FAIRYPROOF

Venus TwoKindsInterest

AUDIT REPORT

Version 1.0.0

Serial No. 2024080400012023

Presented by Fairyproof

August 4, 2024

www.fairyproof.com

01. Introduction

This document includes the results of the audit performed by the Fairyproof team on the Venus TwoKindsInterest project.

Audit Start Time:

August 1, 2024

Audit End Time:

August 4, 2024

Audited Source File's Address:

https://github.com/VenusProtocol/venus-protocol/pull/494

https://github.com/VenusProtocol/isolated-pools/pull/417

Audited Source Files:

The source files audited include all the files as follows:

```
- contracts/InterestRateModels/InterestRateModelV8.sol
- contracts/InterestRateModels/TwoKinksInterestRateModel.sol
 - contracts/TwoKinksInterestRateModel.sol
```

The goal of this audit is to review Venus's solidity implementation for its TwoKindsInterest function, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

We make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

This audit only applies to the specified code, software or any materials supplied by the Venus team for specified versions. Whenever the code, software, materials, settings, environment etc is changed, the comments of this audit will no longer apply.

— Disclaimer

Note that as of the date of publishing, the contents of this report reflect the current understanding of known security patterns and state of the art regarding system security. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. If the audited source files are smart contract files, risks or issues introduced by using data feeds from offchain sources are not extended by this review either.

Given the size of the project, the findings detailed here are not to be considered exhaustive, and further testing and audit is recommended after the issues covered are fixed.

To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

— Methodology

The above files' code was studied in detail in order to acquire a clear impression of how the its specifications were implemented. The codebase was then subject to deep analysis and scrutiny, resulting in a series of observations. The problems and their potential solutions are discussed in this document and, whenever possible, we identify common sources for such problems and comment on them as well.

The Fairyproof auditing process follows a routine series of steps:

1. Code Review, Including:

- Project Diagnosis

Understanding the size, scope and functionality of your project's source code based on the specifications, sources, and instructions provided to Fairyproof.

- Manual Code Review

Reading your source code line-by-line to identify potential vulnerabilities.

- Specification Comparison

Determining whether your project's code successfully and efficiently accomplishes or executes its functions according to the specifications, sources, and instructions provided to Fairyproof.

2. Testing and Automated Analysis, Including:

- Test Coverage Analysis

Determining whether the test cases cover your code and how much of your code is exercised or executed when test cases are run.

- Symbolic Execution

Analyzing a program to determine the specific input that causes different parts of a program to execute its functions.

3. Best Practices Review

Reviewing the source code to improve maintainability, security, and control based on the latest established industry and academic practices, recommendations, and research.

— Structure of the document

This report contains a list of issues and comments on all the above source files. Each issue is assigned a severity level based on the potential impact of the issue and recommendations to fix it, if applicable. For ease of navigation, an index by topic and another by severity are both provided at the beginning of the report.

— Documentation

For this audit, we used the following source(s) of truth about how the token issuance function should work:

Website:https://venus.io/

Source Code:

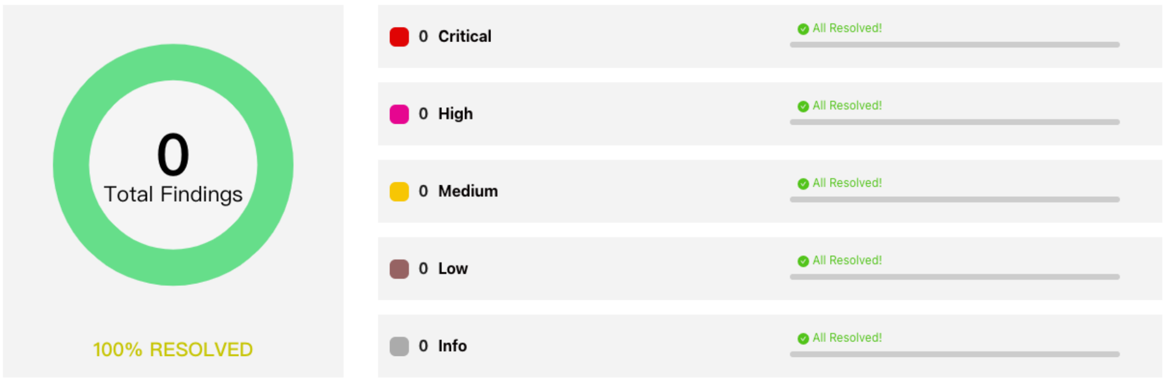
https://github.com/VenusProtocol/venus-protocol/pull/494

https://github.com/VenusProtocol/isolated-pools/pull/417

These were considered the specification, and when discrepancies arose with the actual code behavior, we consulted with the Venus team or reported an issue.

