter when swapping tokens before adding liquidity to the pool.

For example, the `slippage` state variable represents the current state of slippage amount (acceptable price range percentage from price impact), which should be set in advance before using the `swapExactTokensForTokens()` function. So, there will be the `setSlippage()` function for the owner to set the `slippage` value within the `MAX_SLIPPAGE` limit range.

StrategyWardenLP.sol

```
import "./interfaces/IOracle.sol";
contract StrategyWardenLP is OwnableUpgradeable, PausableUpgradeable {
    using SafeERC20 for IERC20;
    using SafeMath for uint256;

    IOracle public oracleBNB;
    IOracle public oracleLP;
    uint constant public MAX_SLIPPAGE = 1000; //for ex, 10%
    uint public slippage;
    ...
    ...
    event SetSlippage(uint _slippage);
    function setSlippage(uint _slippage) external onlyOwner {
        require(_slippage <= MAX_SLIPPAGE);
        slippage = _slippage;

        emit SetSlippage(_slippage);
    }
```

Assuming the `oracle.consult(wantedTokenAddress, toBeSwappedTokenBalance)` function returns the estimated amount of token to gain from swapping using the oracle price, that amount should be used to calculate the minimum amount of token acceptable, for example:

StrategyWardenLP.sol

```
function chargeFees() internal {
    uint256 toWbnb = IERC20(wad).balanceOf(address(this)).mul(40).div(1000); // 4%
    // oracle
    uint256 priceOracle = oracleBNB.consult(wbnb, IERC20(wad).balanceOf(address(this)));
    uint256 minAmountOut = priceOracle.sub(priceOracle.mul(slippage).div(10000));
    WARDEN_ROUTER.swapExactTokensForTokens(toWbnb, minAmountOut, wadToWbnbRoute, address(this), now);

```

```
    uint256 wbnbBal = IERC20(wbnb).balanceOf(address(this));

    uint256 callFeeAmount = wbnbBal.mul(CALL_FEE).div(MAX_FEE); // 0.5%
    IERC20(wbnb).safeTransfer(msg.sender, callFeeAmount);

    uint256 aleswapFeeAmount = wbnbBal.mul(aleswapFee).div(MAX_FEE); // 3.5%
    IERC20(wbnb).safeTransfer(aleswapFeeRecipient, aleswapFeeAmount);

}
```

StrategyWardenLP.sol

```
function addLiquidity() internal {
    uint256 wadHalf = IERC20(wad).balanceOf(address(this)).div(2);

    if (lpToken0 != wad)
        // oracle
        uint256 priceLp0 = oracleLP.consult(lpToken0, IERC20(wad).balanceOf(address(this)));
        uint256 minAmountOut0 = priceLp0.sub(priceLp0.mul(slippage).div(10000));
        WARDEN_ROUTER.swapExactTokensForTokens(wadHalf, minAmountOut0, wadToLp0Route, address(this), now);

    if (lpToken1 != wad)
        // oracle
        uint256 priceLp1 = oracleLP.consult(lpToken1, IERC20(wad).balanceOf(address(this)));
        uint256 minAmountOut1 = priceLp1.sub(priceLp1.mul(slippage).div(10000));
        WARDEN_ROUTER.swapExactTokensForTokens(wadHalf, minAmountOut1, wadToLp1Route, address(this), now);

    uint256 lp0Bal = IERC20(lpToken0).balanceOf(address(this));
    uint256 lp1Bal = IERC20(lpToken1).balanceOf(address(this));
    WARDEN_ROUTER.addLiquidity(lpToken0, lpToken1, lp0Bal, lp1Bal, 1, 1, address(this), now);
}
```

Please note that the remediations for other issues are not yet applied to the example above.

