AutoCompound

Smart Contract Audit Report
Prepared for KillSwitch

Date Issued:	Oct 21, 2021
Project ID:	AUDIT2021010
Version:	v2.0
Confidentiality Level:	Public

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021010
Version	v2.0
Client	KillSwitch
Project	AutoCompound
Auditor(s)	Suvicha Buakhom Peeraphut Punsuwan
Author	Suvicha Buakhom
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
2.0	Oct 21, 2021	Update reassessment scope	Suvicha Buakhom
1.0	Oct 20, 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

As requested by KillSwitch, Inspex team conducted an audit to verify the security posture of the AutoCompound smart contracts between Oct 8, 2021 and Oct 14, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of AutoCompound 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.

The AutoCompound smart contract in this audit is a new improved version of the previously launched alpha version. There are multiple improvements to the functionalities and the token distribution mechanism. Together with the improvements, an invoking of a smart contract without a publicly available source code is added to the codebase. The KillSwitch team has clarified that the contract contains KillSwitch's proprietary code, and decided not to publish the source code, but implemented measures to control the impact from the execution instead.

1.1. Audit Result

In the initial audit, Inspex found 3 high, 3 medium, 3 low, 1 very low, and 2 info-severity issues. With the project team's prompt response, 3 high, 3 medium, 1 low, 1 very low, and 2 info-severity issues were resolved in the reassessment, while 2 low-severity issues were acknowledged by the team. Therefore, Inspex trusts that AutoCompound 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), 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

KillSwitch is a smart yield farming aggregator targeting to increase convenience and security for Binance Smart Chain yield farmers. Users are free to manage their funds and sell their high risk coins instantly in one click.

AutoCompound is a mechanism to bring users' investment funds to invest in the integrated yield farming protocols and use the reward gained for reinvestment to generate compound interest for higher yield.

Scope Information:

Project Name	AutoCompound
Website	https://app.killswitch.finance/
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Oct 8, 2021 - Oct 14, 2021
Reassessment Date	Oct 18, 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: 75a67eac40bd176b63bc0c6c939e0f63b987247f)

Contract	Location (URL)
PronteraV2	https://github.com/killswitchofficial/auto-compound-v2/blob/75a67eac40/PronteraV2.sol
PronteraReserve	https://github.com/killswitchofficial/auto-compound-v2/blob/75a67eac40/PronteraReserve.sol
PancakeByalanLP	https://github.com/killswitchofficial/auto-compound-v2/blob/75a67eac40/PancakeByalanLP.sol
IzludeV2	https://github.com/killswitchofficial/auto-compound-v2/blob/75a67eac40/IzludeV2.sol
GasPrice	https://github.com/killswitchofficial/auto-compound-v2/blob/75a67eac40/GasPrice.sol
FeeKafra	https://github.com/killswitchofficial/auto-compound-v2/blob/75a67eac40/FeeKafra.sol
Emperium	https://github.com/killswitchofficial/auto-compound-v2/blob/75a67eac40/Emperium.sol
AllocKafra	https://github.com/killswitchofficial/auto-compound-v2/blob/75a67eac40/AllocKafra.sol

Reassessment: (Commit: 92626a1dcfc55a28afdf4f996d600fe8bbfd6efd)

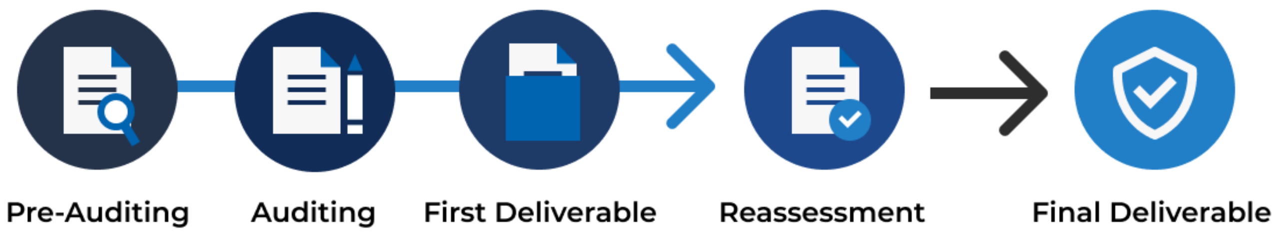
Contract	Location (URL)
PronteraV2	https://github.com/killswitchofficial/auto-compound-v2/tree/92626a1dcf/PronteraV2.sol
PronteraReserve	https://github.com/killswitchofficial/auto-compound-v2/tree/92626a1dcf/PronteraReserve.sol
PancakeByalanLP	https://github.com/killswitchofficial/auto-compound-v2/tree/92626a1dcf/PancakeByalanLP.sol
IzludeV2	https://github.com/killswitchofficial/auto-compound-v2/tree/92626a1dcf/IzludeV2.sol
GasPrice	https://github.com/killswitchofficial/auto-compound-v2/tree/92626a1dcf/GasPrice.sol
FeeKafra	https://github.com/killswitchofficial/auto-compound-v2/tree/92626a1dcf/FeeKafra.sol
Emperium	https://github.com/killswitchofficial/auto-compound-v2/tree/92626a1dcf/Emperium.sol
AllocKafra	https://github.com/killswitchofficial/auto-compound-v2/tree/92626a1dcf/AllocKafra.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
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Use of Upgradable Contract Design
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