— Comments from Auditor

Serial Number	Auditor	Audit Time	Result
2024080400012023	Fairyproof Security Team	Aug 1, 2024 - Aug 4, 2024	Passed



Summary:

The Fairyproof security team used its auto analysis tools and manual work to audit the project. During the audit, no issues were uncovered.

02. About Fairyproof

Fairyproof is a leading technology firm in the blockchain industry, providing consulting and security audits for organizations. Fairyproof has developed industry security standards for designing and deploying blockchain applications.

03. Introduction to Venus

Venus Protocol (“Venus”) is an algorithmic-based money market system designed to bring a complete decentralized finance-based lending and credit system onto Ethereum,Binance Smart Chain,opBNB and Arbitrum.

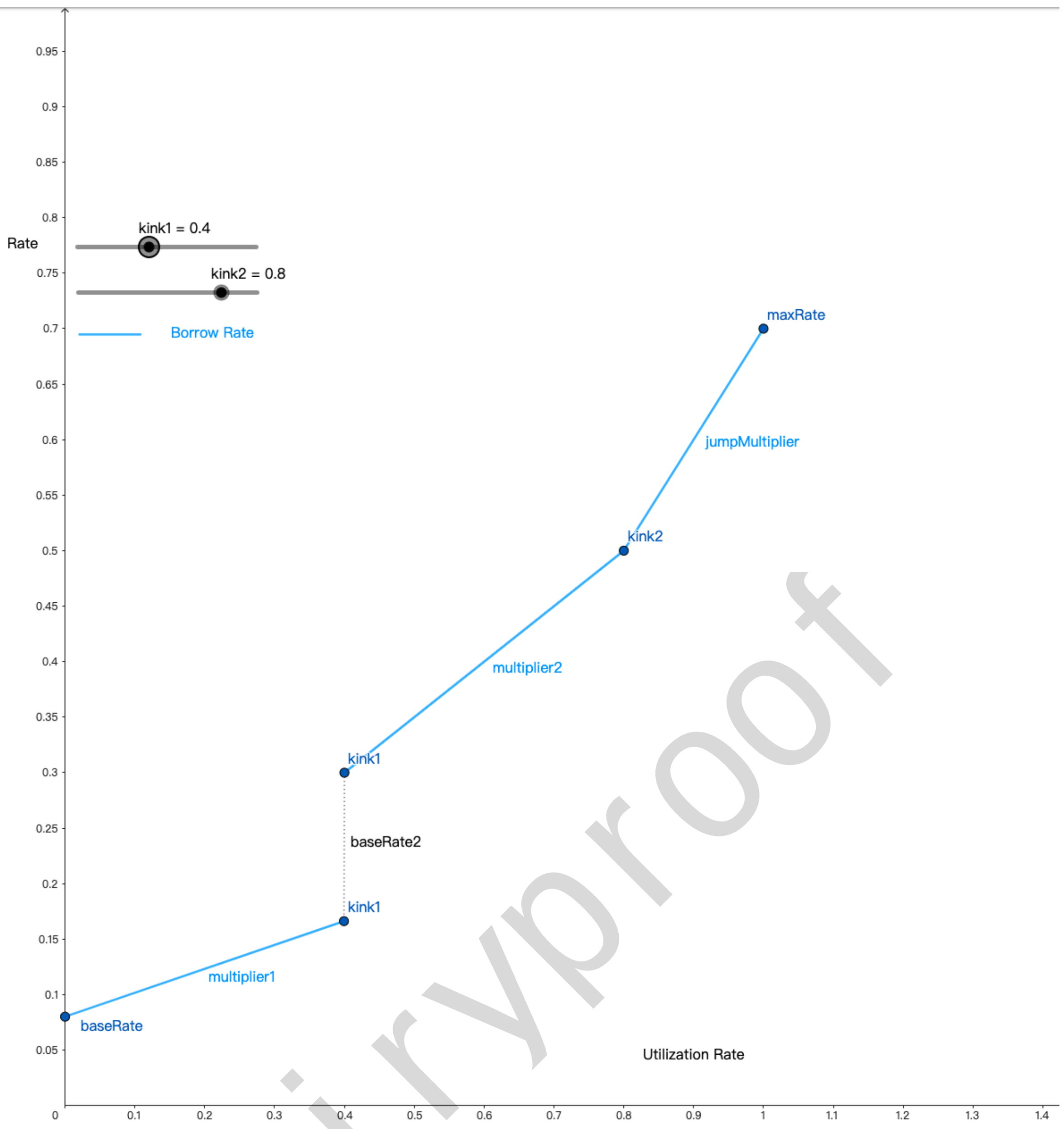
The above description is quoted from relevant documents of Venus.

