YES contracts

Smart Contract Audit Report
Prepared for Yuemmai Rental

Date Issued:	Mar 1, 2022
Project ID:	AUDIT2022005
Version:	v2.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022005
Version	v2.0
Client	Yuemmai Rental
Project	YES contracts
Auditor(s)	Suvicha Buakhom Darunphop Pengkumta
Author(s)	Darunphop Pengkumta
Reviewer	Suvicha Buakhom
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
2.0	Mar 1, 2022	Update executive summary	Darunphop Pengkumta
1.0	Feb 26, 2022	Full report	Darunphop Pengkumta

Contact Information

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

Table of Contents

1. Executive Summary

As requested by Yuemmai Rental, Inspex team conducted an audit to verify the security posture of the YES contracts smart contracts between Jan 24, 2022 and Feb 3, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of YES contracts 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.

Inspex has a concern about the project design. The use of $YES as a collateral asset on the platform could cause a problem if the token's liquidity is not large enough to withstand the price manipulation; a cascading liquidation could happen. With the Yuemmai Rental team's awareness, the Yuemmai platform has measures to cope with the issue by limiting the collateral factor when the platform is released in the early stage, then increasing the collateral factor for each phase of the project to correlate with the growth of $YES. Additionally, the users of the platform have to pass the KYC process. So does the borrowing limit, which has been set individually, so that the platform can limit the impact of the position being liquidated.

1.1. Audit Result

In the initial audit, Inspex found 1 critical, 2 high, 1 medium, 1 low, and 3 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved or mitigated in the reassessment. Therefore, Inspex trusts that YES contracts smart contracts have high-level protections in place to be safe from most attacks.

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

Yuemmai Rental, a decentralized digital asset leasing business developed by Yuemmai (Thailand) Co., Ltd., is a platform that incorporates centralized finance with decentralized finance (CeDeFi). The lessor can rent out digital assets without needing to verify their identities (KYC). Yuemmai (Thailand) Co., Ltd. regulates the rental service to manage the risks for lessors.

YES contracts are smart contracts providing digital assets borrowing functions for the users on Bitkub chain. YES contracts use $YES as a collateral asset for determining the users' borrowability. The users can deposit their assets for gaining interest from other users' borrowing. YES contracts also allow the users to liquidate the borrowing that the borrowed asset value has exceeded the collateral value to gain some incentive from the liquidation.

Scope Information:

Project Name	YES contracts
Website	https://ยืมมั้ย.digital
Smart Contract Type	Ethereum Smart Contract
Chain	Bitkub Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Jan 24, 2022 - Feb 3, 2022
Reassessment Date	Feb 14, 2022

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

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

2.2. Scope

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

Initial Audit: (Commit: de7684e0f771acefabe4d5f6f3648b81e75e83ff)

Contract	Location (URL)
JumpRateModel	https://github.com/inspex-archive/YuemmaiRental_YES-contracts/blob/de7684e0f7/contracts/JumpRateModel.sol
KAP20Lending	https://github.com/inspex-archive/YuemmaiRental_YES-contracts/blob/de7684e0f7/contracts/KAP20Lending.sol
KUBLending	https://github.com/inspex-archive/YuemmaiRental_YES-contracts/blob/de7684e0f7/contracts/KUBLending.sol
LToken	https://github.com/inspex-archive/YuemmaiRental_YES-contracts/blob/de7684e0f7/contracts/LToken.sol
MarketImpl	https://github.com/inspex-archive/YuemmaiRental_YES-contracts/blob/de7684e0f7/contracts/MarketImpl.sol
SlidingWindowOracle	https://github.com/inspex-archive/YuemmaiRental_YES-contracts/blob/de7684e0f7/contracts/SlidingWindowOracle.sol
YESController	https://github.com/inspex-archive/YuemmaiRental_YES-contracts/blob/de7684e0f7/contracts/YESController.sol
YESPriceOracle	https://github.com/inspex-archive/YuemmaiRental_YES-contracts/blob/de7684e0f7/contracts/YESPriceOracle.sol
YESToken	https://github.com/inspex-archive/YuemmaiRental_YES-contracts/blob/de7684e0f7/contracts/YESToken.sol
YESVault	https://github.com/inspex-archive/YuemmaiRental_YES-contracts/blob/de7684e0f7/contracts/YESVault.sol

Reassessment: (Commit: 69e87a886bddc9a2333a496cea48df75612be0dd)

Contract	Location (URL)
JumpRateModel	https://github.com/Finstable/yuemmai-contract/blob/69e87a886b/contracts/JumpRateModel.sol
KAP20Lending	https://github.com/Finstable/yuemmai-contract/blob/69e87a886b/contracts/KAP20Lending.sol
KUBLending	https://github.com/Finstable/yuemmai-contract/blob/69e87a886b/contracts/KUBLending.sol

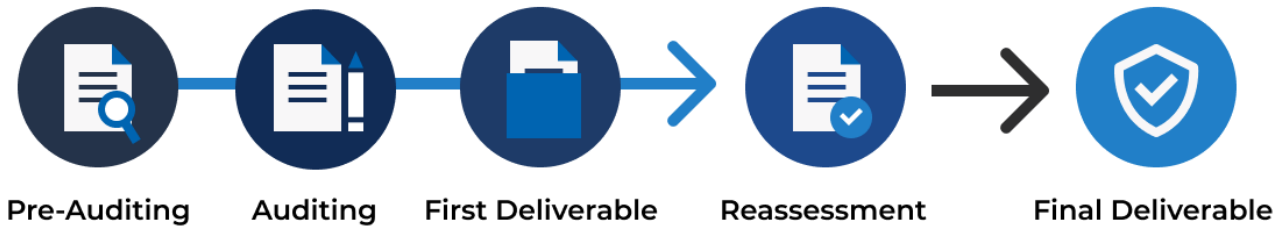
LToken	https://github.com/Finstable/yuemmai-contract/blob/69e87a886b/contracts/LToken.sol
MarketImpl	https://github.com/Finstable/yuemmai-contract/blob/69e87a886b/contracts/MarketImpl.sol
SlidingWindowOracle	https://github.com/Finstable/yuemmai-contract/blob/69e87a886b/contracts/SlidingWindowOracle.sol
YESController	https://github.com/Finstable/yuemmai-contract/blob/69e87a886b/contracts/YESController.sol
YESPriceOracle	https://github.com/Finstable/yuemmai-contract/blob/69e87a886b/contracts/YESPriceOracle.sol
YESToken	https://github.com/Finstable/yuemmai-contract/blob/69e87a886b/contracts/YESToken.sol
YESVault	https://github.com/Finstable/yuemmai-contract/blob/69e87a886b/contracts/YESVault.sol