5.3. Use of Upgradable Contract

ID	IDX-003
Target	AleVault StrategyWardenLP WardenSwapper
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The logic of affected contracts can be arbitrarily changed. This allows the proxy owner to perform malicious actions e.g., stealing the users' funds anytime they want. **Likelihood: Medium** This action can be performed by the proxy owner without any restriction.
Status	**Resolved *** AleSwap team has mitigated this issue by implementing a timelock mechanism. The `StrategyWardenLP`, the `WardenSwapper`, and the `AleVault` contracts are deployed with the `TransparentUpgradeableProxy` contract that is owned by the `ProxyAdmin` contract. The `ProxyAdmin` is owned by the `Timelock` contract. Timelock contract with 3 days minimum delay: https://bscscan.com/address/0xA51eB447A2aa8d591512A1C046fd4862665207FE#code ProxyAdmin contract: https://bscscan.com/address/0xC5d4B19c14e1f47b8dD2dd1Bf8C865b2B6766537#code Ownership transfer of ProxyAdmin to Timelock contract: https://bscscan.com/tx/0x2d05a6b323bf06829c44693bce0fd0148de29b74f60111560c568e36a9d0fb70#eventlog TransparentUpgradeableProxy of StrategyWardenLP contract: https://bscscan.com/address/0x7c9a20eb60f32168b5abca76cd82aa3213b972cf#code Ownership transfer of StrategyWardenLP's TransparentUpgradeableProxy to ProxyAdmin contract: https://bscscan.com/tx/0x60b0a96627736b4e863c1732f5180a7726028db21d9ffd507b415cc96fe46337#eventlog Implementation of StrategyWardenLP contract: https://bscscan.com/address/0x44160a1a408a330db0c790628a6decffca4b3703#code TransparentUpgradeableProxy of WardenSwapper contract:

	https://bscscan.com/address/0x40E1e7809384C5736FC10A04306A22c605D0aE3A#code Ownership transfer of WardenSwapper's TransparentUpgradeableProxy contract: https://bscscan.com/tx/0x611c048ef044d69f21a6c1e5727c5835a8bd32451e891f5658a8a217cefb6e11#eventlog Implementation of WardenSwapper contract at: https://bscscan.com/address/0x9c5a79dc58be4268a2de41e0c13a16a19632d056#code TransparentUpgradeableProxy of AleVault contract: https://bscscan.com/address/0x401b3C4c1240B36B3B0bc126B704364C5A192D41#code Ownership transfer of AleVault's TransparentUpgradeableProxy contract: https://bscscan.com/tx/0x3fba02c966074da9f63cb90af10da9be47a8af5462b2d9aca9814056abbff0e8#eventlog Implementation of AleVault contract at: https://bscscan.com/address/0x65863669bca18ef022f1de6043e98102acb1f5fd#code

5.3.1. Description

Smart contracts are designed to be used as agreements that cannot be changed forever. When a smart contract is upgraded, the agreement can be changed from what was previously agreed upon.

As these smart contracts are upgradable, the logic of them can be modified by the owner anytime, making the smart contracts untrustworthy.

5.3.2. Remediation

Inspex suggests deploying the contracts without the proxy pattern or any solution that can make smart contracts upgradeable.

However, if the upgradability is needed, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient length of time to delay the changes e.g., 3 days. This allows the platform users to monitor the timelock and is notified of the potential changes being done on the smart contracts.

5.4. Centralized Control of State Variable

ID	IDX-004
Target	StrategyWardenLP
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** Medium **Impact:** Medium 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 by the owner; however, the changes are limited by fixed values in the smart contracts.
Status	Resolved AleSwap team has resolved this issue by removing the `setVault()` function in commit `2fc94869a72b74549bcc85645ec102ebca41613e`.

5.4.1. Description

The `setVault()` function in the `StrategyWardenLP` contract can be used by the contract owners to change the `vault` address. The change in the address of the vault is important to notify the user to acknowledge or be notified before the changes are effective.

5.4.2. 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 limiting the use of privileged function via the following options:

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

5.5. Liquidity Token Amount Miscalculation (_swapWBNBToLp)

ID	IDX-005
Target	WardenSwapper
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium A small amount of the users' token will be stuck in the contract on deposit, resulting in the users gaining a lower amount of shares. **Likelihood:** Medium This issue occurs whenever the price impact on one side of the `want` token pair (`token0` or `token1`) is more than the other.
Status	Resolved AleSwap team has resolved this issue as suggested in commit `2fc94869a72b74549bcc85645ec102ebca41613e`.

5.5.1. Description

The `_swapWBNBToLp()` function is called from the `swapTokenToLP()` and the `swapNativeToLp()` functions to transform the deposited token to the liquidity token to deposit to `masterchef`.