04. Major functions of audited code

The audited code mainly replaced the JumpRateMode Interest with TwoKindsMode Interest.

Compared to JumpRateMode, TwoKindsMode added a new kink, thus having two kinks now, kink1 and kink2.

TwoKindsMode Interest can be depicted as follows:



When `u` reaches the first `Kink`, both the slope and the starting point get changed.
When `u` reaches the second `kink`, only the slope gets changed.

Here, `baseRate2` stands for a gap i.e. an additional base interest when `u` reaches `kink1`.

When `baseRate2` is 0, the model becomes a `JumpRateMode` that consists of two lines of different slopes. If `multiplier2 == multiplier2 || multiplier2 == jumpMultiplier`, the model becomes a conventional `JumpRateMode`.
Therefore, using `TwoKindsMode Interest` has more flexibility, achieves better precise control and is compatible with the existing interest model.

In TwoKindsMode Interest, the Utilization Rate is calculated as:

$$UtilizationRate = \frac{borrows}{cash + borrows - reserves}$$

The Borrow Rate is calculated as:

$$BorrowRate = BaseRate + UtilizationRate * Multiplier$$

In this model, different `Borrow Rates` should be calculated based on their corresponding `Us`

Therefore, `Borrow Rate = rate1 + rate2 + rate3` .

Here are the details:

- When `u` is located at `[0,kink1)`,
    1. `rate2 = rate3 = 0`
    2. `rate1 = u * multiplier + baseRate`
    3. `BorrowRate = rate1`
- When `u` is located at `[kink1,kink2)` ,
    1. `rate3 = 0`
    2. `rate1 = kink1 * multiplier + baseRate`
    3. `rate2 = (u-kink1)*multiplier2 + baseRate2`
    4. `BorrowRate = rate1 + rate2`
- When `u` is located at `[kink2,1]` ,
    1. `rate1 = kink1 * multiplier + baseRate`
    2. `rate2 = (kink2-kink1)*multiplier2 + baseRate2`,
    3. `rate3 = (u-kink2) * jumpMultiplier + 0`
    4. `Borrow Rate = rate1 + rate2 + rate3`

Supply Rate is calculated as:

$$SupplyRate = BorrowRate * UtilizationRate * (1 - reserveFactorMantissa)$$

Note:

In `Isolated Pool`, a `badDebt` is introduced. `badDebt` is equivalent to a non-profitable `borrow`. It ties up funds, so it needs to be taken into account when calculating the utilization rate. The formula for the utilization rate changes to:

$$UtilizationRate = \frac{borrows + badDebt}{cash + borrows + badDebt - reserves}$$

However, `badDebt` does not generate interests, so it needs to be deducted when calculating the `Supply Rate`. The formula for the `Supply Rate` changes to:

$$DistributeRate = \frac{borrows}{cash + borrows + badDebt - reserves}$$

$$SupplyRate = BorrowRate * DistributeRate * (1 - reserveFactorMantissa)$$

`Borrow Rate`'s formulate remains unchanged.

Note:

The max value of `UtilizationRate` is 100%.

The min value of `Borrow Rate` is 0.