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

3. Methodology

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

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



3.1. Test Categories

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

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

3.2. Audit Items

The 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
Use of Upgradable Contract Design
Centralized Control of State Variable
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication

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

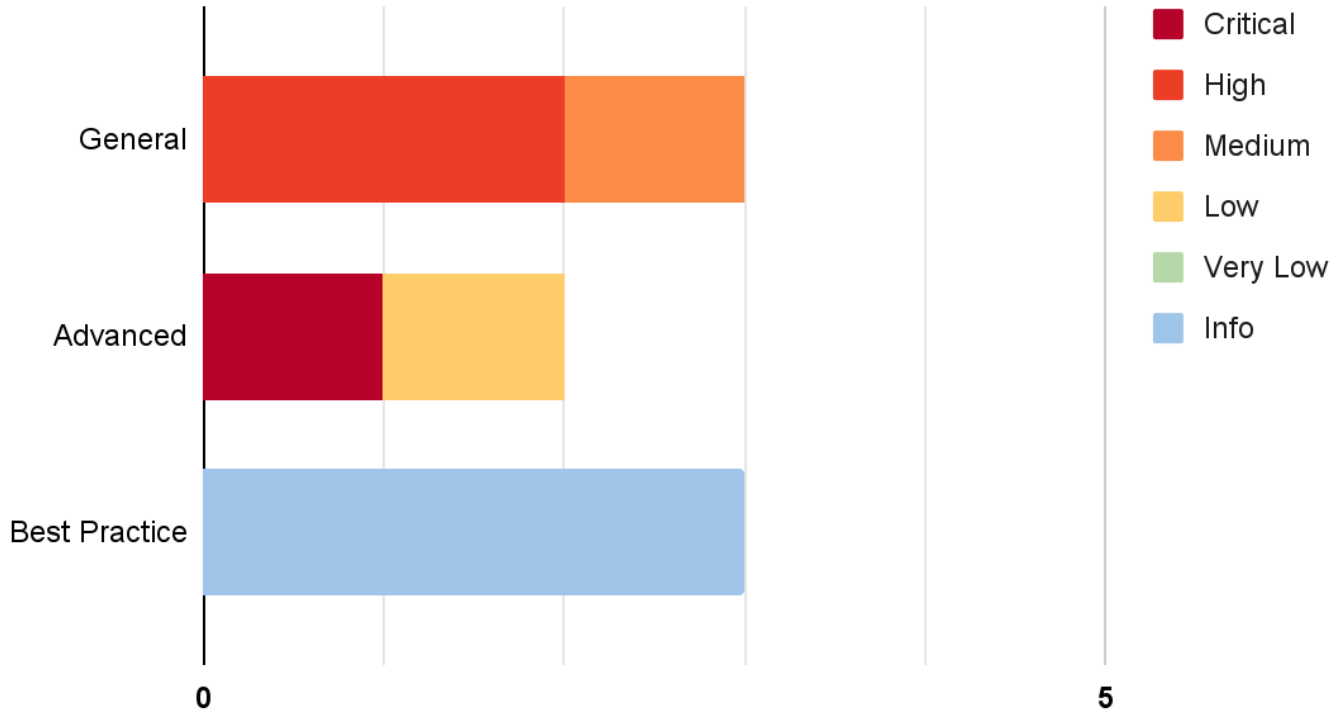
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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**.

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

4. Summary of Findings

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



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further 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	Withdrawable Amount Miscalculation	Advanced	Critical	Resolved
IDX-002	Centralized Control of State Variable	General	High	Resolved *
IDX-003	Invoking of Unreliable Smart Contract	General	High	Resolved *
IDX-004	Transaction Ordering Dependence	General	Medium	Resolved
IDX-005	Improper Parameter Calculation	Advanced	Low	Resolved
IDX-006	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved
IDX-007	Improper Function Visibility	Best Practice	Info	Resolved
IDX-008	Uninitialized State Variable	Best Practice	Info	Resolved

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

5. Detailed Findings Information

5.1. Withdrawable Amount Miscalculation

ID	IDX-001
Target	YESVault
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** Withdrawing $YES affects the collateral value of the withdrawing user. Users can withdraw $YES at the maximum amount to cause the collateral value of the withdrawing user dropping lower than their borrowed asset value and instantly becoming a bad debt, which causes the platform the loss of profit disencouraging investors from investing in the platform. **Likelihood: High** In `YesVault` contract, anyone can withdraw their $YES that they have deposited or have received from Airdrop. The token can be withdrawn until the remaining $YES equals the total value of the borrowed assets.
Status	**Resolved** This issue has already been resolved by the Yuemmai team in commit `69e87a886bddc9a2333a496cea48df75612be0dd` by changing the withdrawable token amount calculation to be the same as the recommended calculation as suggested.

5.1.1. Description

The `_withdraw()` function calculates the withdrawable amount by comparing the amount of $YES owned by the user with the users' borrowed amount. If the user has withdrawn at the maximum amount, the user will have $YES equal to the borrowed asset value; the collateral value will certainly be lower than the borrowed asset value, if the `collateralFactorMantissa` is less than `1e18`.