WardenSwapper.sol

```
function swapTokenToLP(address _from, uint amount, address _to, address
_recipient) public returns (uint) {
    IERC20(_from).safeTransferFrom(msg.sender, address(this), amount);
    _approveTokenIfNeeded(_from);

    uint bnbAmount;

    if (_from != WBNB)
        bnbAmount = _swapTokenForWBNB(_from, amount, address(this));
    else
        bnbAmount = amount;

    return _swapWBNBToLp(_to, bnbAmount, _recipient);
}

function swapNativeToLp(address _to, address _recipient) external payable
returns (uint) {
    IWETH(WBNB).deposit{value: msg.value}();
```

```

    return _swapWBNBToLp(_to, msg.value, _recipient);
}
```

The `_swapWBNBToLp()` function calculates the token amount for the liquidity adding (`token0` and `token1`) by dividing the $WBNB amount deposited by 2 in line 123 as shown below:

WardenSwapper.sol

```
115 function _swapWBNBToLp(address lpToken, uint amount, address receiver) private
returns (uint) {
116     IUniswapV2Pair pair = IUniswapV2Pair(lpToken);
117     address token0 = pair.token0();
118     address token1 = pair.token1();
119
120     uint token0Amount;
121     uint token1Amount;
122
123     uint swapValue = amount.div(2);
124
125     if (token0 == WBNB || token1 == WBNB) {
126         (token0,token1) = token0 == WBNB ? (token0,token1) : (token1,token0);
127
128         token0Amount = amount.sub(swapValue);
129         token1Amount = _swap(WBNB, swapValue,token1, address(this));
130     } else {
131         token0Amount = _swap(WBNB, swapValue,token0, address(this));
132         token1Amount = _swap(WBNB, amount.sub(swapValue),token1,
address(this));
133     }
134     _approveTokenIfNeeded(token0);
135     _approveTokenIfNeeded(token1);
136
137     ( , , amount) = WARDEN_ROUTER.addLiquidity(token0, token1, token0Amount,
token1Amount, 0, 0, receiver, block.timestamp);
138
139     return amount;
140 }
```

There are 2 potential issues when the deposited amount is divided in half:

Scenario 1: Different liquidity pool ratio

This scenario represents a situation where the users deposit a native token for the `want` token that is a pair of tokens $A and $B.

Assuming there are 2 pools and the ratio of $A:$B is 1:1 as in the table below:

Pool	reserve token 0	reserve token 1	ratio
$WBNB - $A	1000 $WBNB	2000 $A	1 $WBNB : 2 $A
$WBNB - $B	1000 $WBNB	3000 $B	1 $WBNB : 3 $B

The attacker can deposit 2 $BNB (wrapped to $WBNB). According to the `_swapWBNBToLp()` function, 2 $BNB will be divided into 1 $WBNB and 1 $WBNB. After swapping, the `WardenSwapper` contract gets ~2 $A and ~3 $B.

The required token amount to be added to the $A:$B liquidity pool is 1:1. Hence, there will be $B as a leftover token in the contract, resulting in fewer LP tokens to be received, and the leftover token will be stuck in the contract.

Scenario 2: Different swapping price impact

Assuming there are 2 pools and the ratio of $A:$B is 1:1 as in table below:

Pool	reserve token 0	reserve token 1	ratio
$WBNB - $A	1,000 $WBNB	100 $A	10 $WBNB : 1 $A
$WBNB - $B	10,000 $WBNB	1,000 $B	10 $WBNB : 1 $B

The attacker can deposit 200 $BNB (wrapped to $WBNB). Referring to `x * y = k` formula, swapping 100 $WBNB to $A in $WBNB:$A pool:

```
Before swapping: 1000 $WBNB * 100 $A = 100,000
After swapping: (1000 $WBNB + 100 $WBNB) * (100 $A - amountAOut) = 100,000
amountAOut = 100 - (100,000 / 1,100) = 9.1 $A
```

Swapping 100 $WBNB to $B in $WBNB:$B pool:

```
Before swapping: 10000 $WBNB * 100 $B = 10,000,000
After swapping: (10000 $WBNB + 100 $WBNB) * (100 $B - amountBOut) = 10,000,000
amountBOut = 100 - (10,000,000 / 10,100) = 9.91 $B
```

The `WardenSwapper` contract will get 9.1 $A and 9.91 $B. Spending all of these tokens for liquidity adding will leave 0.81 $B left in the contract since the $A:$B liquidity pool ratio is 1:1.