05. Coverage of issues

The issues that the Fairyproof team covered when conducting the audit include but are not limited to the following ones:

- Access Control
- Admin Rights
- Arithmetic Precision
- Code Improvement
- Contract Upgrade/Migration
- Delete Trap
- Design Vulnerability
- DoS Attack
- EOA Call Trap
- Fake Deposit
- Function Visibility
- Gas Consumption
- Implementation Vulnerability
- Inappropriate Callback Function
- Injection Attack
- Integer Overflow/Underflow
- IsContract Trap
- Miner's Advantage
- Misc
- Price Manipulation
- Proxy selector clashing
- Pseudo Random Number
- Re-entrancy Attack
- Replay Attack
- Rollback Attack
- Shadow Variable
- Slot Conflict
- Token Issuance
- Tx.origin Authentication
- Uninitialized Storage Pointer

06. Severity level reference

Every issue in this report was assigned a severity level from the following:

Critical severity issues need to be fixed as soon as possible.

High severity issues will probably bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Informational is not an issue or risk but a suggestion for code improvement.

07. Major areas that need attention

Based on the provided source code the Fairyproof team focused on the possible issues and risks related to the following functions or areas.

- Function Implementation

We checked whether or not the functions were correctly implemented.
We didn't find issues or risks in these functions or areas at the time of writing.

- Access Control

We checked each of the functions that could modify a state, especially those functions that could only be accessed by owner or administrator
We didn't find issues or risks in these functions or areas at the time of writing.

- Token Issuance & Transfer

We examined token issuance and transfers for situations that could harm the interests of holders.
We didn't find issues or risks in these functions or areas at the time of writing.

- State Update

We checked some key state variables which should only be set at initialization.
We didn't find issues or risks in these functions or areas at the time of writing.

- Asset Security

We checked whether or not all the functions that transfer assets were safely handled.
We didn't find issues or risks in these functions or areas at the time of writing.

- Miscellaneous

We checked the code for optimization and robustness.
We didn't find issues or risks in these functions or areas at the time of writing.

08. issues by severity

- N/A

09. Issue descriptions

- N/A

10. Recommendations to enhance the overall security

We list some recommendations in this section. They are not mandatory but will enhance the overall security of the system if they are adopted.

`BASE_RATE_2_PER_BLOCK` is actually the value added relative to `BASE_RATE_PER_BLOCK`, so it is better to rename it to `BASE_RATE_PER_BLOCK_ADDED` for clearer description.