YESVault.sol

```
109 function _withdraw(uint256 amount, address sender) private {
110     (, , , uint256 borrowValue) = _controller.getAccountLiquidity(sender);
111     uint256 senderTokens = tokensOf(sender);
112
113     require(senderTokens >= borrowValue, "YESVault: ACCOUNT_SHORT_FALL");
114     require(
115         senderTokens - borrowValue >= amount,
116         "YESVault: NOT_ENOUGH_BALANCE"
```

```
117     );
118
119     uint256 senderAirdrop = _airdropOf[sender];
120
121     if (amount <= _airdropOf[sender]) {
122         _airdropOf[sender] -= amount;
123     } else {
124         _airdropOf[sender] = 0;
125         _depositOf[sender] -= (amount - senderAirdrop);
126     }
127
128     _totalAllocated -= amount;
129
130     _yesToken.transfer(sender, amount);
131
132     emit Withdraw(sender, amount);
133 }
```

Users can withdraw assets this way after they have borrowed assets, virtually causing the platform to allow the users to borrow at 100% of the collateral value. Resulting in any borrowing in the platform can become bad debts as soon as they start borrowing by the intention of the users.

5.1.2. Remediation

Inspex suggests using the check of the collateral value after being withdrawn to be the condition for controlling the maximum withdrawable amount.

To have the code calculate the maximum withdrawable amount correctly, we suggest modifying the code in line 110 to receive collateral value returning from the `_controller.getAccountLiquidity()` function and removing the code at line 111, because the `senderToken` variable is not needed for the upcoming calculation. Instead of checking for the current $YES with the borrowed value at line 112, the condition should check for the current collateral value with the borrowed value. The maximum withdrawable amount condition should be as the example at line 114.

YESVault.sol

```
110 (,uint256 collateralValue , , uint256 borrowValue) =
    _controller.getAccountLiquidity(sender);
111
112 require(collateralValue >= borrowValue, "YESVault: ACCOUNT_SHORT_FALL");
113 require(
114     divScalarByExpTruncate(collateralValue - borrowValue, Exp({mantissa:
    collateralFactorMantissa})) >= amount,
115     "YESVault: NOT_ENOUGH_BALANCE"
116 );
```

The formula for the calculation at line 114 is derived from this formula, `(tokenOf - amount) * collateralFactorMantissa >= borrowValue`. The left-hand side of the formula signifies the collateral value after $YES being withdrawn and the right-hand side signifies the borrowed asset value. Therefore, the whole formula denotes that the collateral value after $YES being withdrawn must be more than the borrowed asset value.

5.2. Centralized Control of State Variable

ID	IDX-002
Target	BorrowLimitOracle KAP20Lending KUBLending YESController YESPriceOracle YESVault
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The controlling authorities can change the critical state variables to gain additional profit. For example, the admin can call the `_setYESToken()` function to set the $YES address to a dummy token address, then deposit the dummy token that has been minted unlimitedly to the vault to have the `_depositOf` state increase tremendously. Then the admin uses the `_setYESToken()` function again to change the address back to $YES. The admin can withdraw $YES in the vault as much as the amount of the deposited dummy token. **Likelihood: Medium** There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner only.
Status	Resolved * The Yuemmai team partially resolved this issue by changing the role that can change the critical states of the contracts to be another role, `SuperAdmin`. This role will be controlled by the Time lock mechanism with a minimum 24 hours delay, which the Yuemmai team promises to enforce within the week of the deployment. The users need to check if the timelock implemented contract has already been deployed and integrated into the `SuperAdmin` contract. Moreover, the `setYESToken()` function and some unused functions have already been removed.

5.2.1. Description

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

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

The controllable privileged state update functions are as follows:

Target	Contract	Function	Modifier
BorrowLimitOracle.sol (L: 13)	BorrowLimitOracle	setBorrowLimit()	onlyAdmin
LendingSetter.sol (L: 8)	KAP20Lending, KUBLending	_setController()	onlyAdmin
LendingSetter.sol (L: 24)	KAP20Lending, KUBLending	_setInterestRateModel()	onlyAdmin
LendingSetter.sol (L: 68)	KAP20Lending, KUBLending	_setPlatformReserveFactor()	onlyAdmin
LendingSetter.sol (L: 120)	KAP20Lending, KUBLending	_setPoolReserveFactor()	onlyAdmin
LendingSetter.sol (L: 171)	KAP20Lending, KUBLending	_setPlatformReserveExecutionPoint()	onlyAdmin
LendingSetter.sol (L: 211)	KAP20Lending, KUBLending	_setPoolReserveExecutionPoint()	onlyAdmin
LendingSetter.sol (L: 252)	KAP20Lending, KUBLending	_setBeneficiary()	onlyAdmin
LendingSetter.sol (L: 290)	KAP20Lending, KUBLending	_setSlippageTolerrance()	onlyAdmin
YESController.sol (L: 472)	YESController	_supportMarket()	onlyAdmin
YESController.sol (L: 504)	YESController	_setPriceOracle()	onlyAdmin
YESController.sol (L: 517)	YESController	_setYESVault()	onlyAdmin
YESController.sol (L: 524)	YESController	_setCollateralFactor()	onlyAdmin
YESController.sol (L: 543)	YESController	_setLiquidationIncentive()	onlyAdmin
YESPriceOracle.sol (L: 27)	YESPriceOracle	_addStableCoin()	onlyOwner
YESPriceOracle.sol (L: 33)	YESPriceOracle	_removeStableCoin()	onlyOwner
YESVault.sol (L: 206)	YESVault	_setYESToken()	onlyAdmin
YESVault.sol (L: 212)	YESVault	_setMarketImpl()	onlyAdmin
YESVault.sol (L: 218)	YESVault	_setMarket()	onlyAdmin
YESVault.sol (L: 224)	YESVault	_setSlippageTolerrance()	onlyAdmin
YESVault.sol (L: 234)	YESVault	activateOnlyKycAddress()	onlyAdmin

YESVault.sol (L: 238)	YESVault	setKYC()	onlyAdmin
YESVault.sol (L: 242)	YESVault	setAcceptedKycLevel()	onlyAdmin

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 community-run governance to control the use of these functions
- Using a time lock mechanism to delay the changes for a reasonable amount of time