As a result, not all of the tokens are used for adding liquidity to the liquidity pool, causing the resulting amount of LP token to be less than what it should be, and the leftover token will be stuck in the contract.

5.5.2. Remediation

Inspex suggests calculating the exact amount of token needed before adding liquidity to the pool. In order for the tokens to be spent optimally.

For example, Inspex suggests implementing the `optimalDeposit()` function to calculate the exact token amount needed in order to swap to the target token.

WardenSwapper.sol

```
import "@uniswap/lib/contracts/libraries/Babylonian.sol";

/// @param amtA amount of token A desired to deposit
/// @param amtB amount of token B desired to deposit
/// @param resA amount of token A in reserve
/// @param resB amount of token B in reserve
function optimalDeposit(
    uint256 amtA,
    uint256 amtB,
    uint256 resA,
    uint256 resB
) internal pure returns (uint256 swapAmt, bool isReversed) {
    if (amtA * resB >= amtB * resA) {
        swapAmt = _optimalDepositA(amtA, amtB, resA, resB);
        isReversed = false;
    } else {
        swapAmt = _optimalDepositA(amtB, amtA, resB, resA);
        isReversed = true;
    }
}

/// @param amtA amount of token A desired to deposit
/// @param amtB amount of token B desired to deposit
/// @param resA amount of token A in reserve
/// @param resB amount of token B in reserve
// e - b / a * 2
// Math.sqrt((b * b) + d) - b / 9970 * 2
// (19970 * resA) * (19970 * resA) + (a*c*4) / 19950

// e-b / 9970
function _optimalDepositA(
    uint256 amtA,
    uint256 amtB,
    uint256 resA,
    uint256 resB
) private pure returns (uint256) {
    require(amtA * resB >= amtB * resA, "Reversed");

```

```
    uint256 a = 997;    // change fee here
    uint256 b = 1997 * resA;    // change fee here
    uint256 _c = (amtA * resB) - (amtB * resA);
    uint256 c = ((_c * 1000) / (amtB + resB)) * resA;

    uint256 d = a * c * 4;
    uint256 e = Babylonian.sqrt((b * b) + d);

    uint256 numerator = e - b;
    uint256 denominator = a * 2;

    return numerator / denominator;
}
```

Finally, we suggest applying the `optimalDeposit()` function in the `_swapWBNBToLp()` function.

WardenSwapper.sol

```
function _swapWBNBToLp(address lpToken, uint amount, address receiver) private
returns (uint) {
    IUniswapV2Pair pair = IUniswapV2Pair(lpToken);
    address token0 = pair.token0();
    address token1 = pair.token1();

    uint token0Amount;
    uint token1Amount;

    if (token0 == WBNB || token1 == WBNB) {
        (token0,token1) = token0 == WBNB ? (token0,token1) : (token1,token0);

        (uint256 lpToken0Reserve, uint256 lpToken1Reserve, ) =
IUniswapV2Pair(lpToken).getReserves();
        address otherToken = token0 == WBNB ? token1 : token0;
        uint256 swapAmt;
        (swapAmt, ) = optimalDeposit(
            IERC20(WBNB).balanceOf(address(this)),
            IERC20(otherToken).balanceOf(address(this)),
            lpToken0Reserve,
            lpToken1Reserve
        );
        token0Amount = amount.sub(swapAmt);
        token1Amount = _swap(WBNB, swapAmt, token1, address(this));


    } else {
        uint256 swapAmt;
        token0Amount = _swap(WBNB, amount, token0, address(this));

```

```
        (uint256 lpToken0Reserve, uint256 lpToken1Reserve, ) =
IUniswapV2Pair(lpToken).getReserves();
        (swapAmt, ) = optimalDeposit(
            IERC20(token0).balanceOf(address(this)),
            IERC20(token1).balanceOf(address(this)),
            lpToken0Reserve,
            lpToken1Reserve
        );
        _approveTokenIfNeeded(token0);
        token0Amount = token0Amount.sub(swapAmt);
        token1Amount = _swap(token0, swapAmt, token1, address(this));
    }

    _approveTokenIfNeeded(token1);

    ( , , amount) = WARDEN_ROUTER.addLiquidity(token0, token1, token0Amount,
token1Amount, 0, 0, receiver, block.timestamp);

    return amount;
}
```

Please note that the remediations for other issues are not yet applied to the example above.