11. Appendices

11.1 Unit Test

1. InterestRateModel01.t.js

```
const {
    loadFixture,
} = require("@nomicfoundation/hardhat-network-helpers");

const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("InterestRateModel Unit test", function() {
    const {getBigInt} = ethers;
    const EXP_SCALE = ethers.WeiPerEther;
    const HUNDRED = getBigInt(100);

    const kink1 = EXP_SCALE * getBigInt(40) / HUNDRED;
    const kink2 = EXP_SCALE * getBigInt(80) / HUNDRED;
    const base_rate = EXP_SCALE * getBigInt(5) / HUNDRED;
    const added_base_rate = EXP_SCALE * getBigInt(10) / HUNDRED;
    const block_per_year = getBigInt(60 * 60 * 24 * 365 / 3);
    const multiplier1 =  EXP_SCALE * getBigInt(5) / HUNDRED;
    const multiplier2 =  EXP_SCALE * getBigInt(8) / HUNDRED;
    const jump = EXP_SCALE * getBigInt(10) / HUNDRED;

    async function  deployFixture() {
        const TwoKinksInterestRateModel = await
ethers.getContractFactory("contracts/venus_ven_2663/TwoKinksInterestRateModel.sol
:TwoKinksInterestRateModel");
        let args = [
            base_rate * block_per_year,
            multiplier1 * block_per_year,
            kink1,
            multiplier2 * block_per_year,
            added_base_rate * block_per_year,
            kink2,
            jump * block_per_year
        ];
        const instance = await TwoKinksInterestRateModel.deploy(...args);
        return instance;

    }

    describe("utilizationRate test", function() {
        it("utilizationRate should be borrow div supply", async () => {
            const instance = await loadFixture(deployFixture);
            const borrows = EXP_SCALE;
            const cash = getBigInt(2) *EXP_SCALE;
            const reserves = EXP_SCALE * getBigInt(5) /HUNDRED;
            let rate = borrows * EXP_SCALE / (borrows + cash - reserves);
            expect(await
instance.utilizationRate(cash,borrows,reserves)).eq(rate);
        });

        it("utilizationRate should not greater than SCALE", async () => {
            const instance = await loadFixture(deployFixture);
            const borrows = EXP_SCALE;
            const cash = ethers.getBigInt(0);
            const reserves = EXP_SCALE * getBigInt(5) /HUNDRED;
            let rate = borrows * EXP_SCALE / (borrows + cash - reserves);
            expect(rate).gt(EXP_SCALE);
            expect(await
instance.utilizationRate(cash,borrows,reserves)).eq(EXP_SCALE);
        });
    });

    describe("BorrowRate and SupplyRate test", function() {
        it("BorrowRate and SupplyRate while utilization rate is little than
kink1", async () => {
            const instance = await loadFixture(deployFixture);
            const borrows = EXP_SCALE;
            const cash = getBigInt(2) *EXP_SCALE;
            const reserves = EXP_SCALE * getBigInt(5) /HUNDRED;

            let utilization_rate = borrows * EXP_SCALE / (borrows + cash -
reserves);
            expect(utilization_rate).lt(kink1);
            console.log("utilization_rate:
",ethers.formatEther(utilization_rate));

            let borrow_rate = base_rate + utilization_rate * multiplier1 /
EXP_SCALE;
            expect(await
instance.getBorrowRate(cash,borrows,reserves)).eq(borrow_rate);
            console.log("borrow_rate: ",ethers.formatEther(borrow_rate));

            let reserveFactorMantissa = EXP_SCALE * getBigInt(5) / HUNDRED;
            let supply_rate = borrow_rate * (EXP_SCALE - reserveFactorMantissa) /
EXP_SCALE * utilization_rate / EXP_SCALE;
            expect(await
instance.getSupplyRate(cash,borrows,reserves,reserveFactorMantissa)).eq(supply_ra
te);
            console.log("supply_rate: ",ethers.formatEther(supply_rate));
        });

        it("BorrowRate and SupplyRate while utilization rate is greater than
kink1 and little than kink2", async () => {
            const instance = await loadFixture(deployFixture);
            const borrows = getBigInt(2) * EXP_SCALE;
            const cash = EXP_SCALE;
            const reserves = EXP_SCALE * getBigInt(5) /HUNDRED;
            let utilization_rate = borrows * EXP_SCALE / (borrows + cash -
reserves);
            expect(utilization_rate).gt(kink1);
            expect(utilization_rate).lt(kink2);
            console.log("utilization_rate:
",ethers.formatEther(utilization_rate));

            let borrow_rate1 = base_rate + kink1 * multiplier1 / EXP_SCALE;
            let borrow_rate2 = added_base_rate + (utilization_rate - kink1) *
multiplier2 / EXP_SCALE;
            let borrow_rate = borrow_rate1 + borrow_rate2;
            expect(await
instance.getBorrowRate(cash,borrows,reserves)).eq(borrow_rate);
            console.log("borrow_rate: ",ethers.formatEther(borrow_rate));

            let reserveFactorMantissa = EXP_SCALE * getBigInt(5) / HUNDRED;
            let supply_rate = borrow_rate * (EXP_SCALE - reserveFactorMantissa) /
EXP_SCALE * utilization_rate / EXP_SCALE;
            expect(await
instance.getSupplyRate(cash,borrows,reserves,reserveFactorMantissa)).eq(supply_ra
te);
            console.log("supply_rate: ",ethers.formatEther(supply_rate));
        });


        it("BorrowRate and SupplyRate while utilization rate is greater than
kink2", async () => {
            const instance = await loadFixture(deployFixture);
            const borrows = getBigInt(4) * EXP_SCALE;
            const cash = EXP_SCALE;
            const reserves = EXP_SCALE * getBigInt(5) /HUNDRED;

            let utilization_rate = borrows * EXP_SCALE / (borrows + cash -
reserves);
            expect(utilization_rate).gt(kink2);

            console.log("utilization_rate:
",ethers.formatEther(utilization_rate));

            let borrow_rate1 = base_rate + kink1 * multiplier1 / EXP_SCALE;
            let borrow_rate2 = added_base_rate + (kink2 - kink1) * multiplier2 /
EXP_SCALE;
            let borrow_rate3 = (utilization_rate - kink2) * jump / EXP_SCALE;
            let borrow_rate = borrow_rate1 + borrow_rate2 + borrow_rate3;
            expect(await
instance.getBorrowRate(cash,borrows,reserves)).eq(borrow_rate);
            console.log("borrow_rate: ",ethers.formatEther(borrow_rate));
            let reserveFactorMantissa = EXP_SCALE * getBigInt(5) / HUNDRED;
            let supply_rate = borrow_rate * (EXP_SCALE - reserveFactorMantissa) /
EXP_SCALE * utilization_rate / EXP_SCALE;
            expect(await
instance.getSupplyRate(cash,borrows,reserves,reserveFactorMantissa)).eq(supply_ra
te);
            console.log("supply_rate: ",ethers.formatEther(supply_rate));
        });
    });
```

```
});
```