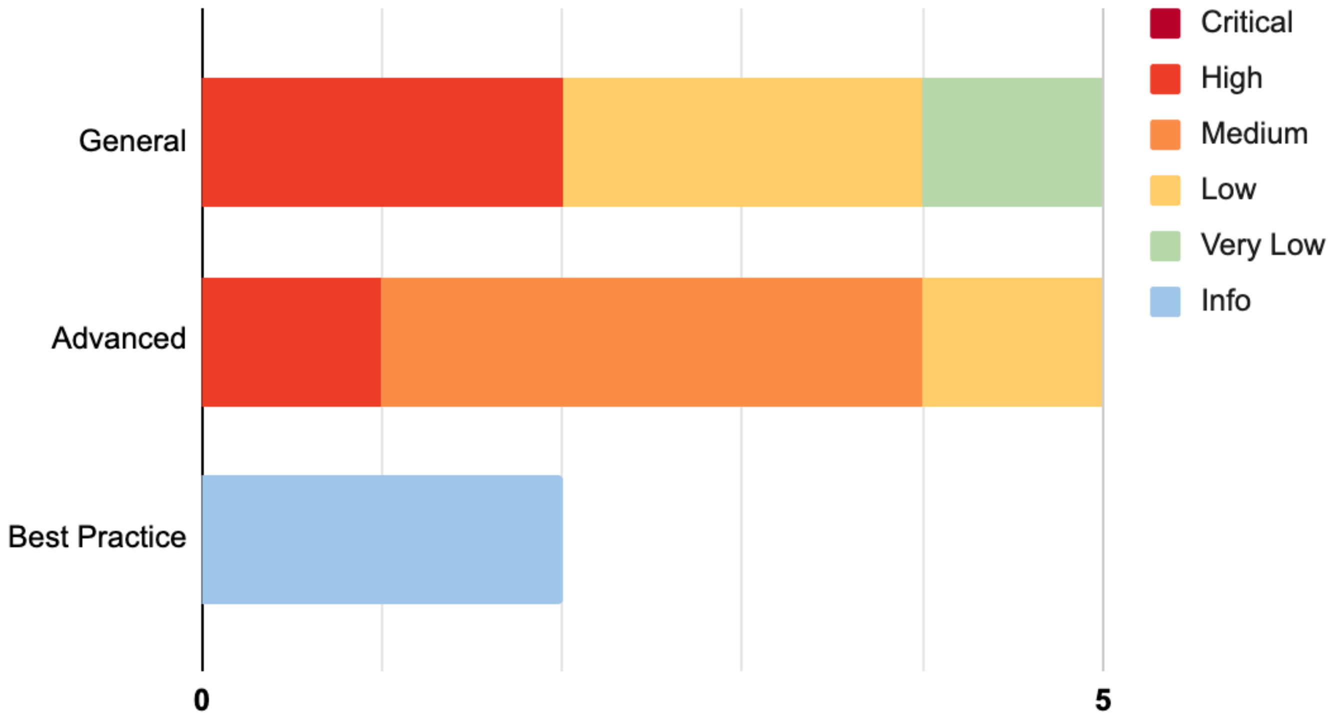
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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 12 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	Invoking of Unreliable Smart Contract	General	High	Resolved *
IDX-002	Centralized Control of State Variable	General	High	Resolved *
IDX-003	Changing of Strategy Contract Implementation	Advanced	High	Resolved *
IDX-004	Design Flaw in emergencyWithdraw() Function	Advanced	Medium	Resolved
IDX-005	Improper Reward Calculation in PronteraV2	Advanced	Medium	Resolved *
IDX-006	Improper Reward Calculation in Emperium	Advanced	Medium	Resolved
IDX-007	Transaction Ordering Dependence	General	Low	Acknowledged
IDX-008	Design Flaw in massUpdatePool() Function	General	Low	Resolved *
IDX-009	Liquidity Token Amount Miscalculation	Advanced	Low	Acknowledged
IDX-010	Insufficient Logging for Privileged Function	General	Very Low	Resolved
IDX-011	Improper Function Visibility	Best Practice	Info	Resolved
IDX-012	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved

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

5. Detailed Findings Information

5.1. Invoking of Unreliable Smart Contract

ID	IDX-001
Target	PronteraV2
Category	General Smart Contract Vulnerability
CWE	CWE-829: Inclusion of Functionality from Untrusted Control Sphere
Risk	**Severity:** High **Impact:** High The `juno` address may include malicious functions that can cause a multitude of unknown risks to the users. This results in potential monetary loss for the users and reputation damage to the platform. **Likelihood:** Medium The `Juno` contract can be set by the `junoGuide` role of the `PronteraV2` contract, which is defined by the contract owner. It is possible that the `junoGuide` will set the `juno` address to gain unfair benefit.
Status	Resolved * KillSwitch team has clarified that the `Juno` contract contains proprietary code for smart automated LP adding/removing and token swapping. Thus, KillSwitch team cannot publish the `Juno` contract source code. However, KillSwitch team has already mitigated the issue by setting the expected minimum amount of the tokens received from the `Juno` contract. This can help mitigate the impact of a malicious `Juno` contract.

5.1.1. Description

In the `PronteraV2` contract there are many functions that invoke the `Juno` contract that have no source code published.

For example, the following source code is an example of the `depositToken()` function that transfers tokens to the `Juno` contract and calls its function with the information from the `data` parameter, allowing the calling of an arbitrary function in the `juno` contract to perform arbitrary actions on the user's tokens.

PronteraV2.sol

```
function depositToken(
    address izlude,
    IERC20[] calldata tokens,
    uint256[] calldata tokenAmounts,
    uint256 amountOutMin,
```

```
    uint256 deadline,
    bytes calldata data
) external nonReentrant ensure(deadline) {
    require(tokens.length == tokenAmounts.length, "length mismatch");
    PoolInfo storage pool = poolInfo[izlude];
    IERC20 want = pool.want;
    uint256 beforeBal = want.balanceOf(address(this));

    for (uint256 i = 0; i < tokens.length; i++) {
        require(_safeERC20TransferIn(tokens[i], tokenAmounts[i]) ==
tokenAmounts[i], "!amount");
        if (tokens[i] != want) {
            tokens[i].safeTransfer(juno, tokenAmounts[i]);
        }
    }
    juno.functionCall(data, "juno: failed");

    uint256 amount = want.balanceOf(address(this)) - beforeBal;
    require(amount >= amountOutMin, "insufficient output amount");
    _deposit(msg.sender, izlude, want, amount);
    emit DepositToken(msg.sender, izlude, tokenAmounts, amount);
}
```

The unverifiable contract may have some hidden side effects that cannot be identified, resulting in a lack of transparency and posing unknown risks to the users.

5.1.2. Remediation

Inspex suggests disclosing the source code of the `Juno` contract or verifying the contract source code on the block explorer.