5.6. Liquidity Token Amount Miscalculation (addLiquidity)

ID	IDX-006
Target	StrategyWardenLP
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Low A small amount of reward harvested will be left in the contract, resulting in a lower amount of tokens used in compounding. **Likelihood:** Medium This issue occurs whenever the price impact on one side of `lpToken0` or `lpToken1` is always more than the other or the liquidity pool does not include $WAD.
Status	**Acknowledged** AleSwap team has acknowledged this issue. The risks are quite low due to the amount of reward token that is being reinvested is small compared to the liquidity in the swap pool.

5.6.1. Description

The `addLiquidity()` function is called from the `harvest()` function to compound the pending reward to the liquidity pool.

StrategyWardenLP.sol

```
function harvest() external whenNotPaused onlyEOA {
    IMasterChef(masterchef).deposit(poolId, 0);
    chargeFees();
    addLiquidity();
    deposit();

    emit StratHarvest(msg.sender);
}
```

The `addLiquidity()` function calculates the amount of token that is used for the liquidity adding (`lpToken0` and `lpToken1`) by dividing the total reward token harvested from `masterchef` by 2 in line 154 as shown below:

StrategyWardenLP.sol

```
153 function addLiquidity() internal {
154     uint256 wadHalf = IERC20(wad).balanceOf(address(this)).div(2);
155
```

```
    if (lpToken0 != wad)
        WARDEN_ROUTER.swapExactTokensForTokens(wadHalf, 0, wadToLp0Route, address(this), now);

    if (lpToken1 != wad)
        WARDEN_ROUTER.swapExactTokensForTokens(wadHalf, 0, wadToLp1Route, address(this), now);

    uint256 lp0Bal = IERC20(lpToken0).balanceOf(address(this));
    uint256 lp1Bal = IERC20(lpToken1).balanceOf(address(this));
    WARDEN_ROUTER.addLiquidity(lpToken0, lpToken1, lp0Bal, lp1Bal, 1, 1, address(this), now);
}
```

There are 2 potential issues when the harvested reward is divided into half:

Scenario 1: Different liquidity pool ratio

This scenario represents a situation where the added liquidity pool does not consist of $WAD as a token for the liquidity, and the price of $WAD to each token in the LP pair is not the same as the ratio of a pair itself.

Assuming there are 2 pools and the ratio of $A:$B is 1:1 as in the table below:

Pool	reserve token 0	reserve token 1	ratio
$WAD - $A	1000 $WAD	2000 $A	1 $WAD : 2 $A
$WAD - $B	1000 $WAD	3000 $B	1 $WAD : 3 $B

If the harvested reward is 2 $WAD, according to the `addLiquidity()` function, 2 $WAD will be divided into 1 $WAD and 1 $WAD. After swapping, the `StrategyWardenLP` contract gets ~2 $A and ~3 $B.

However, the required token amount to be added to the $A:$B liquidity pool is 1:1. Hence, there will be $B as a leftover token in the contract, resulting in fewer LP tokens to be received.

Scenario 2: Different swapping price impact

Assuming there are 2 pools and the ratio of $A:$B is 1:1 as in the table below:

Pool	reserve token 0	reserve token 1	ratio
$WAD - $A	1,000 $WAD	100 $A	10 $WAD : 1 $A
$WAD - $B	10,000 $WAD	1,000 $B	10 $WAD : 1 $B

If the harvested reward is 200 $WAD, referring to `x * y = k` formula, the tokens will be swapped as follows:

Swapping 100 $WAD to $A in pool $WAD - $A:

```
Before swapping: 1000 $WAD * 100 $A = 100,000
After swapping: (1000 $WAD + 100 $WAD) * (100 $A - amountAOut) = 100,000
amountAOut = 100 - (100,000 / 1,100) = 9.1 $A
```

Swapping 100 $WAD to $B in pool $WAD - $B:

```
Before swapping: 10000 $WAD * 100 $B = 10,000,000
After swapping: (10000 $WAD + 100 $WAD) * (100 $B - amountBOut) = 10,000,000
amountBOut = 100 - (10,000,000 / 10,100) = 9.91 $B
```

As a result, not all tokens are used for adding liquidity to the liquidity pool, causing the amount of LP token to be less than the expected amount.