2. InterestRateModel02.t.js

```
const {
    loadFixture,
} = require("@nomicfoundation/hardhat-network-helpers");

const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("InterestRateModel Unit test", function() {
    const {getBigInt} = ethers;
    const EXP_SCALE = ethers.WeiPerEther;
    const HUNDRED = getBigInt(100);

    const kink1 = EXP_SCALE * getBigInt(40) / HUNDRED;
    const kink2 = EXP_SCALE * getBigInt(80) / HUNDRED;
    const base_rate = EXP_SCALE * getBigInt(5) / HUNDRED;
    const added_base_rate = EXP_SCALE * getBigInt(10) / HUNDRED;
    const block_per_year = getBigInt(60 * 60 * 24 * 365 / 3);
    const multiplier1 =  EXP_SCALE * getBigInt(5) / HUNDRED;
    const multiplier2 =  EXP_SCALE * getBigInt(8) / HUNDRED;
    const jump = EXP_SCALE * getBigInt(10) / HUNDRED;

    async function  deployFixture() {
        const TwoKinksInterestRateModel = await
ethers.getContractFactory("contracts/venus_ven_2263_isolated/TwoKinksInterestRate
Model.sol:TwoKinksInterestRateModel");
        let args = [
            base_rate * block_per_year,
            multiplier1 * block_per_year,
            kink1,
            multiplier2 * block_per_year,
            added_base_rate * block_per_year,
            kink2,
            jump * block_per_year,
            false,
            block_per_year
        ];
        const instance = await TwoKinksInterestRateModel.deploy(...args);
        return instance;

    }

    describe("utilizationRate test", function() {
        it("utilizationRate should be borrow div supply", async () => {
            const instance = await loadFixture(deployFixture);
            const badDebt = EXP_SCALE * getBigInt(1) / HUNDRED;
            const borrows = EXP_SCALE;
            const cash = getBigInt(2) *EXP_SCALE;
            const reserves = EXP_SCALE * getBigInt(5) /HUNDRED;
            let rate = EXP_SCALE * (borrows + badDebt) / (borrows + badDebt +
cash - reserves);
            expect(await
instance.utilizationRate(cash,borrows,reserves,badDebt)).eq(rate);
        });

        it("utilizationRate should not greater than SCALE", async () => {
            const instance = await loadFixture(deployFixture);
            const borrows = EXP_SCALE;
            const badDebt = EXP_SCALE * getBigInt(1) / HUNDRED;
            const cash = ethers.getBigInt(0);
            const reserves = EXP_SCALE * getBigInt(5) /HUNDRED;
            let rate = EXP_SCALE * (borrows + badDebt) / (borrows + badDebt +
cash - reserves);
            expect(rate).gt(EXP_SCALE);
            expect(await
instance.utilizationRate(cash,borrows,reserves,badDebt)).eq(EXP_SCALE);
        });
    });


    describe("BorrowRate and SupplyRate test", function() {
        it("BorrowRate and SupplyRate while utilization rate is little than
kink1", async () => {
            const instance = await loadFixture(deployFixture);
            const borrows = EXP_SCALE;
            const badDebt = EXP_SCALE * getBigInt(1) / HUNDRED;
            const cash = getBigInt(2) *EXP_SCALE;
            const reserves = EXP_SCALE * getBigInt(5) /HUNDRED;

            let utilization_rate = EXP_SCALE * (borrows + badDebt) / (borrows +
badDebt + cash - reserves);
            let valid_rate = EXP_SCALE * (borrows) / (borrows + badDebt + cash -
reserves); // only borrows can get interest
            expect(utilization_rate).lt(kink1);
            console.log("utilization_rate:
",ethers.formatEther(utilization_rate));

            let borrow_rate = base_rate + utilization_rate * multiplier1 /
EXP_SCALE;
            expect(await
instance.getBorrowRate(cash,borrows,reserves,badDebt)).eq(borrow_rate);
            console.log("borrow_rate: ",ethers.formatEther(borrow_rate));

            let reserveFactorMantissa = EXP_SCALE * getBigInt(5) / HUNDRED;
            let supply_rate = borrow_rate * (EXP_SCALE - reserveFactorMantissa) /
EXP_SCALE * valid_rate / EXP_SCALE;
            expect(await
instance.getSupplyRate(cash,borrows,reserves,reserveFactorMantissa,badDebt)).eq(s
upply_rate);
            console.log("supply_rate: ",ethers.formatEther(supply_rate));
        });
        it("BorrowRate and SupplyRate while utilization rate is greater than
kink1 and little than kink2", async () => {
            const instance = await loadFixture(deployFixture);
            const borrows = getBigInt(2) * EXP_SCALE;
            const badDebt = EXP_SCALE * getBigInt(1) / HUNDRED;
            const cash = EXP_SCALE;
            const reserves = EXP_SCALE * getBigInt(5) /HUNDRED;

            let utilization_rate = EXP_SCALE * (borrows + badDebt) / (borrows +
badDebt + cash - reserves);
            let valid_rate = EXP_SCALE * (borrows) / (borrows + badDebt + cash -
reserves); // only borrows can get interest
            expect(utilization_rate).gt(kink1);
            expect(utilization_rate).lt(kink2);
            console.log("utilization_rate:
",ethers.formatEther(utilization_rate));

            let borrow_rate1 = base_rate + kink1 * multiplier1 / EXP_SCALE;
            let borrow_rate2 = added_base_rate + (utilization_rate - kink1) *
multiplier2 / EXP_SCALE;
            let borrow_rate = borrow_rate1 + borrow_rate2;
            expect(await
instance.getBorrowRate(cash,borrows,reserves,badDebt)).eq(borrow_rate);
            console.log("borrow_rate: ",ethers.formatEther(borrow_rate));

            let reserveFactorMantissa = EXP_SCALE * getBigInt(5) / HUNDRED;
            let supply_rate = borrow_rate * (EXP_SCALE - reserveFactorMantissa) /
EXP_SCALE * valid_rate / EXP_SCALE;
            expect(await
instance.getSupplyRate(cash,borrows,reserves,reserveFactorMantissa,badDebt)).eq(s
upply_rate);
            console.log("supply_rate: ",ethers.formatEther(supply_rate));
        });


        it("BorrowRate and SupplyRate while utilization rate is greater than
kink2", async () => {
            const instance = await loadFixture(deployFixture);
            const borrows = getBigInt(4) * EXP_SCALE;
            const badDebt = EXP_SCALE * getBigInt(1) / HUNDRED;
            const cash = EXP_SCALE;
            const reserves = EXP_SCALE * getBigInt(5) /HUNDRED;

            let utilization_rate = EXP_SCALE * (borrows + badDebt) / (borrows +
badDebt + cash - reserves);
            let valid_rate = EXP_SCALE * (borrows) / (borrows + badDebt + cash -
reserves);
            expect(utilization_rate).gt(kink2);
            console.log("utilization_rate:
",ethers.formatEther(utilization_rate));

            let borrow_rate1 = base_rate + kink1 * multiplier1 / EXP_SCALE;
            let borrow_rate2 = added_base_rate + (kink2 - kink1) * multiplier2 /
EXP_SCALE;
            let borrow_rate3 = (utilization_rate - kink2) * jump / EXP_SCALE;
            let borrow_rate = borrow_rate1 + borrow_rate2 + borrow_rate3;
            expect(await
instance.getBorrowRate(cash,borrows,reserves,badDebt)).eq(borrow_rate);
            console.log("borrow_rate: ",ethers.formatEther(borrow_rate));
            let reserveFactorMantissa = EXP_SCALE * getBigInt(5) / HUNDRED;
            let supply_rate = borrow_rate * (EXP_SCALE - reserveFactorMantissa) /
EXP_SCALE * valid_rate / EXP_SCALE;
            expect(await
instance.getSupplyRate(cash,borrows,reserves,reserveFactorMantissa,badDebt)).eq(s
upply_rate);
            console.log("supply_rate: ",ethers.formatEther(supply_rate));
        });
    });
});
```