5.2. Centralized Control of State Variable

ID	IDX-002
Target	AllocKafra Emperium FeeKafra GasPrice IzludeV2
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
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 and can cause significant monetary loss to the users. **Likelihood: Medium** There is nothing to restrict the changes from being done; however, these actions can only be performed by the contract owner.
Status	Resolved * KillSwitch team has confirmed that the team will implement the timelock mechanism with 1 day delay when deploying the smart contracts to mainnet. The users will be able to monitor the timelock for the execution of critical functions and act accordingly if they are being misused. At the time of the reassessment, the contracts are not deployed yet, so the use of timelock is not confirmed. For the platform users, please verify that the timelock is properly deployed before using this 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, as the contract is not yet deployed, there is potentially 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	Role / Modifier
AllocKafra.sol (L:78)	AllocKafra	renounceOwnership()	onlyOwner
AllocKafra.sol (L:86)	AllocKafra	transferOwnership()	onlyOwner

AllocKafra.sol (L:146)	AllocKafra	setLimitAllocation()	onlyOwner
Emperium.sol (L:78)	Emperium	renounceOwnership()	onlyOwner
Emperium.sol (L:86)	Emperium	transferOwnership()	onlyOwner
Emperium.sol (L:684)	Emperium	add()	onlyOwner
Emperium.sol (L:697)	Emperium	set()	onlyOwner
Emperium.sol (L:794)	Emperium	setKSWPerSecond()	onlyOwner
FeeKafra.sol (L:78)	FeeKafra	renounceOwnership()	onlyOwner
FeeKafra.sol (L:86)	FeeKafra	transferOwnership()	onlyOwner
FeeKafra.sol (L:556)	FeeKafra	setWithdrawFee()	onlyOwner
FeeKafra.sol (L:563)	FeeKafra	setTreasuryFeeWithdraw()	onlyOwner
FeeKafra.sol (L:569)	FeeKafra	setKSWFeeWithdraw()	onlyOwner
FeeKafra.sol (L:575)	FeeKafra	setKSWFeeRecipient()	onlyOwner
FeeKafra.sol (L:580)	FeeKafra	setTreasuryFeeRecipient()	onlyOwner
GasPrice.sol (L:78)	GasPrice	renounceOwnership()	onlyOwner
GasPrice.sol (L:86)	GasPrice	transferOwnership()	onlyOwner
GasPrice.sol (L:109)	GasPrice	setMaxGasPrice()	onlyOwner
IzludeV2.sol (L:78)	IzludeV2	renounceOwnership()	onlyOwner
IzludeV2.sol (L:86)	IzludeV2	transferOwnership()	onlyOwner
IzludeV2.sol (L:733)	IzludeV2	setFeeKafra()	onlyOwner
IzludeV2.sol (L:738)	IzludeV2	setAllocKafra()	onlyOwner
IzludeV2.sol (L:743)	IzludeV2	setTva()	tva
IzludeV2.sol (L:834)	IzludeV2	upgradeStrategy()	tva
IzludeV2.sol (L:853)	IzludeV2	inCaseTokensGetStuck()	onlyOwner
PancakeByalanLP.sol (L:951)	ByalanIsland	renounceOwnership()	onlyOwner
PancakeByalanLP.sol (L:959)	ByalanIsland	transferOwnership()	onlyOwner
PancakeByalanLP.sol (L:1145)	ByalanIsland	setHydra()	onlyHydra

PancakeByalanLP.sol (L:1150)	ByalanIsland	setUnirouter()	onlyOwner
PancakeByalanLP.sol (L:1155)	ByalanIsland	setIzlude()	onlyOwner
PancakeByalanLP.sol (L:1160)	ByalanIsland	setTreasuryFeeRecipient()	onlyOwner
PancakeByalanLP.sol (L:1165)	ByalanIsland	setKswFeeRecipient()	onlyOwner
PancakeByalanLP.sol (L:1170)	ByalanIsland	setHarvester()	onlyOwner
PancakeByalanLP.sol (L:1175)	ByalanIsland	setGasPrice()	onlyHydra
PancakeByalanLP.sol (L:1202)	Sailor	setTotalFee()	onlyOwner
PancakeByalanLP.sol (L:1209)	Sailor	setCallFee()	onlyOwner
PancakeByalanLP.sol (L:1215)	Sailor	setTreasuryFee()	onlyOwner
PancakeByalanLP.sol (L:1221)	Sailor	setKSWFee()	onlyOwner
PronteraReserve.sol (L:78)	PronteraReserve	renounceOwnership()	onlyOwner
PronteraReserve.sol (L:86)	PronteraReserve	transferOwnership()	onlyOwner
PronteraReserve.sol (L:571)	PronteraReserve	setProntera()	onlyOwner
PronteraReserve.sol (L:578)	PronteraReserve	setEmperium()	onlyOwner
PronteraV2.sol (L:78)	PronteraV2	renounceOwnership()	onlyOwner
PronteraV2.sol (L:86)	PronteraV2	transferOwnership()	onlyOwner
PronteraV2.sol (L:900)	PronteraV2	add()	onlyOwner
PronteraV2.sol (L:928)	PronteraV2	set()	onlyOwner
PronteraV2.sol (L:1334)	PronteraV2	setKSWPerSecond()	onlyOwner
PronteraV2.sol (L:1340)	PronteraV2	setJuno()	junoGuide
PronteraV2.sol (L:1346)	PronteraV2	setJunoGuide()	onlyOwner

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 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 sufficient amount of time, e.g., 24 hours

5.3. Changing of Strategy Contract Implementation

ID	IDX-003
Target	IzludeV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The tokens deposited into the contract by the users can be drained by the owner of the `IzludeV2` contract, causing monetary loss for the users. **Likelihood: Medium** Only the owner can set the `tva` address that can call the `upgradeStrategy()` function. However, there is no restriction to prevent the owner and `tva` address from performing this attack to gain unfair benefit.
Status	**Resolved *** KillSwitch team has confirmed that the team will mitigate this issue by implementing the timelock mechanism with 7 days delay when deploying the smart contracts to mainnet. The users will be able to monitor the timelock for the upgrade of the contract and act accordingly if it is being misused. At the time of reassessment, the contracts are not deployed yet, so the use of timelock is not confirmed. For the platform users, please verify that the timelock is properly deployed before using this platform.