For the `_setMintPaused()` function of the `YesController` contract that is used in only emergency cases, if the time lock mechanism is implemented to the `onlyAdmin` role, the `_setMintPaused()` function will be delayed. Inspex suggests creating a new role to avoid the delay, moreover, for good practice on the function naming, a public or an external function shouldn't be named with _ as a prefix, for example:

YesController.sol

```
35  address public pauseGuardian;
36
37  modifier onlyPauseGuardian() {
38      require(msg.sender == pauseGuardian, "Restricted only pause guardian");
39      _;
40  }
41
42  constructor(address adminRouter_, address borrowLimitOracle_, address
    pauseGuardian_) Authorization(adminRouter_) {
43      _borrowLimitOracle = IBorrowLimitOracle(borrowLimitOracle_);
44      pauseGuardian = pauseGuardian_;
45  }
```

YesController.sol

```
496 function setMintPaused(address lContractAddress, bool state) public
    onlyPauseGuardian returns (bool) {
497     require(_markets[lContractAddress].isListed, "cannot pause a market that is
    not listed");
498
499     mintGuardianPaused[lContractAddress] = state;
500     emit ActionPaused(lContractAddress, "Mint", state);
501 }
```

5.3. Invoking of Unreliable Smart Contract

ID	IDX-003
Target	KAP20Lending KUBLending LToken YESController YESToken YESVault
Category	General Smart Contract Vulnerability
CWE	CWE-829: Inclusion of Functionality from Untrusted Control Sphere
Risk	**Severity: High** **Impact: High** Using the `adminRouter` and `committee` contracts may cause harm to the user, e.g., arbitrarily depositing and withdrawing user funds, or setting user address to blacklist to make it non transferable. This results in potential monetary loss for the users and reputation damage to the platform. **Likelihood: Medium** The contract address of unreliable contracts which are controlled by a third party can be set by the contract owner only. It is possible that a third party who owns an unreliable contract may perform something malicious, intentionally or unintentionally.
Status	Resolved * The `committee`, `adminRouter`, and `kyc` contracts have been claimed to be possessed by `Bitkub`. `Bitkub` has published a document[3] to clarify the procedures of the contracts. Users are advised to understand the procedures of the contracts from the attached document before using them. The `Yuemmai` team has used the document from `Bitkub` to clarify the issue as follow: - The `kyc` contract The contract is claimed to be mainly used for verifying the user who interacts with the `Yuemmai` platform via `BitkubNext`. The users do not hold the private key of the wallet; `Bitkub` holds it instead and performs the transaction according to the user's instruction. - The `committee` contract This contract is used for approving all super admin functions in the `Bitkub chain`. The procedures are claimed to have a voting system for each committee-proposed proposal, i.e., the executions from the committee have to pass the voting before being executed. To have the proposal passed, the proposal has to have approval from more than half of the committees in which `Bitkub` does not disclose the exact number of the committees nor the identity of each

	committee. - The `adminRouter` contract The contract is claimed to be used for the `KAP20` token transferring on behalf of the user of `BitkubNext`. Without using this contract, the users cannot use `BitkubNext`'s wallet on this project. The `KAP20` token has already been implemented to prevent `Bitkub's admin` accounts from transferring the token of the user who does not use `BitkubNext`. However, `Bitkub` does not claim that these contracts have had a security control audited nor published a security audited report.

5.3.1. Description

The `committee`, `adminRouter`, and `kyc` contracts that are calling in the Yuemmai Rental platform are designed to be controlled by a third party. Designing it to be controlled by a third party may present unexpected dangers.

For example, in the `KAP20` contract that is inherited by the `LToken` and `YESToken` contracts, the `addBlacklist()` function is used to assign any address to blacklist by the `committee` role. The blacklisted address will be non transferable.

LToken.sol

```
194 function addBlacklist(address account) external override onlyCommittee {
195     _addBlacklist(account);
196 }
```

The following contracts contain unreliable smart contracts calling.

Target Contract	Unreliable Contract
KAP20Lending	committee, adminRouter, kyc
KUBLending	committee, adminRouter, kyc
LToken	committee, adminRouter, kyc
YESController	adminRouter
YESToken	committee, adminRouter, kyc
YESVault	adminRouter, kyc

5.3.2. Remediation

Inspex suggests terminating the use of unreliable smart contracts or redesigning by removing malicious functions, such as functions that can control other people's tokens.

5.4. Transaction Ordering Dependence

ID	IDX-004
Target	MarketImpl
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Medium **Impact:** Medium The price support (from pool reserve) can have a lower effect than it should be, i.e., when the platform will buy $YES back from the market, the platform has to pay more to get the $YES token. The mechanism for protecting from buying the higher price does not work properly. Also the amount of underlying that retrieves from seizing the collateral asset could be lower than it should be for the same reason. **Likelihood:** Medium The swap amount can be enough to meet the needs of the front-runner because the swap amount is a cumulative amount leading to motivation for the front-runner.
Status	Resolved This issue has already been resolved by the Yuemmai team in commit `69e87a886bddc9a2333a496cea48df75612be0dd` by changing the functions in `MarketImpl` contract to use the actual value from the liquidator instead of calculating the amount within the contract. Also, the higher level function of the issued function has a mechanic to have the liquidator take responsibility for a bad debt to demotivate them from leverage in the front-running in the liquidation process. The liquidator also can specify the minimum desired reward to prevent the transaction being impacted from the front-running. The name of the function has also changed to `swapExactInput()` for `merketSell()` and `swapExactOutput()` for `marketBuy()`.

5.4.1. Description

The `merketSell()` and the `marketBuy()` functions in the `MarketImpl` contract are used as a swapping route of another contract of the Yuemmai Rental platform. However, the `amounts` that is calculated in these functions is calculated by using the quote price by calling the `router.getAmountsOut()` function in the same transaction as the swapping transaction in lines 31 and 81.