5.6.2. Remediation

Inspex suggests calculating the exact token amount needed before adding liquidity to the pool for the tokens to be spent optimally.

For example, the `optimalDeposit()` function should be implemented to calculate the exact amount of token needed in order to swap to the target token.

StrategyWardenLP.sol

```
import "@uniswap/lib/contracts/libraries/Babylonian.sol";

/// @param amtA amount of token A desired to deposit
/// @param amtB amount of token B desired to deposit
/// @param resA amount of token A in reserve
/// @param resB amount of token B in reserve
function optimalDeposit(
    uint256 amtA,
    uint256 amtB,
    uint256 resA,
    uint256 resB
) internal pure returns (uint256 swapAmt, bool isReversed) {
    if (amtA * resB >= amtB * resA) {
        swapAmt = _optimalDepositA(amtA, amtB, resA, resB);
        isReversed = false;
    } else {
        swapAmt = _optimalDepositA(amtB, amtA, resB, resA);
        isReversed = true;
    }
}

/// @param amtA amount of token A desired to deposit
```

```
/// @param amtB amount of token B desired to deposit
/// @param resA amount of token A in reserve
/// @param resB amount of token B in reserve
// e - b / a * 2
// Math.sqrt((b * b) + d) - b / 9970 * 2
// (19970 * resA) * (19970 * resA) + (a*c*4) / 19950

// e-b / 9970
function _optimalDepositA(
    uint256 amtA,
    uint256 amtB,
    uint256 resA,
    uint256 resB
) private pure returns (uint256) {
    require(amtA * resB >= amtB * resA, "Reversed");

    uint256 a = 997;     // change fee here
    uint256 b = 1997 * resA;     // change fee here
    uint256 _c = (amtA * resB) - (amtB * resA);
    uint256 c = ((_c * 1000) / (amtB + resB)) * resA;

    uint256 d = a * c * 4;
    uint256 e = Babylonian.sqrt((b * b) + d);

    uint256 numerator = e - b;
    uint256 denominator = a * 2;

    return numerator / denominator;
}
```

The `optimalDeposit()` function should then be used in the `addLiquidity()` function.

StrategyWardenLP.sol

```
function addLiquidity(uint256 amountOutMin) internal {
    address[] memory path = new address[](2);
    uint256 swapAmt;
    bool isReversed;

    if (lpToken0 != wad && lpToken1 != wad) {    // convert all to lp0

WARDEN_ROUTER.swapExactTokensForTokens(IERC20(wad).balanceOf(address(this)),
amountOutMin, wadToLp0Route, address(this), now);
        (uint256 lpToken0Reserve, uint256 lpToken1Reserve, ) =
IUniswapV2Pair(want).getReserves();
        (swapAmt, isReversed) = optimalDeposit(
            IERC20(lpToken0).balanceOf(address(this)),
            IERC20(lpToken1).balanceOf(address(this)),
```

```
            lpToken0Reserve,
            lpToken1Reserve
        );
        (path[0], path[1]) = isReversed ? (lpToken1, wbnb, lpToken0) : (lpToken0, wbnb, lpToken1);
    }
    else {
        (uint256 lpToken0Reserve, uint256 lpToken1Reserve, ) = IUniswapV2Pair(want).getReserves();
        address otherToken = lpToken0 == wad ? lpToken1 : lpToken0;
        (swapAmt, isReversed) = optimalDeposit(
            IERC20(wad).balanceOf(address(this)),
            IERC20(otherToken).balanceOf(address(this)),
            lpToken0Reserve,
            lpToken1Reserve
        );
        (path[0], path[1]) = isReversed ? (otherToken, wbnb, wad) : (wad, wbnb, otherToken);
    }
    WARDEN_ROUTER.swapExactTokensForTokens(swapAmt, 0, path, address(this), block.timestamp);

    uint256 lp0Bal = IERC20(lpToken0).balanceOf(address(this));
    uint256 lp1Bal = IERC20(lpToken1).balanceOf(address(this));
    WARDEN_ROUTER.addLiquidity(lpToken0, lpToken1, lp0Bal, lp1Bal, 1, 1, address(this), now);
}
```

Please note that the remediations for other issues are not yet applied to the example above.