5.3.1. Description

The `upgradeStrategy()` function can be used by the `tva` role to change the address of the `byalan` which is the contract that contains the reinvestment logic.

IzludeV2.sol

```
function upgradeStrategy(address implementation) external {
    require(tva == msg.sender, "!TVA");
    require(address(this) == IByalan(implementation).izlude(), "invalid byalan");
    require(want == IERC20(byalan.want()), "invalid byalan want");

    // retire old byalan
    byalan.retireStrategy();

    // new byalan
    byalan = IByalan(implementation);
```

```
    earn();

    emit UpgradeStrategy(implementation);
}
```

When the `upgradeStrategy()` occurs, all of the balance of the `want` token will be withdrawn from `MASTERCHEF` and sent back to the `IzludeV2` contract.

PancakeByalanLP.sol

```
function retireStrategy() external override onlyIzlude {
    IMasterChef(MASTERCHEF).emergencyWithdraw(pid);

    uint256 wantBal = IERC20(want).balanceOf(address(this));
    IERC20(want).transfer(izlude, wantBal);
}
```

Then, the `want` from `IzludeV2` will be sent to the new `byalan` contract.

IzludeV2.sol

```
function earn() public {
    want.safeTransfer(address(byalan), want.balanceOf(address(this)));
    byalan.deposit();
}
```

The `byalan` address can be set to any address by `tva` role, so the `tva` role can change `byalan` address to a contract with malicious implementation and drain all `want` tokens from `IzludeV2` contract.

5.3.2. Remediation

Inspex suggests removing the `upgradeStrategy()` function to keep the integrity of the smart contract.

However, if the strategy (`byalan`) is required to be modifiable, Inspex suggests mitigating this issue by limiting the use of `upgradeStrategy()` 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, e.g., 24 hours

5.4. Design Flaw in emergencyWithdraw() Function

ID	IDX-004
Target	PronteraV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High The users can not withdraw deposited funds in the `PronteraV2` contract when the emergency case occurs. **Likelihood:** Low This issue occurs when the users' share is kept by the store and the emergency case happens.
Status	Resolved KillSwitch team has resolved this issue as suggested in commit `c9b001b2597ba9b173d63f5b9d8e642cf35a709e`.

5.4.1. Description

In the `PronteraV2` contract, the users can give an allowance to the `store` address to use the `storeKeepJellopy()` function to keep users' share (`jellopy`). This allows users' `jellopy` to be used by other contracts in an upcoming functionality.

PronteraV2.sol

```
function storeKeepJellopy(
    address _user,
    address izlude,
    uint256 amount
) external {
    require(amount > 0, "invalid amount");
    UserInfo storage user = userInfo[izlude][_user];
    user.jellopy -= amount;
    user.storedJellopy += amount;
    jellopyStorage[_user][izlude][msg.sender] += amount;

    uint256 currentAllowance = _storeAllowances[_user][izlude][msg.sender];
    require(currentAllowance >= amount, "keep amount exceeds allowance");
    unchecked {
        _approveStore(_user, izlude, msg.sender, currentAllowance - amount);
    }
    emit StoreKeepJellopy(_user, izlude, msg.sender, amount);
```

```
}
```

The kept share can be returned to the user by the `store` address calling the `storeReturnJellopy` function.

PronteraV2.sol

```
function storeReturnJellopy(
    address _user,
    address izlude,
    uint256 amount
) external {
    require(amount > 0, "invalid amount");
    UserInfo storage user = userInfo[izlude][_user];
    jellopyStorage[_user][izlude][msg.sender] -= amount;
    user.storedJellopy -= amount;
    user.jellopy += amount;
    emit StoreReturnJellopy(_user, izlude, msg.sender, amount);
}
```

In the emergency case, the `emergencyWithdraw()` function requires the `user.storedJellopy` equal to 0 to withdraw.

PronteraV2.sol

```
function emergencyWithdraw(address izlude) external {
    PoolInfo storage pool = poolInfo[izlude];
    UserInfo storage user = userInfo[izlude][msg.sender];
    require(user.storedJellopy == 0, "stored jellopy must be 0");

    uint256 jellopy = user.jellopy;
    user.jellopy = 0;
    user.rewardDebt = 0;
    if (jellopy > 0) {
        IERC20 want = pool.want;
        uint256 wantBefore = want.balanceOf(address(this));
        IIzludeV2(izlude).withdraw(msg.sender, jellopy);
        uint256 wantAfter = want.balanceOf(address(this));
        want.safeTransfer(msg.sender, wantAfter - wantBefore);
    }
    emit EmergencyWithdraw(msg.sender, izlude, jellopy);
}
```

However, users can not return the kept share by themselves. If the `store` does not return the kept share to users, the users will not be able to withdraw any of their funds by calling the `emergencyWithdraw()` function.

5.4.2. Remediation

Inspex suggests removing the require `user.storedJellopy == 0` statement and allow the withdrawal of the unkept funds, for example:

PronteraV2.sol

```
function emergencyWithdraw(address izlude) external {
    PoolInfo storage pool = poolInfo[izlude];
    UserInfo storage user = userInfo[izlude][msg.sender];

    uint256 jellopy = user.jellopy;
    user.jellopy = 0;
    user.rewardDebt = (user.storedJellopy * pool.accKSWPerJellopy) / 1e12;

    if (jellopy > 0) {
        IERC20 want = pool.want;
        uint256 wantBefore = want.balanceOf(address(this));
        IIzludeV2(izlude).withdraw(msg.sender, jellopy);
        uint256 wantAfter = want.balanceOf(address(this));
        want.safeTransfer(msg.sender, wantAfter - wantBefore);
    }
    emit EmergencyWithdraw(msg.sender, izlude, jellopy);
}
```

5.5. Improper Reward Calculation in PronteraV2

ID	IDX-005
Target	PronteraV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The $KSW reward miscalculation can lead to an unfair $KSW token distribution to the users, which may not comply with the tokenomics defined and announced to the users. This will result in loss of reputation for the platform and monetary loss for the users. **Likelihood:** Medium The `add()` and the `set()` functions can only be called by the contract owner, but it is possible that the `totalAllocPoint` state may be changed without setting the `_withUpdate` parameter to `true`.
Status	Resolved * KillSwitch team has confirmed that the team will always set the `withUpdate` parameter as `true` when calling the `add()` and the `set()` functions.