3. UnitTestOutput

```
  InterestRateModel Unit test
    utilizationRate test
      ✓ utilizationRate should be borrow div supply (1157ms)
      ✓ utilizationRate should not greater than SCALE
    BorrowRate and SupplyRate test
utilization_rate:  0.338983050847457627
borrow_rate:  0.066949152542372881
supply_rate:  0.021559896581442113
      ✓ BorrowRate and SupplyRate while utilization rate is little than kink1
utilization_rate:  0.677966101694915254
borrow_rate:  0.192237288135593322
supply_rate:  0.123813846595805802
      ✓ BorrowRate and SupplyRate while utilization rate is greater than kink1
and little than kink2
utilization_rate:  0.808080808080808080
borrow_rate:  0.202808080808080808
supply_rate:  0.155691051933476175
      ✓ BorrowRate and SupplyRate while utilization rate is greater than kink2

 InterestRateModel of Isolated Pool Unit test
    utilizationRate test
      ✓ utilizationRate should be borrow div supply (856ms)
      ✓ utilizationRate should not greater than SCALE
    BorrowRate and SupplyRate test
utilization_rate:  0.341216216216216216
borrow_rate:  0.067060810810810810
supply_rate:  0.021522895361577793
      ✓ BorrowRate and SupplyRate while utilization rate is little than kink1
utilization_rate:  0.679054054054054054
borrow_rate:  0.192324324324324324
supply_rate:  0.123451424397370342
      ✓ BorrowRate and SupplyRate while utilization rate is greater than kink1
and little than kink2
utilization_rate:  0.808467741935483870
borrow_rate:  0.202846774193548387
supply_rate:  0.155406802809573360
```

```
    ✓ BorrowRate and SupplyRate while utilization rate is greater than kink2


  10 passing (1884ms)
```