MarketImpl.sol

```
10 function merketSell(
11     address market,
12     address srcToken,
13     address destToken,
```

```
14      uint256 amountIn,
15      address payable beneficiary,
16      uint256 slippageTolerrance
17  ) external override returns (uint256) {
18      uint256 INTERVAL = 5 * 60 * 60;
19      uint256 deadline = block.timestamp + INTERVAL;
20      uint256[] memory amounts;
21      address[] memory path = new address[](2);
22      uint256 amountOutMin;
23
24      IUniswapRouter02 router = IUniswapRouter02(market);
25      path[0] = srcToken;
26
27      IKAP20(srcToken).approve(address(router), amountIn * 10);
28
29      if (destToken == address(0)) {
30          path[1] = router.WETH();
31          amounts = router.getAmountsOut(amountIn, path);
32          amountOutMin =
33              (amounts[amounts.length - 1] * (1e18 - slippageTolerrance)) /
34              1e18;
35          amounts = router.swapExactTokensForETH(
36              amountIn,
37              amountOutMin,
38              path,
39              beneficiary,
40              deadline
41          );
42      } else {
43          path[1] = destToken;
44          amounts = router.getAmountsOut(amountIn, path);
45          amountOutMin =
46              (amounts[amounts.length - 1] * (1e18 - slippageTolerrance)) /
47              1e18;
48          amounts = router.swapExactTokensForTokens(
49              amountIn,
50              amountOutMin,
51              path,
52              beneficiary,
53              deadline
54          );
55      }
56
57      return amounts[amounts.length - 1];
58  }
59
60  function marketBuy(
```

```
61      address market,
62      address srcToken,
63      address destToken,
64      uint256 amountIn,
65      address payable beneficiary,
66      uint256 slippageTolerrance
67  ) external payable override returns (uint256) {
68      uint256 INTERVAL = 5 * 60 * 60;
69      uint256 deadline = block.timestamp + INTERVAL;
70
71      uint256[] memory amounts;
72      address[] memory path = new address[](2);
73      uint256 amountOutMin;
74
75      IUniswapRouter02 router = IUniswapRouter02(market);
76
77      if (srcToken == address(0)) {
78          path[0] = router.WETH();
79          path[1] = destToken;
80
81          amounts = router.getAmountsOut(amountIn, path);
82          amountOutMin = (amounts[amounts.length - 1] * (1e18 -
    slippageTolerrance)) / 1e18;
83          amounts = router.swapExactETHForTokens{value: amountIn}(
84              amountOutMin,
85              path,
86              beneficiary,
87              deadline
88          );
89      } else {
90          IKAP20(srcToken).approve(address(router), amountIn * 10);
91          path[0] = srcToken;
92          path[1] = destToken;
93          amounts = router.getAmountsOut(amountIn, path);
94          amountOutMin = (amounts[amounts.length - 1] * (1e18 -
    slippageTolerrance)) / 1e18;
95          amounts = router.swapExactTokensForTokens(
96              amountIn,
97              amountOutMin,
98              path,
99              beneficiary,
100             deadline
101         );
102     }
103
104     return amounts[amounts.length - 1];
105 }
```

The `merketSell()` function is used in the liquidation process to sell $YES (seized token) to the `underlying` token (borrowed token).

The `marketBuy()` function is used in the repay borrow process to buy $YES back to the vault when the `poolReserve`, which is the cumulation of the lending interest, has reached the `poolReserveExecutionPoint` value.

So, if the front-run happens, the amount out will be calculated after the front-run has happened, which the calculation will never takes effect of `slippageTolerrance` parameter. Resulting in the calculated amount out will not be protected from the front-running impact. The bought and sold tokens will be less than it should be. The impact also includes the `deadline` variable that uses `block.timestamp`, which `deadline` variable will never be reached when the transaction is being executed.

5.4.2. Remediation

Inspex suggests solutions for each functions:

Function 1: merketSell()

Firstly, the contract should have the functions that are using the `merketSell()` function and also the functions above that input `priceTolerance` from the liquidator.

Secondly, the liquidation process should be changed from selling $YES from the vault directly to having the liquidator paying the difference between the borrower's $YES and the borrowed value, in case of bad debt. If the attacker also pays for the bad debt, the benefit from the front-running will be deducted with the bad debt payment. This will discourage users from front-running the `merketSell()` function.

Function 2: marketBuy()

Inspex suggests reducing the `poolReserveExecutionPoint` value to a very low value. The amount for buying the token should be a low value to reduce the impact from the front-running. This can be done by lowering the value of `poolReserveExecutionPoint` state to be relatively low compared with the pool size. If the amount for buying $YES is low, the impact from the price will also be lower.

5.5. Improper Parameter Calculation

ID	IDX-005
Target	SlidingWindowOracle
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The price from the `consult()` function can be easily affected when the conditions are met. This price has been used in the calculation of the liquidity, so manipulating the price also affects other functions of the platform. **Likelihood:** Low It requires a great amount of the asset to buy or sell the target tokens and it also requires sequential execution of functions within a short amount of time. The execution also requires it to be executed at the exact time that only happens a few times within a day depending on the value of `windowSize` and `granularity`.
Status	Resolved This issue has already been resolved by the Yuemmai team in commit `69e87a886bddc9a2333a496cea48df75612be0dd` as suggested.

5.5.1. Description

The `SlidingWindowOracle` contract is an oracle that stores the price of the tokens and is used to get an estimated price of the tokens. Basically, the price `SlidingWindowOracle` contract gets the price by using the price's moving average over the time.

There are three states that can only be set on the constructor of the contract and affect the calculation of the contract:

1. `windowSize`, it is the size of the window used to calculate the average value.
2. `granularity`, it is the number of sub-window or the observation point of `windowSize` state. It also must be divisible to the value of `windowSize` state.
3. `periodSize`, it is the ratio between `windowSize` and `granularity` state, which is the size of each sub-window.