5.7. Improper Usage of SafeERC20.safeApprove()

ID	IDX-007
Target	WardenSwapper
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The functions that require allowance will be unusable due to an unfulfilled condition. **Likelihood:** Low It is unlikely that the approval amount will be depleted.
Status	Resolved AleSwap team has resolved this issue as suggested in commit `2fc94869a72b74549bcc85645ec102ebca41613e`.

5.7.1. Description

In the `WardenSwapper` contract, it is used for handling token swapping. For the swapped tokens, it is required to give the allowance to the router to let it spend the tokens.

In this case, the `_approveTokenIfNeeded()` function is used to handle this scenario. However, it will only work for the first time only since it is required that the `allowance` amount must be 0 in order to execute the `IERC20(token).safeApprove()` function as in line number 98.

WardenSwapper.sol

```
97  function _approveTokenIfNeeded(address token) private {
98      if (IERC20(token).allowance(address(this), address(WARDEN_ROUTER)) == 0) {
99          IERC20(token).safeApprove(address(WARDEN_ROUTER), uint(- 1));
100     }
101 }
```

This means if the current allowance amount is less than the required allowance amount to spend (greater than zero), it will cause the contract to be unable to spend tokens as it intends to.

5.7.2. Remediation

Inspex suggests changing the `safeApprove()` function to `safeIncreaseAllowance()` function with the amount of allowance required, for example:

WardenSwapper.sol

```
function _approveTokenIfNeeded(address token, uint256 amount) private {
    uint256 currentAllowance = IERC20(token).allowance(address(this), 
address(WARDEN_ROUTER));
    if (currentAllowance < amount) {
        IERC20(token).safeIncreaseAllowance(address(WARDEN_ROUTER), amount - 
currentAllowance);
    }
}
```

5.8. Insufficient Logging for Privileged Functions

ID	IDX-008
Target	AleVault StrategyWardenLP
Category	Advanced Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' 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** AleSwap team has resolved this issue as suggested in commit `2fc94869a72b74549bcc85645ec102ebca41613e`.

5.8.1. Description

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

For example, the owner can set the fee recipient by executing the `setaleswapFeeRecipient()` function in the `StrategyWardenLP` contract, and no events are emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function
AleVault.sol (L:160)	AleVault	inCaseTokensGetStuck()
StrategyWardenLP.sol (L:203)	StrategyWardenLP	setVault()
StrategyWardenLP.sol (L:207)	StrategyWardenLP	setaleswapFeeRecipient()

5.8.2. Remediation

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

AleVault.sol

```
event InCaseTokensGetStuck(address _token);
function inCaseTokensGetStuck(address _token) external onlyOwner {
    require(_token != address(want()), "!token");

    uint256 amount = IERC20(_token).balanceOf(address(this));
    IERC20(_token).safeTransfer(msg.sender, amount);
    emit InCaseTokensGetStuck(_token);
}
```

5.9. Improper Price Tolerance Design

ID	IDX-009
Target	WardenSwapper
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved AleSwap team has resolved this issue as suggested in commit `592f5f4a1cc195c144464c2b734a33d9ca182a50`

5.9.1. Description

The price tolerance mechanism is not implemented in the `WardenSwapper` contract. Thus, the front running attack can be performed since it accepts all price impacts that would occur, resulting in a bad swapping rate and a lower bounty.

However, the price impact does not truly have any effect since these functions are called by other functions which validate the swapped token amounts, and revert the transaction if it is less than the acceptable threshold.

The validation happens in the `AleVault` contract, which accepts `_minAmount` as the price tolerance threshold.

AleVault.sol

```
function depositFromNative(uint256 _minAmount) public payable nonReentrant {
    uint _amount = swapper.swapNativeToLp{value:
msg.value}(address(want()),address(this));
    require(_amount >= _minAmount, "INSUFFICIENT_OUTPUT_AMOUNT");

    _depositAndMintVaultToken(_amount);
}
```

If new contracts are implemented and integrated with the `WardenSwapper` contract, the front running attack can be performed if the price impact is not validated properly in the caller contract.

The following table represents the entry point functions that can be used to perform token swapping.