11.2 External Functions Check Points

1. TwokindsInterestRateModel.sol.output.md

File: contracts/venus_ven_2663/TwoKinksInterestRateModel.sol

(Empty fields in the table represent things that are not required or relevant)

contract: TwoKinksInterestRateModel is InterestRateModelV8

Index	Function	Visibility	StateMutability	Permission Check	Param Check	IsUserInterface	Unit Test
1	getBorrowRate(uint256,uint256,uint256)	external	view			False	Passed
2	getSupplyRate(uint256,uint256,uint256,uint256)	public	view			False	Passed
3	utilizationRate(uint256,uint256,uint256)	public	pure			False	Passed

File: contracts/venus_ven_2263_isolated/TwoKinksInterestRateModel.sol

(Empty fields in the table represent things that are not required or relevant)

contract: TwoKinksInterestRateModel is InterestRateModel, TimeManagerV8

Index	Function	Visibility	StateMutability	Permission Check	Param Check	IsUserInterface	Unit Test
1	getBorrowRate(uint256,uint256,uint256,uint256)	external	view			False	Passed
2	getSupplyRate(uint256,uint256,uint256,uint256,uint256)	public	view			False	Passed
3	utilizationRate(uint256,uint256,uint256,uint256)	public	pure			False	Passed

FAIRYPROOF

https://medium.com/@FairyproofT

https://twitter.com/FairyproofT

https://www.linkedin.com/company/fairyproof-tech

https://t.me/Fairyproof_tech

Reddit: https://www.reddit.com/user/FairyproofTech