SlidingWindowOracle.sol

```
37 constructor(address factory_, uint windowSize_, uint8 granularity_) public {
38     require(granularity_ > 1, 'SlidingWindowOracle: GRANULARITY');
39     require(
40         (periodSize = windowSize_ / granularity_) * granularity_ ==
```

```
    windowSize_,
41          'SlidingWindowOracle: WINDOW_NOT_EVENLY_DIVISIBLE'
42      );
43      factory = factory_;
44      windowSize = windowSize_;
45      granularity = granularity_;
46  }
```

When the time has passed by the value of `periodSize` state and `update()` function is called, the cumulative price of the pair of the tokens and the current time will be stored in `_pairObservations[pair]` array, as shown at line from 130 to 132. `_pairObservations` state is a mapping of each pair of tokens with an array of observation points.

SlidingWindowOracle.sol

```
74  function update(address tokenA, address tokenB) external override {
75      address pair = IUniswapV2Factory(factory).getPair(tokenA, tokenB);
76
77      // populate the array with empty observations (first call only)
78      for (uint i = _pairObservations[pair].length; i < granularity; i++) {
79          _pairObservations[pair].push();
80      }
81
82      // get the observation for the current period
83      uint8 observationIndex = observationIndexOf(block.timestamp);
84      Observation storage observation =
    _pairObservations[pair][observationIndex];
85
86      // we only want to commit updates once per period (i.e. windowSize /
    granularity)
87      uint timeElapsed = block.timestamp - observation.timestamp;
88
89      if (timeElapsed > periodSize) {
90          (uint price0Cumulative, uint price1Cumulative,) =
    UniswapV2OracleLibrary.currentCumulativePrices(pair);
91          observation.timestamp = block.timestamp;
92          observation.price0Cumulative = price0Cumulative;
93          observation.price1Cumulative = price1Cumulative;
94      }
95  }
```

To get the price from the contract, the users must call the `consult()` function. The function will get an observation point from the `getFirstObservationInWindow()` function to be the first reference point for the upcoming calculation. The second reference point for the calculation is at the moment when this function is called. When the function has prepared two reference points, it will calculate the price through the `computeAmountOut()` function at line 126 and 128.

SlidingWindowOracle.sol

```
113 function consult(address tokenIn, uint amountIn, address tokenOut) external
    view override returns (uint amountOut) {
114     address pair = IUniswapV2Factory(factory).getPair(tokenIn, tokenOut);
115     Observation storage firstObservation = getFirstObservationInWindow(pair);
116
117     uint timeElapsed = block.timestamp - firstObservation.timestamp;
118     require(timeElapsed <= windowSize, 'SlidingWindowOracle:
    MISSING_HISTORICAL_OBSERVATION');
119     // should never happen.
120     require(timeElapsed >= windowSize - periodSize * 2, 'SlidingWindowOracle:
    UNEXPECTED_TIME_ELAPSED');
121
122     (uint price0Cumulative, uint price1Cumulative,) =
    UniswapV2OracleLibrary.currentCumulativePrices(pair);
123     (address token0,) = sortTokens(tokenIn, tokenOut);
124
125     if (token0 == tokenIn) {
126         return computeAmountOut(firstObservation.price0Cumulative,
    price0Cumulative, timeElapsed, amountIn);
127     } else {
128         return computeAmountOut(firstObservation.price1Cumulative,
    price1Cumulative, timeElapsed, amountIn);
129     }
130 }
```

This function calculates the price of the pair of tokens by differentiating the cumulative price between two reference points. Then, it will be divided with a timeElapsed parameter. The parameter acts as a damper in the window sliding scheme to prevent a price from being abruptly fluctuated. If the price has been raised in a short time but the value of `timeElapsed` is relatively big enough, the price will not be affected much. This is where the problem might arise, if the value of `timeElapsed` is relatively small.

SlidingWindowOracle.sol

```
 99 function computeAmountOut(
100     uint priceCumulativeStart, uint priceCumulativeEnd,
101     uint timeElapsed, uint amountIn
102 ) private pure returns (uint amountOut) {
103     // overflow is desired.
104     FixedPoint.uq112x112 memory priceAverage = FixedPoint.uq112x112(
105         uint224((priceCumulativeEnd - priceCumulativeStart) / timeElapsed)
106     );
107     amountOut = priceAverage.mul(amountIn).decode144();
108 }
```

The value of `timeElapsed` should be more than the value of `windowSize - periodSize * 2`. If the value of `granularity` is 2, the condition at line 120 will accept any value. The value of `timeElapsed` can be a very

small amount and can cause the price fluctuation. The value of `timeElapsed` is directly affected by the `getFirstObservationInWindow()` function.

SlidingWindowOracle.sol

```
113 function consult(address tokenIn, uint amountIn, address tokenOut) external
    view override returns (uint amountOut) {
114     address pair = IUniswapV2Factory(factory).getPair(tokenIn, tokenOut);
115     Observation storage firstObservation = getFirstObservationInWindow(pair);
116
117     uint timeElapsed = block.timestamp - firstObservation.timestamp;
118     require(timeElapsed <= windowSize, 'SlidingWindowOracle:
    MISSING_HISTORICAL_OBSERVATION');
119     // should never happen.
120     require(timeElapsed >= windowSize - periodSize * 2, 'SlidingWindowOracle:
    UNEXPECTED_TIME_ELAPSED');
121
122     (uint price0Cumulative, uint price1Cumulative,) =
    UniswapV2OracleLibrary.currentCumulativePrices(pair);
123     (address token0,) = sortTokens(tokenIn, tokenOut);
124
125     if (token0 == tokenIn) {
126         return computeAmountOut(firstObservation.price0Cumulative,
    price0Cumulative, timeElapsed, amountIn);
127     } else {
128         return computeAmountOut(firstObservation.price1Cumulative,
    price1Cumulative, timeElapsed, amountIn);
129     }
130 }
```

The function, `getFirstObservationInWindow()`, gets **the latest** of observation points that have been set in `_pairObservations[pair]` array. At line 66, it gets the index of `_pairObservations[pair]` array at the current timestamp, which is the latest index of `_pairObservations[pair]` array. The array itself has a size equal to the value `granularity` state. So, the array was implemented to be a circular array by the help from the `observationIndexOf()` function that is used to get an index of the observation point at each timestamp.