5.5.1. Description

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

In the `add()` and `set()` functions shown below, if `_withUpdate` is set to false, the `totalAllocPoint` variable will be modified without updating the rewards (`massUpdatePools()`).

PronteraV2.sol

```
function add(
    address izlude,
    uint64 allocPoint,
    bool withUpdate
) external onlyOwner {
    require(IIzludeV2(izlude).prontera() == address(this), "?");
    require(IIzludeV2(izlude).totalSupply() >= 0, "??");
    require(poolInfo[izlude].izlude == address(0), "duplicated");
    if (withUpdate) {
        massUpdatePools();
```

```
    }

    poolInfo[izlude] = PoolInfo({
        want: IIzludeV2(izlude).want(),
        izlude: izlude,
        allocPoint: allocPoint,
        lastRewardTime: uint64(block.timestamp),
        accKSWPerJellopy: 0
    });
    totalPool += 1;
    totalAllocPoint += allocPoint;
    if (allocPoint > 0) {
        _addTraversal(izlude);
    }
    emit AddPool(izlude, allocPoint, withUpdate);
}
```

PronteraV2.sol

```
function set(
    address izlude,
    uint64 allocPoint,
    bool withUpdate
) external onlyOwner {
    require(izlude != address(0), "invalid izlude");
    PoolInfo storage pool = poolInfo[izlude];
    require(pool.izlude == izlude, "!found");
    if (withUpdate) {
        massUpdatePools();
    }

    totalAllocPoint = (totalAllocPoint - pool.allocPoint) + allocPoint;
    pool.allocPoint = allocPoint;
    if (allocPoint > 0) {
        _addTraversal(izlude);
    }
    emit SetPool(izlude, allocPoint, withUpdate);
}
```

For example, assuming that on block time is 1000000, `setKSWPerSecond` is 5 $KSW per second, `totalAllocPoint` is 5000, and `allocPoint` of pool id 0 is 500.

Block time	Action
1000000	All pools' rewards are updated
1100000	A new pool is added using the add() function, causing the totalAllocPoint to be changed from 5000 to 10000
1200000	The pools' rewards are updated once again.

From current logic, the total rewards allocated to the pool id 0 during block time 1000000 to 1200000 is equal to 50,000 $KSW calculated using the following equation:

Block time	Total Block Reward	Total Allocation Point	Total $KSW per second for pool 0 (kswPerSecond*pool0allocPoint/totalAllocPoint)	Total pool 0 $KSW Reward
1000000 - 1200000	200000	10,000	0.25 $KSW per second	50,000 $KSW

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

Block time	Total Block Reward	Total Allocation Point	Total $KSW per second for pool 0 (kswPerSecond*pool0allocPoint/totalAllocPoint)	Total pool 0 $KSW Reward
1000000 - 1100000	100000	5,000	0.5 $KSW per second	50,000 $KSW
1100000 - 1200000	100000	10,000	0.25 $KSW per second	25,000 $KSW

The correct total $KSW reward is 75,000 $KSW, which is different from the miscalculated reward by 25,000 $KSW.

5.5.2. Remediation

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

PronteraV2.sol

```
function add(
    address izlude,
    uint64 allocPoint
) external onlyOwner {
    require(IIzludeV2(izlude).prontera() == address(this), "?");
    require(IIzludeV2(izlude).totalSupply() >= 0, "??");
    require(poolInfo[izlude].izlude == address(0), "duplicated");
    massUpdatePools();
    poolInfo[izlude] = PoolInfo({
        want: IIzludeV2(izlude).want(),
        izlude: izlude,
        allocPoint: allocPoint,
        lastRewardTime: uint64(block.timestamp),
        accKSWPerJellopy: 0
    });
    totalPool += 1;
    totalAllocPoint += allocPoint;
    if (allocPoint > 0) {
        _addTraversal(izlude);
    }
    emit AddPool(izlude, allocPoint);
}
```

PronteraV2.sol

```
function set(
    address izlude,
    uint64 allocPoint
) external onlyOwner {
    require(izlude != address(0), "invalid izlude");
    PoolInfo storage pool = poolInfo[izlude];
    require(pool.izlude == izlude, "!found");
    massUpdatePools();
    totalAllocPoint = (totalAllocPoint - pool.allocPoint) + allocPoint;
    pool.allocPoint = allocPoint;
    if (allocPoint > 0) {
        _addTraversal(izlude);
    }
    emit SetPool(izlude, allocPoint);
}
```

5.6. Improper Reward Calculation in Emperium

ID	IDX-006
Target	Emperium
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The reward of the pool that has the same staking token as the reward token will be slightly lower than what it should be, resulting in monetary loss for the users and loss of reputation for the platform. **Likelihood:** Medium It is likely that the pool with the same staking token as the reward token will be added by the contract owner.
Status	Resolved KillSwitch team has resolved this issue by storing the amount of the token staked as suggested in commit `c9b001b2597ba9b173d63f5b9d8e642cf35a709e`.

5.6.1. Description

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

Emperium.sol

```
function add(IERC20 token, uint64 allocPoint) external onlyOwner {
    require(!isTokenInPool[token], "duplicated");
    massUpdatePools();

    isTokenInPool[token] = true;
    totalAllocPoint += allocPoint;
    poolInfo.push(
        PoolInfo({token: token, allocPoint: allocPoint, lastRewardTime:
uint64(block.timestamp), accKSWPerShare: 0})
    );
    emit AddPool(poolInfo.length - 1, token, allocPoint);
}
```

When the `token` is the same as $KSW, the reward calculation for that pool in the `updatePool()` function can be incorrect.

This is because the current balance of the `token` in the contract is used in the calculation of the reward.

Since the `token` is the same as the reward, the reward that is transferred from reserve to the contract will inflate the value of `tokenSupply`, causing the reward of that pool to be less than what it should be.

Emperium.sol

```
function updatePool(uint256 pid) public {
    PoolInfo storage pool = poolInfo[pid];
    if (block.timestamp > pool.lastRewardTime) {
        uint256 tokenSupply = pool.token.balanceOf(address(this));
        if (tokenSupply > 0) {
            uint256 time = block.timestamp - pool.lastRewardTime;
            uint256 kswReward = (time * kswPerSecond * pool.allocPoint) /
totalAllocPoint;
            uint256 r = reserve.withdraw(address(this), kswReward);
            pool.accKSWPerShare += (r * 1e12) / tokenSupply;
        }
        pool.lastRewardTime = uint64(block.timestamp);
    }
}
```

5.6.2. Remediation

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

Emperium.sol

```
function add(IERC20 token, uint64 allocPoint) external onlyOwner {
    require(!isTokenInPool[token], "duplicated");
    require(address(isTokenInPool[token]) != ksw, "Is rewards token");
    massUpdatePools();

    isTokenInPool[token] = true;
    totalAllocPoint += allocPoint;
    poolInfo.push(
        PoolInfo({token: token, allocPoint: allocPoint, lastRewardTime:
uint64(block.timestamp), accKSWPerShare: 0})
    );
    emit AddPool(poolInfo.length - 1, token, allocPoint);
}
```

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, or store the amount of the token staked to use in the reward calculation, for example:

The `totalKSWDeposited` variable to store the total users' $KSW staked should be declared.

Emperium.sol

```
1 uint256 public totalKSWDeposited;
```

The `totalKSWDeposited` should be added with the `amount` value when the user deposits the $KSW in `deposit()` function (line 761), and when the user withdraws the $KSW token, subtract the `totalKSWDeposited` with `amount` value in the `withdraw()` function (line 781).

Emperium.sol

```
745 function deposit(uint256 pid, uint256 amount) external {
746     PoolInfo storage pool = poolInfo[pid];
747     UserInfo storage user = userInfo[pid][msg.sender];
748     updatePool(pid);
749     if (user.amount > 0) {
750         uint256 pending = ((user.amount * pool.accKSWPerShare) / 1e12) -
user.rewardDebt;
751         if (pending > 0) {
752             IERC20(ksw).transfer(msg.sender, pending);
753         }
754     }
755 
756     user.amount += amount;
757     user.rewardDebt = (user.amount * pool.accKSWPerShare) / 1e12;
758     if (amount > 0) {
759         require(_safeERC20TransferIn(pool.token, amount) == amount, "!amount");
760     }
761     if (pool.token == ksw) totalKSWDeposited = totalKSWDeposited + amount;
762     emit Deposit(msg.sender, pid, amount);
763 }
764 
765 function withdraw(uint256 pid, uint256 amount) external {
766     PoolInfo storage pool = poolInfo[pid];
767     UserInfo storage user = userInfo[pid][msg.sender];
768     require(user.amount >= amount, "withdraw: not good");
769 
770     updatePool(pid);
771     uint256 pending = ((user.amount * pool.accKSWPerShare) / 1e12) -
user.rewardDebt;
772     if (pending > 0) {
773         IERC20(ksw).transfer(msg.sender, pending);
774     }
775 
776     user.amount -= amount;
777     user.rewardDebt = (user.amount * pool.accKSWPerShare) / 1e12;
778     if (amount > 0) {
779         pool.token.safeTransfer(msg.sender, amount);
780     }
```

```
    if (pool.token == ksw) totalKSWDeposited = totalKSWDeposited - amount;
    emit Withdraw(msg.sender, pid, amount);
}
```

The `totalKSWDeposited` should be used to calculate the reward in the `pendingKSW()` function and `updatePool()` function when the `pool.token` is $KSW.

Emperium.sol

```
function pendingKSW(uint256 pid, address _user) external view returns (uint256)
{
    PoolInfo storage pool = poolInfo[pid];
    UserInfo storage user = userInfo[pid][_user];
    uint256 accKSWPerShare = pool.accKSWPerShare;
    uint256 tokenSupply;
    if (pool.token == ksw) {
      tokenSupply = totalKSWDeposited;
    } else {
      tokenSupply = pool.token.balanceOf(address(this));
    }


    if (block.timestamp > pool.lastRewardTime && tokenSupply != 0) {
        uint256 time = block.timestamp - pool.lastRewardTime;
        uint256 kswReward = (time * kswPerSecond * pool.allocPoint) /
totalAllocPoint;

        uint256 stakingBal = reserve.balances();
        accKSWPerShare += (Math.min(kswReward, stakingBal) * 1e12) /
tokenSupply;
    }

    uint256 r = ((user.amount * accKSWPerShare) / 1e12) - user.rewardDebt;
    return r;
}
```

Emperium.sol

```
function updatePool(uint256 pid) public {
    PoolInfo storage pool = poolInfo[pid];
    if (block.timestamp > pool.lastRewardTime) {
        uint256 tokenSupply;
        if (pool.token == ksw) {
          tokenSupply = totalKSWDeposited;
        } else {
          tokenSupply = pool.token.balanceOf(address(this));
        }

```

```
        if (tokenSupply > 0) {
            uint256 time = block.timestamp - pool.lastRewardTime;
            uint256 kswReward = (time * kswPerSecond * pool.allocPoint) / 
totalAllocPoint;
            uint256 r = reserve.withdraw(address(this), kswReward);
            pool.accKSWPerShare += (r * 1e12) / tokenSupply;
        }
        pool.lastRewardTime = uint64(block.timestamp);
    }
}
```

5.7. Transaction Ordering Dependence

ID	IDX-007
Target	PancakeByalanLP
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Low **Impact:** Medium The users and the platform will lose a portion of tokens from the front-running attack when compounding the reward. **Likelihood:** Low There is low profit for the attacker because the reserve in the $WBNB-$CAKE pool is high, and the harvest volume is very low compared to the amount in the pool, so there is a low motivation for the attack.
Status	**Acknowledged** KillSwitch team has acknowledged this issue. However, the reinvestment is performed every 6 hours. As a consequence, the risk is quite low since the amount of token reinvested is very small relative to the liquidity in the swap pool.

5.7.1. Description

The `harvest()` function will collect the rewards from the `MASTERCHEF` and consume these rewards through the `chargeFees()` function, the `addLiquidity()` function, and the `deposit()` function.

PancakeByalanLP.sol

```
function harvest() external override whenNotPaused onlyEOA onlyHarvester
gasThrottle {
    IMasterChef(MASTERCHEF).deposit(pid, 0);
    chargeFees();
    addLiquidity();
    deposit();

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

The rewards will be swapped into $WBNB and a pair of liquidity tokens for the liquidity adding.

During the swapping, the `IUniswapV2Router02(unirouter).swapExactTokensForETH()` function is executed in the `chargeFees()` and `addLiquidity()` functions by setting the `amountOutMin` as 0 as shown below:

PancakeByalanLP.sol

```
function chargeFees() internal nonReentrant {
    uint256 toBnb = (IERC20(CAKE).balanceOf(address(this)) * totalFee) /
MAX_FEE;
    IUniswapV2Router02(unirouter).swapExactTokensForETH(toBnb, 0,
cakeToWbnbRoute, address(this), block.timestamp);

    uint256 bnbBal = address(this).balance;

    uint256 callFeeAmount = (bnbBal * callFee) / feeSum;
    payable(msg.sender).sendValue(callFeeAmount);

    uint256 treasuryFeeAmount = (bnbBal * treasuryFee) / feeSum;
    payable(treasuryFeeRecipient).sendValue(treasuryFeeAmount);

    uint256 kswFeeAmount = (bnbBal * kswFee) / feeSum;
    payable(kswFeeRecipient).sendValue(kswFeeAmount);
}
```

PancakeByalanLP.sol

```
function addLiquidity() internal {
    uint256 cakeHalf = IERC20(CAKE).balanceOf(address(this)) / 2;

    if (lpToken0 != CAKE) {
        IUniswapV2Router02(unirouter).swapExactTokensForTokens(
            cakeHalf,
            0,
            cakeToLp0Route,
            address(this),
            block.timestamp
        );
    }

    if (lpToken1 != CAKE) {
        IUniswapV2Router02(unirouter).swapExactTokensForTokens(
            cakeHalf,
            0,
            cakeToLp1Route,
            address(this),
            block.timestamp
        );
    }
```

```
        IUniswapV2Router02(unirouter).addLiquidity(
            lpToken0,
            lpToken1,
            IERC20(lpToken0).balanceOf(address(this)),
            IERC20(lpToken1).balanceOf(address(this)),
            0,
            0,
            address(this),
            block.timestamp
        );
    }
```

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

For example, when the compounding is happening, the `swapExactTokensForETH()` function was executed with 0 `amountOutMin` to swap claimed reward ($CAKE) to $WBNB in the `chargeFee()` function, and swap a half remaining reward with 0 `amountOutMin` to `lpToken0` or `lpToken1` to make liquidity token for farming in the `addLiquidity()` function. The attacker can monitor `PancakeByalanLP` contract to wait for compounding transactions to happen, then submit swapping transactions with the same token pair with the higher gas price to make the attacker's transaction completed 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 - $CAKE	50 $WBNB	50 $CAKE

The platform swaps 5 $CAKE to $WBNB.

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

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

However, if this transaction is being front-run with the same input (5 $CAKE), 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 - $CAKE	45.43 $WBNB	55 $CAKE

The platform then swaps 5 $CAKE to $WBNB.

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

As a result, swapping 5 $CAKE 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.

5.7.2. 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.

5.8. Design Flaw in massUpdatePool() Function

ID	IDX-008
Target	Emperium
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `massUpdatePool()` function will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the `poolInfo` size will be raised until the `massUpdatePool()` is eventually unusable.
Status	Resolved * KillSwitch team has confirmed that the team will add only 1 pool in the `Emperium` contract. Therefore, it is very unlikely that the poolInfo size will be raised until the `massUpdatePool()` is eventually unusable.

5.8.1. Description

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

Emperium.sol

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

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

5.8.2. Remediation

Inspex suggests making the contract capable of removing unnecessary or ended pools to reduce the loop rounds in the `massUpdatePools()` function.

5.9. Liquidity Token Amount Miscalculation

ID	IDX-009
Target	PancakeByalanLP
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Low A small amount of token can be left in the contract, resulting in a lower amount of tokens used in the compounding. However, the leftover token will be used in the next execution of harvesting. **Likelihood:** Medium This issue only occurs when the `harvest()` function is executed by the address allowed in the `onlyHarvester` modifier.
Status	**Acknowledged** KillSwitch team has acknowledged this issue. However, the risk is quite low because some leftover tokens will be used in the next execution of harvesting.

5.9.1. Description

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

PancakeByalanLP.sol

```
function harvest() external override whenNotPaused onlyEOA onlyHarvester
gasThrottle {
    IMasterChef(MASTERCHEF).deposit(pid, 0);
    chargeFees();
    addLiquidity();
    deposit();

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

The `addLiquidity()` function calculates the tokens that is used for adding liquidity (`lpToken0` and `lpToken1`) amounts by dividing total reward tokens harvested from `MASTERCHEF` by 2 in line 1369 as shown below:

PancakeByalanLP.sol

```
function addLiquidity() internal {
    uint256 cakeHalf = IERC20(CAKE).balanceOf(address(this)) / 2;

    if (lpToken0 != CAKE) {
        IUniswapV2Router02(unirouter).swapExactTokensForTokens(
            cakeHalf,
            0,
            cakeToLp0Route,
            address(this),
            block.timestamp
        );
    }

    if (lpToken1 != CAKE) {
        IUniswapV2Router02(unirouter).swapExactTokensForTokens(
            cakeHalf,
            0,
            cakeToLp1Route,
            address(this),
            block.timestamp
        );
    }

    IUniswapV2Router02(unirouter).addLiquidity(
        lpToken0,
        lpToken1,
        IERC20(lpToken0).balanceOf(address(this)),
        IERC20(lpToken1).balanceOf(address(this)),
        0,
        0,
        address(this),
        block.timestamp
    );
}
```

The first portion of reward used to swap to `lpToken0`, and the second portion used to swap to `lpToken1`.

When the price impact of swapping from reward token to `lpToken0` and `lpToken1` is different, some amount of `lpToken0` or `lpToken1` will be left in the contract.

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

Pool	Reserve token 0	Reserve token 1	Ratio
$CAKE - $A	1,000 $CAKE	100 $A	10 $CAKE : 1 $A
$CAKE - $B	10,000 $CAKE	1,000 $B	10 $CAKE : 1 $B

- The harvested rewards are 200 $CAKE.

Refer to `x * y = k` formula,

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

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

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

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

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

5.9.2. Remediation

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

For example, implementing the `optimalDeposit()` function to calculate the exact needed token amounts in order to swap to the targeted tokens, which the value will be equal.

PancakeByalanLP.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;
}
```

Apply the `optimalDeposit()` function in the `addLiquidity()` function.

PancakeByalanLP.sol

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

    if (lpToken0 != CAKE && lpToken1 != CAKE) { // convert all to lp0
        IUniswapV2Router02(unirouter).swapExactTokensForTokens(
IERC20(CAKE).balanceOf(address(this)), 0, cakeToLp0Route, address(this),
block.timestamp);
        (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 == CAKE ? lpToken1 : lpToken0;
            (swapAmt, isReversed) = optimalDeposit(
                IERC20(CAKE).balanceOf(address(this)),
                IERC20(otherToken).balanceOf(address(this)),
                lpToken0Reserve,
                lpToken1Reserve
            );
            (path[0], path[1]) = isReversed ? (otherToken, WBNB, CAKE) : (CAKE, WBNB, otherToken);
        }
        IUniswapV2Router02(unirouter).swapExactTokensForTokens(swapAmt, 0, path, address(this), block.timestamp);

        IUniswapV2Router02(unirouter).addLiquidity(
            lpToken0,
            lpToken1,
            IERC20(lpToken0).balanceOf(address(this)),
            IERC20(lpToken1).balanceOf(address(this)),
            0,
            0,
            address(this),
            block.timestamp
        );
}
```

Note that the remediation of the other issue is not applied in this code.

5.10. Insufficient Logging for Privileged Function

ID	IDX-010
Target	IzludeV2 PancakeByalanLP PronteraReserve
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged function execution cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged function will be a malicious action.
Status	Resolved Some of the affected privileged functions still do not have any event emitted. However, KillSwitch team has clarified that that the `setIzlude()`, `setProntera()`, and `setEmperium()` functions can be called only once, and the `inCaseTokensGetStuck()` function is used when the unrelated tokens (not `want` token) are stuck in the contract. Therefore, there is no reason to monitor them.

5.10.1. Description

Privileged function that is executable by the controlling parties is not logged properly by emitting events. Without an event, 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 to the platform.

For example, the owner can set the address of `_prontera` which is a contract that can withdraw funds from the `PronteraReserve` contract, and no event will be emitted.

PronteraReserve.sol

```
function setProntera(address _prontera) external onlyOwner {
    require(prontera == address(0), "?");
    require(IReserveWithdrawer(_prontera).reserve() == address(this), "invalid prontera");

    prontera = _prontera;
}
```

The privileged functions without sufficient logging are as follows:

File	Contract	Function
IzludeV2.sol (L:853)	IzludeV2	inCaseTokensGetStuck()
PancakeByalanLP.sol (L:1155)	ByalanIsland	setIzlude()
PronteraReserve.sol (L:571)	PronteraReserve	setProntera()
PronteraReserve.sol (L:578)	PronteraReserve	setEmperium()

5.10.2. Remediation

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

PronteraReserve

```
571 event SetProntera(address prontera)
572 function setProntera(address _prontera) external onlyOwner {
573     require(prontera == address(0), "?");
574     require(IReserveWithdrawer(_prontera).reserve() == address(this), "invalid
prontera");
575 
576     prontera = _prontera;
577     emit SetProntera(_prontera);
578 }
```

5.11. Improper Function Visibility

ID	IDX-011
Target	Emperium
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved KillSwitch team has resolved this issue as suggested in commit `c9b001b2597ba9b173d63f5b9d8e642cf35a709e`

5.11.1. Description

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

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

Emperium.sol

```
function set(uint256 pid, uint64 allocPoint) public onlyOwner {
    massUpdatePools();

    totalAllocPoint = (totalAllocPoint - poolInfo[pid].allocPoint) +
allocPoint;
    poolInfo[pid].allocPoint = allocPoint;
    emit SetPool(pid, allocPoint);
}
```

5.11.2. Remediation

Inspex suggests changing the `set()` function's visibility to `external` if it is not called from any internal function as shown in the following example:

Emperium.sol

```
function set(uint256 pid, uint64 allocPoint) external onlyOwner {
    massUpdatePools();

    totalAllocPoint = (totalAllocPoint - poolInfo[pid].allocPoint) +
```

```
allocPoint;
    poolInfo[pid].allocPoint = allocPoint;
    emit SetPool(pid, allocPoint);
}
```

5.12. Inexplicit Solidity Compiler Version

ID	IDX-012
Target	PronteraV2 PronteraReserve PancakeByalanLP IzludeV2 GasPrice FeeKafra Emperium AllocKafra
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved KillSwitch team has resolved this issue as suggested in commit `986190679fff1b7b36c4197acd887ad1363e9e53`

5.12.1. 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.

PronteraV2.sol

```
pragma solidity ^0.8.0;
```

The following table contains all targets with the inexplicit compiler version:

Contract	Version
PronteraV2	^0.8.0
PronteraReserve	^0.8.0
PancakeByalanLP	^0.8.0
IzludeV2	^0.8.0
GasPrice	^0.8.0

FeeKafra	^0.8.0
Emperium	^0.8.0
AllocKafra	^0.8.0

5.12.2. Remediation

Inspex suggests fixing the solidity compiler to the latest stable version. At the time of the audit, the latest stable version of the Solidity compiler in major 0.8 is v0.8.9.

PronteraV2.sol

```
pragma solidity 0.8.9;
```

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