Contract	Function
WardenSwapper (L:32)	swapLpToToken()
WardenSwapper (L:67)	swapLpToNative()
WardenSwapper (L:75)	swapTokenToLP()
WardenSwapper (L:89)	swapNativeToLp()

5.9.2. Remediation

Inspex suggests that the validation of the swapped token amount should be implemented on the entry point functions at the `WardenSwapper` contract to prevent this problem from happening in any integrated contract, for example:

WardenSwapper.sol

```
32 function swapLpToToken(address _from, uint amount, address _to, uint _amountOutMin, address _recipient) public returns (uint) {
33     IERC20(_from).safeTransferFrom(msg.sender, address(this), amount);
34     _approveTokenIfNeeded(_from);
35 
36     IUniswapV2Pair pair = IUniswapV2Pair(_from);
37     address token0 = pair.token0();
38     address token1 = pair.token1();
39 
40     (uint token0Amount,uint token1Amount) = WARDEN_ROUTER.removeLiquidity(token0, token1, amount, 0, 0, address(this), block.timestamp);
41 
42     if (_to != CRAFT) {
43         if (token0 != _to)
44             token0Amount = _swap(token0,token0Amount,_to,address(this));
45         if (token1 != _to)
46             token1Amount = _swap(token1,token1Amount,_to,address(this));
47 
48         amount = token0Amount.add(token1Amount);
49         IERC20(_to).safeTransfer(_recipient, amount);
50         require(amount >= _amountOutMin, "INSUFFICIENT_OUTPUT_AMOUNT");
51         return amount;
52     } else {
53         uint bnbAmount;
54         if (token0 != WBNB)
55             bnbAmount = _swapTokenForWBNB(token0, token0Amount, address(this));
56         else
57             bnbAmount = token0Amount;
```

```
        if (token1 != WBNB)
            bnbAmount = bnbAmount.add(_swapTokenForWBNB(token1, token1Amount, address(this)));
        else
            bnbAmount = bnbAmount.add(token1Amount);

        uint craftAmount = _swapWBNBtoCRAFT(bnbAmount, _recipient);
        require(craftAmount >= _amountOutMin, "INSUFFICIENT_OUTPUT_AMOUNT");
        return craftAmount;
    }
}
```

5.10. Improper Function Visibility

ID	IDX-010
Target	AleVault StrategyWardenLP WardenSwapper
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved AleSwap team has resolved this issue as suggested in commit `2fc94869a72b74549bcc85645ec102ebca41613e`.

5.10.1. 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.

For example, the following source code shows that the `getPricePerFullShare()` function of the `AleVault` contract is set to public and it is never called from any internal function.

AleVault.sol

```
50 function getPricePerFullShare() public view returns (uint256) {
51     return totalSupply() == 0 ? 1e18 : balance().mul(1e18).div(totalSupply());
52 }
```

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

File	Contract	Function
AleVault.sol (L:50)	AleVault	getPricePerFullShare()
AleVault.sol (L:70)	AleVault	depositFromNative()
AleVault.sol (L:88)	AleVault	depositAllFromToken()
StrategyWardenLP.sol (L:168)	StrategyWardenLP	balanceOf()

StrategyWardenLP.sol (L:184)	StrategyWardenLP	panic()
WardenSwapper.sol (L:75)	WardenSwapper	swapTokenToLP()

5.10.2. Remediation

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

AleVault.sol

```
function getPricePerFullShare() external view returns (uint256) {
    return totalSupply() == 0 ? 1e18 : balance().mul(1e18).div(totalSupply());
}
```

5.11. Inexplicit Solidity Compiler Version

ID	IDX-011
Target	AleVault StrategyWardenLP WardenSwapper
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved AleSwap team has resolved this issue as suggested in commit `2fc94869a72b74549bcc85645ec102ebca41613e`.

5.11.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues, for example:

AleVault.sol

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;
```

The following table contains all targets which the inexplicit compiler version is declared.

Contract	Version
AleVault	^0.6.0
StrategyWardenLP	^0.6.0
WardenSwapper	^0.6.12

5.11.2. Remediation

Inspex suggests fixing the solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.6 is v0.6.12, for example:

AleVault.sol

```
//SPDX-License-Identifier: MIT
pragma solidity 0.6.12;
```

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

6.2. References

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

inspex

CYBERSECURITY PROFESSIONAL SERVICE