SlidingWindowOracle.sol

```
59 function observationIndexOf(uint timestamp) public view override returns (uint8
   index) {
60     uint epochPeriod = timestamp / periodSize;
61     return uint8(epochPeriod % granularity);
62 }
63
64 // returns the observation from the oldest epoch (at the beginning of the
   window) relative to the current time
```

```
65  function getFirstObservationInWindow(address pair) private view returns
    (Observation storage firstObservation) {
66      uint8 observationIndex = observationIndexOf(block.timestamp);
67      // no overflow issue. if observationIndex overflows, result is still zero.
68      uint8 firstObservationIndex = observationIndex % granularity;
69      firstObservation = _pairObservations[pair][firstObservationIndex];
70  }
```

From the calling of `getFirstObservationInWindow()` function at line 115 in the `consult()` function, the value of `timeElapsed` parameter will be a very small amount, if the value of `granularity` is `2` and `consult()` function has been called exactly after the `update()` function is called; both functions also have to be executed exactly after the current timestamp has started a new period.

SlidingWindowOracle.sol

```
113 function consult(address tokenIn, uint amountIn, address tokenOut) external
    view override returns (uint amountOut) {
114     address pair = IUniswapV2Factory(factory).getPair(tokenIn, tokenOut);
115     Observation storage firstObservation = getFirstObservationInWindow(pair);
116
117     uint timeElapsed = block.timestamp - firstObservation.timestamp;
118     require(timeElapsed <= windowSize, 'SlidingWindowOracle:
    MISSING_HISTORICAL_OBSERVATION');
119     // should never happen.
120     require(timeElapsed >= windowSize - periodSize * 2, 'SlidingWindowOracle:
    UNEXPECTED_TIME_ELAPSED');
121
122     (uint price0Cumulative, uint price1Cumulative,) =
    UniswapV2OracleLibrary.currentCumulativePrices(pair);
123     (address token0,) = sortTokens(tokenIn, tokenOut);
124
125     if (token0 == tokenIn) {
126         return computeAmountOut(firstObservation.price0Cumulative,
    price0Cumulative, timeElapsed, amountIn);
127     } else {
128         return computeAmountOut(firstObservation.price1Cumulative,
    price1Cumulative, timeElapsed, amountIn);
129     }
130 }
```

When users buy or sell the tokens and achieve the condition that causes `timeElapsed` parameter to be a small value, the price will moderately be affected depending on the amount that the users have bought or sold and the passed time of `timeElapsed` parameter.

5.5.2. Remediation

Inspex suggests adding `1` to the `observationIndex` variable in line 68 to have the function get the oldest observation point instead. If the oldest observation point is used as the first reference point, the value of `timeElapsed` will be guaranteed to be at least `windowSize - periodSize`.

Alternatively, by changing `granularity` to be more than `2` can also prevent the condition of `windowSize - periodSize * 2` being any value.

SlidingWindowOracle.sol

```
65  function getFirstObservationInWindow(address pair) private view returns
    (Observation storage firstObservation) {
66      uint8 observationIndex = observationIndexOf(block.timestamp);
67      // no overflow issue. if observationIndex overflows, result is still zero.
68      uint8 firstObservationIndex = (observationIndex + 1) % granularity;
69      firstObservation = _pairObservations[pair][firstObservationIndex];
70  }
```

Please be noted that the size of the `windowSize` state should be correlated with the liquidity pool of each asset. The current size of the liquidity pool of each asset can be considered a small pool. Therefore, the price of $YES has the potential to be manipulated and arbitraged. The use of `Sliding Window Oracle` could reduce the impact of price manipulation. For the suitable value of the `windowSize` state, it should be determined from using the statistical model. In a general rule of thumb, a smaller liquidity pool needs a bigger `windowSize` state.

5.6. Inexplicit Solidity Compiler Version

ID	IDX-006
Target	BorrowLimitOracle JumpRateModel KAP20Lending KUBLending LToken MarketImpl SlidingWindowOracle YESController YESPriceOracle YESToken YESVault
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved This issue has already been resolved by the Yuemmai team in commit `69e87a886bddc9a2333a496cea48df75612be0dd` as suggested.

5.6.1. 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.

KAP20Lending.sol

```
1 //SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
```

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

Contract	Version
BorrowLimitOracle	^0.8.0
JumpRateModel	^0.8.0
KAP20Lending	^0.8.0

KUBLending	^0.8.0
LToken	^0.8.0
MarketImpl	^0.8.0
SlidingWindowOracle	^0.8.0
YESController	^0.8.0
YESPriceOracle	^0.8.0
YESToken	^0.8.0
YESVault	^0.8.0

5.6.2. 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 0.8 is v0.8.11[2].

KAP20Lending.sol

```
1 //SPDX-License-Identifier: MIT
2 pragma solidity 0.8.11;
```

5.7. Improper Function Visibility

ID	IDX-007
Target	YESController YESPriceOrcale YESVault
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved This issue has already been resolved by the Yuemmai team in commit `69e87a886bddc9a2333a496cea48df75612be0dd` by either changing the suggested function visibility to `external` or removing some functions.

5.7.1. Description

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

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

YesController.sol

```
50 function enterMarkets(address[] memory lContracts) public override returns
   (uint[] memory) {
51     uint len = lContracts.length;
52
53     uint[] memory results = new uint[](len);
54     for (uint i = 0; i < len; i++) {
55         results[i] = uint(addToMarketInternal(lContracts[i], msg.sender));
56     }
57
58     return results;
59 }
```

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

Target	Function()
YESController (L: 50)	enterMarkets()
YESController (L: 496)	_setMintPaused()
YESController (L: 504)	_setPriceOracle()
YESController (L: 517)	_setYESVault()
YESPriceOrcale.sol (L: 27)	_addStableCoin()
YESPriceOrcale.sol (L: 33)	_removeStableCoin()
YESVault.sol (L: 234)	activateOnlyKycAddress()
YESVault.sol (L: 236)	setKYC()
YESVault.sol (L: 242)	setAcceptedKycLevel()
YESVault.sol (L: 252)	depositOf()
YESVault.sol (L: 256)	airdropOf()
YESVault.sol (L: 285)	slippageTolerrance()
YESVault.sol (L: 289)	totalAllocated()

5.7.2. Remediation

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

YesController.sol

```
50  function enterMarkets(address[] memory lContracts) external override returns
    (uint[] memory) {
51      uint len = lContracts.length;
52
53      uint[] memory results = new uint[](len);
54      for (uint i = 0; i < len; i++) {
55          results[i] = uint(addToMarketInternal(lContracts[i], msg.sender));
56      }
57
58      return results;
59  }
```

5.8. Uninitialized State Variable

ID	IDX-008
Target	YESController
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved This issue has already been resolved by the Yuemmai team in commit `69e87a886bddc9a2333a496cea48df75612be0dd` by removing the unused states and adding the setter function for `depositGuardianPaused`, `seizeGuardianPaused` and `borrowGuardianPaused`.

5.8.1. Description

The `transferGuardianPaused`, the `seizeGuardianPaused`, the `borrowGuardianPaused`, and the `borrowCaps` variables are never assigned value and should be removed for reducing gas used during deployment.

YESController.sol

```
28 bool public override transferGuardianPaused;
29 bool public override seizeGuardianPaused;
30
31 mapping(address => bool) public override mintGuardianPaused;
32 mapping(address => bool) public override borrowGuardianPaused;
33 mapping(address => uint) public override borrowCaps;
```

The default value of boolean is `false`, and the default value of uint is 0. When the unassigned value is used in the condition check, the value will be immutable. The conditions that use the unassigned value should be removed too.

The `transferGuardianPaused` is used in the `transferAllowed()` function in line 188.

YESController.sol

```
185 function transferAllowed(address lContract, address src) external override view
    returns (uint) {
186
187     // Pausing is a very serious situation - we revert to sound the alarms
```

```
188     require(!transferGuardianPaused, "transfer is paused");
189
190     // Currently the only consideration is whether or not
191     //  the src is allowed to withdraw this many tokens
192     uint allowed = withdrawAllowedInternal(lContract, src);
193     if (allowed != uint(Error.NO_ERROR)) {
194         return allowed;
195     }
196
197     return uint(Error.NO_ERROR);
198 }
```

The `seizeGuardianPaused` variable is used in the `seizeAllowed()` function in line 291.

YESController.sol

```
287 function seizeAllowed(
288     address lContract
289 ) external override view returns (uint) {
290     // Pausing is a very serious situation - we revert to sound the alarms
291     require(!seizeGuardianPaused, "seize is paused");
292
293     if (!_markets[lContract].isListed) {
294         return uint(Error.MARKET_NOT_LISTED);
295     }
296
297     if (address(ILending(lContract).controller()) != address(this)) {
298         return uint(Error.CONTROLLER_MISMATCH);
299     }
300
301     return uint(Error.NO_ERROR);
302 }
```

The `borrowGuardianPaused` variable is used in the `borrowAllowed()` and the `isDeprecated()` functions in lines 202 and 461.

YESController.sol

```
200 function borrowAllowed(address lContract, address borrower, uint borrowAmount)
201 external override returns  (uint) {
202     // Pausing is a very serious situation - we revert to sound the alarms
203     require(!borrowGuardianPaused[lContract], "borrow is paused");
    [...]
```

YESController.sol

```
459 function isDeprecated(ILending lContract) public view returns (bool) {
460     return
```

```
461         borrowGuardianPaused[address(lContract)] == true &&
462         lContract.reserveFactorMantissa() == 1e18
463     ;
464 }
```

The `borrowCaps` variable is used in the `borrowAllowed()` function in lines 226 and 228.

YESController.sol

```
200 function borrowAllowed(address lContract, address borrower, uint borrowAmount)
    external override returns  (uint) {
201     // Pausing is a very serious situation - we revert to sound the alarms
202     require(!borrowGuardianPaused[lContract], "borrow is paused");
203
204     if (!_markets[lContract].isListed) {
205         return uint(Error.MARKET_NOT_LISTED);
206     }
207
208     if (!_markets[lContract].accountMembership[borrower]) {
209         // only lContracts may call borrowAllowed if borrower not in market
210         require(msg.sender == lContract, "sender must be lContract");
211
212         // attempt to add borrower to the market
213         Error err_ = addToMarketInternal(msg.sender, borrower);
214         if (err_ != Error.NO_ERROR) {
215             return uint(err_);
216         }
217
218         // it should be impossible to break the important invariant
219         assert(_markets[lContract].accountMembership[borrower]);
220     }
221
222     if (_oracle.getLatestPrice(ILending(lContract).underlyingToken()) == 0) {
223         return uint(Error.PRICE_ERROR);
224     }
225
226     uint borrowCap = borrowCaps[lContract];
227     // Borrow cap of 0 corresponds to unlimited borrowing
228     if (borrowCap != 0) {
229         uint totalBorrows = ILending(lContract).totalBorrows();
230         uint nextTotalBorrows = add_(totalBorrows, borrowAmount);
231         require(nextTotalBorrows < borrowCap, "market borrow cap reached");
232     }
233 [...]
```

The following conditions are always evaluated to `true`, can can be removed:

- `require(!transferGuardianPaused, "transfer is paused");`
- `require(!seizeGuardianPaused, "seize is paused");`
- `require(!borrowGuardianPaused[lContract], "borrow is paused");`
- `borrowGuardianPaused[address(lContract)] == true`

The `borrowCaps[lContract]` value is always 0, so the condition that checks `if (borrowCap != 0)` is always evaluated to `false`. The codes in the condition will be unused and removable.

5.8.2. Remediation

Inspex suggests removing the unassigned variables and conditions that use them.

6. Appendix

6.1. About Inspex

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

Follow Us On:

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

6.2. References

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

[2] “Releases · ethereum/solidity.” [Online]. Available: https://github.com/ethereum/solidity/releases. [Accessed: 28-Jan-2022]

[3] “Smart Contract - Bitkub Chain Docs” [Online]. Available: https://docs.bitkubchain.org/smart-contract. [Accessed: 25-Feb-2022]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE