Venus Vault

AUDIT REPORT

Version 1.0.0

Serial No. 20230517000120 23

Presented by Fairyproof

May 17, 2023

01. Introduction

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

Audit Start Time:

April 29, 2023

Audit End Time:

May 16, 2023

Audited Code's Github Repository:

https://github.com/VenusProtocol/venus-protocol/tree/feature/vault-upgrades

Audited Code's Github Commit Number When Audit Started:

9909ca76e7e7be7e896b202ed1869583edf4fe80

Audited Code's Github Commit Number When Audit Ended:

34523237bffbce8c7b142b3d6d4073d55745ffa1

Audited Source Files:

The source files audited include all the files as follows:

```
contracts
├── Utils
│   ├── Address.sol
│   ├── ECDSA.sol
│   ├── IBEP20.sol
│   ├── SafeBEP20.sol
│   ├── SafeCast.sol
│   └── SafeMath.sol
├── VRTVault
│   ├── VRTVault.sol
│   ├── VRTVaultProxy.sol
│   └── VRTVaultStorage.sol
└── XVSVault
    ├── XVSStore.sol
    ├── XVSVault.sol
    ├── XVSVaultErrorReporter.sol
    ├── XVSVaultProxy.sol
    └── XVSVaultStorage.sol

3 directories, 14 files
```

The goal of this audit is to review Venus's solidity implementation for its Vault 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/

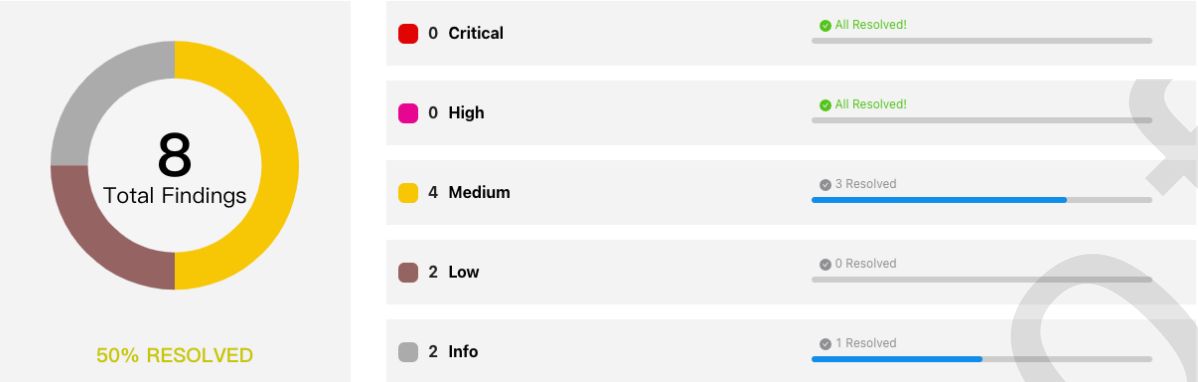
Whitepaper:https://venus.io/Whitepaper.pdf

Source Code: https://github.com/VenusProtocol/venus-protocol/tree/feature/vault-upgrades

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
2023051700012023	Fairyproof Security Team	Apr 29, 2023 - May 16, 2023	Medium Risk



Summary:

The Fairyproof security team used its auto analysis tools and manual work to audit the project. During the audit, four issues of medium-severity, two issues of low-severity and two issues of info-severity were uncovered. The Venus team fixed three issues of medium and one issue of info, and acknowledged the remaining issues.

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 Binance Smart Chain.

The above description is quoted from relevant documents of Venus.

04. Major functions of audited code

- The staking function allows users to stake various crypto assets to win rewards in BEP-20 tokens.
- All the Vault related contracts are upgradeable and implemented as a proxy/implementation mode. The admin 's access control should be managed with great care and transferred to a multi-sig wallet whenever necessary.

Note: taxed tokens cannot be staked in the vaults.

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 found some issues, for more details please refer to [FP-2,FP-4,FP-6,FP-7] in "09. Issue description".

- 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 found one issue, for more details please refer to [FP-5] in "09. Issue description".

- 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 found some issues, for more details please refer to [FP-1,FP-3] in "09. Issue description".

- Miscellaneous

We checked the code for optimization and robustness.
We found one issue, for more details please refer to [FP-8] in "09. Issue description".

08. List of issues by severity

Index	Title	Issue/Risk	Severity	Status
FP-1	`withdrawBep20` Has Excessive Withdrawal Righ	Design Vulnerability	Medium	Acknowledged
FP-2	User Can Deposit Before Reward Transfer	Design Vulnerability	Low	Acknowledged
FP-3	Staked Asset and Reward Asset Not Separated	Implementation Vulnerability	Low	Acknowledged
FP-4	Reward Tokens "Swallowed"	Design Vulnerability	Info	Acknowledged
FP-5	Admin's Access Control Can be Recovered	Admin Rights	Medium	✓ Fixed
FP-6	Core Parameters Can be Reset	Design Vulnerability	Medium	✓ Fixed
FP-7	Flawed Reward for Multiple Pools with Same Staked Token	Implementation Vulnerability	Medium	✓ Fixed
FP-8	Code Improvement	Code Improvement	Info	✓ Fixed

09. Issue descriptions

[FP-1] `withdrawBep20` Has Excessive Withdrawal Righ

Design Vulnerability Medium Acknowledged

Issue/Risk: Design Vulnerability

Description:

In `VRTVault/VRTVault.sol`, the withdrawBep20 function can be called to withdraw any tokens including staked assets (vrt).

Recommendation:

Consider adding a restriction that disallows the vrt token to be withdrawn.

Update/Status:

The Venus team replied that they would execute withdrawals with great care and preferred to keep it for the time being.

[FP-2] User Can Deposit Before Reward Transfer

Design Vulnerability | Low | Acknowledged

Issue/Risk: Design Vulnerability

Description:

In Vault/VAIVault.sol , as long as a user deposits the vai token before the reward is updated, the user can get rewards in vxs no matter how long the vai token has been deposited in the vault. Therefore when a user can detect a transaction that deposits reward tokens into the vault in the memory pool, he/she can deposit tokens before this transaction and get rewards. This can be repeated and the application would be exploited.

Recommendation:

Consider adding a requirement to check the staking period.

Update/Status:

The Venus team replied that this is a legitimate issue and preferred to keep it for now.

[FP-3] Staked Asset and Reward Asset Not Separated

Implementation Vulnerability | Low | Acknowledged

Issue/Risk: Implementation Vulnerability

Description:

In the contract's implementation the staked asset and the reward assets are both vrt and they are not separated. Since it is impossible the calculate an interest for every user's staked assets and there is not upper bound for the interest, it is possible that the tokens in the contract are insufficient for paying interests. In this case when a user withdraws his/her interest, other users' staked assets may be withdrawn as this user's interest.

Recommendation:

Consider using a variable to keep the status of all users' staked assets and making sure after withdrawing interests or staked assets, the balance should be greater than all users' staked assets. In addition, consider redefining the withdraw function to handle staked assets and interests separately.

Update/Status:

The Venus team replied they would be careful with their operation and maintenance, and preferred to keep it for the time being

[FP-4] Reward Tokens "Swallowed"

Design Vulnerability | Info | Acknowledged

Issue/Risk: Design Vulnerability

Description:

In `Vault/VAIVault.sol` , transferring the xvs token to the contract and calling updatePendingRewards to update rewards are not executed in a single transaction. Other operations can happen between these two operations. When xvs is distributed as a reward token, safeXVSTransfer will update xvsBalance with the latest balance of the xvs token and xvsBalance will include the reward that is transferred into the vault but hasn't been updated. In this case the xvs tokens that are transferred into the vault will be swallowed and will not be distributed as rewards or be withdrawn.

Recommendation:

Consider defining updatePendingRewards as public and adding a directive to call this function at the beginning of the implementation of the updateVault function.

Update/Status:

The Venus team replied that the rewards are distributed in an automated fashion in the Comptroller contract, so it is currently impossible that updatePendingRewards is not called. The recommendation sounds reasonable and it could help prevent "reward swallowing" in case Comptroller code is modified to remove this call. However, it is important to limit the scope of the changes for the upgrade, so they will postpone fixing this for now.

[FP-5] Admin's Access Control Can be Recovered

Admin Rights Medium ✓ Fixed

Issue/Risk: Admin Rights

Description:

In `Vault/VAIVault.sol`, the `burnAdmin` doesn't check whether pendingAdmin is zero. If `_setPendingAdmin` is called to set pendingAdmin to address A, and then burnAdmin or setNewAdmin is called to abandon or transfer admin 's access control, address A can call `_acceptAdmin` to get admin 's access control.

Recommendation:

Consider allowing only one function to transfer admin 's access control or adding `pendingAdmin = address(0)` in both `setNewAdmin` and `burnAdmin` to prevent admin't access control from being recovered.

Update/Status:

The Venus team has removed `burnAdmin` and `setNewAdmin`.

[FP-6] Core Parameters Can be Reset

Design Vulnerability Medium ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

In Vault/VAIVault.sol , the setVenusInfo function can be called repeatedly. And this would result in unexpected issues. If the settings for xvs and vai are incorrectly set, users staked assets may be lost.

Recommendation:

Consider allowing this function to be called only once.

Update/Status:

The Venus team has fixed the issue.

[FP-7] Flawed Reward for Multiple Pools with Same Staked Token

Implementation Vulnerability Medium ✓ Fixed

Issue/Risk: Implementation Vulnerability

Description:

In `XVSVault/XVSVault.sol`, the implementation allows various tokens to be rewardToken s. However when these various rewardToken s are for a same staked token e.g. USDT , calculating the staked asset's supply calculates all vaults' USDT balances rather than a single rewardToken 's USDT balance. In this case, the calculated reward is less than the actual reward.

Recommendation:

Consider disallowing a single token to be universally used as a same staked token for various rewardTokens.

Update/Status:

The Venus team has disallowed adding pools with the same staked token.

[FP-8] Code Improvement

Code Improvement Info ✓ Fixed

Issue/Risk: Code Improvement

Description:

- In `XVSVault.sol`, the `setRewardAmountPerBlock` function doesn't check whether `rewardToken` is effective.
- In `XVSVault.sol`, consider adding a `if (pending >0)` conditional check in the deposit function and changing `address(msg.sender)` to `msg.sender`.
- In `XVSVault.sol`, `executeWithdrawal` function , in the condition checks , it should be clearly state that at least one shouldn't be greater than 0 (and one should be equal to 0).
- In 'VAIVault.sol `,the` getAdmin `function is redundant since when` admin` is public , it is readable.

Recommendation:

Optimize relevant code to increase robustness.

Update:

The Venus team has disallowed adding pools with the same staked token.

Status:

The Venus team has fixed these issues.

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.

- Consider managing the admin's access control with great care and transfering it to a multi-sig wallet or DAO when necessary.

11. Appendices

11.1 Unit Test

1. VAIVault.js

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

const zero_address = ethers.constants.AddressZero;

describe("VAIVault Unit Test", function () {
    async function deployAndBindFixture() {
        // get users;
        const [Owner, Alice,Bob,...users] = await ethers.getSigners();
        // Deploy access_controller
        const MockAccessControlManagerV5 = await ethers.getContractFactory("MockAccessControlManagerV5");
        const access_controller = await MockAccessControlManagerV5.deploy();
        // deploy VAIVault;
        const VAIVault = await ethers.getContractFactory("VAIVault");
        const vault = await VAIVault.deploy();
        // deploy proxy
        const VAIVaultProxy = await ethers.getContractFactory("VAIVaultProxy");
        let proxy = await VAIVaultProxy.deploy();
        // binding
        await proxy._setPendingImplementation(vault.address);
        await vault._become(proxy.address);
        proxy = VAIVault.attach(proxy.address);
        await proxy.setAccessControl(access_controller.address);
        // deploy token
        const MockERC20 = await ethers.getContractFactory("MockERC20");
        const xvs = await MockERC20.deploy("XVS","XVS",ethers.utils.parseEther("100000000"));
        const vai = await MockERC20.deploy("VAI","VAI",ethers.utils.parseEther("100000000"));
        await proxy.setVenusInfo(xvs.address,vai.address);
        // return
        return {
            Owner,Alice,Bob,users,proxy,vault,xvs,vai,access_controller
        };
    }

    describe("Initial state check", function() {
        it("VAIVaultStorage state check", async function() {
            const {Owner,proxy,vault,xvs,vai,access_controller} = await loadFixture(deployAndBindFixture);
            expect(await proxy.admin()).to.equal(Owner.address);
```

```
            expect(await proxy.pendingAdmin()).to.equal(zero_address);
            expect(await proxy.vaiVaultImplementation()).to.equal(vault.address);
            expect(await proxy.pendingVAIVaultImplementation()).to.equal(zero_address);
            expect(await proxy.xvs()).to.equal(xvs.address);
            expect(await proxy.vai()).to.equal(vai.address);
            expect(await proxy.xvsBalance()).to.equal(0);
            expect(await proxy.accXVSPerShare()).to.equal(0);
            expect(await proxy.pendingRewards()).to.equal(0);
            expect(await proxy.vaultPaused()).to.equal(false);
            expect(await proxy.accessControlManager()).to.equal(access_controller.address);
        });
    });

    describe("Change admin and implement test", function() {
        it("only admin can change admin or implement", async function() {
            const {Alice,proxy,users} = await loadFixture(deployAndBindFixture);
            const VAIVaultProxy = await ethers.getContractFactory("VAIVaultProxy");
            let instance = VAIVaultProxy.attach(proxy.address);
            await expect(instance.connect(Alice)._setPendingAdmin(users[0].address)).to.emit(
                instance,"Failure"
            ).withArgs(1,2,0);
            expect(await proxy.pendingAdmin()).to.equal(zero_address);

            await expect(instance.connect(Alice)._setPendingImplementation(users[0].address)).to.emit(
                instance,"Failure"
            ).withArgs(1,3,0);
            expect(await proxy.pendingVAIVaultImplementation()).to.equal(zero_address);
        });

        it("only pending can accept", async function() {
            const {Owner,Alice,Bob,proxy,vault} = await loadFixture(deployAndBindFixture);
            const VAIVaultProxy = await ethers.getContractFactory("VAIVaultProxy");
            let instance = VAIVaultProxy.attach(proxy.address);

            // set pending
            await expect(instance._setPendingAdmin(Alice.address)).to.emit(
                instance,"NewPendingAdmin"
            ).withArgs(zero_address,Alice.address);
            expect(await proxy.pendingAdmin()).to.equal(Alice.address);
            await expect(instance._setPendingImplementation(Bob.address)).to.emit(
                instance,"NewPendingImplementation"
            ).withArgs(zero_address,Bob.address);
            expect(await proxy.pendingVAIVaultImplementation()).to.equal(Bob.address);

            // accept without pending
            await expect(instance._acceptAdmin()).to.emit(
                instance,"Failure"
            ).withArgs(1,0,0);
            expect(await proxy.pendingAdmin()).to.equal(Alice.address);
            await expect(instance._acceptImplementation()).to.emit(
                instance,"Failure"
            ).withArgs(1,1,0);
            expect(await proxy.pendingVAIVaultImplementation()).to.equal(Bob.address);

            // accept with pending
            await expect(instance.connect(Alice)._acceptAdmin()).to.emit(
                instance,"NewAdmin"
            ).withArgs(Owner.address,Alice.address);

            await expect(instance.connect(Bob)._acceptImplementation()).to.emit(
                instance,"NewImplementation"
            ).withArgs(vault.address,Bob.address);

            // check final state
            expect(await proxy.admin()).to.equal(Alice.address);
            expect(await proxy.pendingAdmin()).to.equal(zero_address);
            expect(await proxy.vaiVaultImplementation()).to.equal(Bob.address);
            expect(await proxy.pendingVAIVaultImplementation()).to.equal(zero_address);
        });
    });

    describe("Pause and resume test", function() {
        it("pause  and resume contract test", async function() {
```

```
            const {Owner,Alice,proxy} = await loadFixture(deployAndBindFixture);
            // only Owner
            await expect(proxy.connect(Alice).pause()).to.revertedWith("Unauthorized");
            await expect(proxy.connect(Alice).resume()).to.revertedWith("Unauthorized");
            // pause emit event
            await expect(proxy.pause()).to.emit(
                proxy,"VaultPaused"
            ).withArgs(Owner.address);
            expect(await proxy.vaultPaused()).to.equal(true);
            // pause twice should be failed
            await expect(proxy.pause()).to.revertedWith("Vault is already paused");

            // resume emit event
            await expect(proxy.resume()).to.emit(
                proxy,"VaultResumed"
            ).withArgs(Owner.address);
            expect(await proxy.vaultPaused()).to.equal(false);
            // resume twice should be failed
            await expect(proxy.resume()).to.revertedWith("Vault is not paused");
        });
    });

    //
    describe("setVenusInfo test", function() {
        it("only admin can set info", async function() {
            const {proxy,Alice,users} = await loadFixture(deployAndBindFixture);
            await
expect(proxy.connect(Alice).setVenusInfo(users[0].address,users[1].address)).to.revertedWith("only admin
can");
        });

        // reset
        it("VenusInfo can't be reset", async function() {
            const {proxy,users} = await loadFixture(deployAndBindFixture);
            await expect(proxy.setVenusInfo(users[0].address,users[1].address)).to.revertedWith("addresses
already set");
        });
    });

    // describe("Get/Set Admin or Burn Admin test", function() {
    //     // redundant
    //     it("get  and burn admin test", async function() {
    //         const {Owner,proxy} = await loadFixture(deployAndBindFixture);
    //         expect(await proxy.getAdmin()).to.equal(Owner.address);
    //     });

    //     it("only admin can burn admin", async function () {
    //         const {Alice,proxy} = await loadFixture(deployAndBindFixture);
    //         await expect(proxy.connect(Alice).burnAdmin()).to.revertedWith("only admin can");
    //     });

    //     // AdminTransfered => AdminTransferred
    //     it("Burn admin should emit event and change state", async function() {
    //         const {Owner,proxy} = await loadFixture(deployAndBindFixture);
    //         await expect(proxy.burnAdmin()).to.emit(
    //             proxy,"AdminTransfered"
    //         ).withArgs(Owner.address,zero_address);
    //     });

    //     // redundant
    //     it("Set only admin", async function() {
    //         const {Alice,proxy} = await loadFixture(deployAndBindFixture);
    //         await expect(proxy.connect(Alice).setNewAdmin(Alice.address)).to.revertedWith("only admin
can");
    //     });

    //     it("Set to zero address should be failed", async function() {
    //         const {proxy} = await loadFixture(deployAndBindFixture);
    //         await expect(proxy.setNewAdmin(zero_address)).to.revertedWith("new owner is the zero address");
    //     });

    //     // AdminTransfered => AdminTransferred
    //     it("Set new admin should emit event and change state", async function() {
```

```
    //         const {Owner,proxy,Alice} = await loadFixture(deployAndBindFixture);
    //         await expect(proxy.setNewAdmin(Alice.address)).to.emit(
    //             proxy,"AdminTransfered"
    //         ).withArgs(Owner.address,Alice.address);
    //     });
    // });

    describe("setAccessControl test", function() {
        it("Only admin can set", async function() {
            const {Alice,proxy} = await loadFixture(deployAndBindFixture);
            await expect(proxy.connect(Alice).setAccessControl(zero_address)).to.revertedWith("only admin
can");
        });
        it("Can not set to zero address", async function() {
            const {proxy} = await loadFixture(deployAndBindFixture);
            await expect(proxy.setAccessControl(zero_address)).to.revertedWith("invalid acess control manager
address");
        });
        it("setAccessControl should change state and emit event", async function() {
            const {proxy,access_controller,Bob} = await loadFixture(deployAndBindFixture);
            await expect(proxy.setAccessControl(Bob.address)).to.emit(
                proxy,"NewAccessControlManager"
            ).withArgs(access_controller.address,Bob.address);
            expect(await proxy.accessControlManager()).to.equal(Bob.address);
        });
    });

    describe("updatePendingRewards test", function() {
        it("updatePending should change records", async function() {
            let value = ethers.constants.WeiPerEther;
            const {xvs,proxy} = await loadFixture(deployAndBindFixture);
            await xvs.transfer(proxy.address, value);
            await proxy.updatePendingRewards();
            expect(await proxy.xvsBalance()).to.equal(value);
            expect(await proxy.pendingRewards()).to.equal(value);
        });
    });

    //
    describe("Deposit test", function () {
        it("Deposit should be failed while paused", async function() {
            const {Alice,proxy} = await loadFixture(deployAndBindFixture);
            await proxy.pause();
            await expect(proxy.connect(Alice).deposit(10000)).to.revertedWith("Vault is paused");
        });

        it("Deposit should change state and emit event", async function() {
            const {xvs,vai,Alice,Bob,proxy} = await loadFixture(deployAndBindFixture);
            // prepare
            await xvs.transfer(proxy.address,ethers.constants.WeiPerEther);
            await proxy.updatePendingRewards();
            await vai.transfer(Alice.address,ethers.utils.parseEther("10000"));
            await vai.transfer(Bob.address,ethers.utils.parseEther("10000"));
            await vai.connect(Alice).approve(proxy.address,ethers.constants.MaxInt256);
            await vai.connect(Bob).approve(proxy.address,ethers.constants.MaxInt256);
            // Alice deposit
            let deposit_amount = ethers.utils.parseEther("100");
            await expect(proxy.connect(Alice).deposit(deposit_amount)).to.emit(
                proxy,"Deposit"
            ).withArgs(Alice.address,deposit_amount);
            let alice_info = await proxy.userInfo(Alice.address);
            expect(alice_info.amount).to.equal(deposit_amount);
            expect(alice_info.rewardDebt).to.equal(0);
            expect(await proxy.accXVSPerShare()).to.equal(0);
            // Bob deposit
            await proxy.connect(Bob).deposit(deposit_amount);
            expect(await proxy.accXVSPerShare()).to.equal(ethers.constants.WeiPerEther.div(100));
            let bob_info = await proxy.userInfo(Bob.address);
            expect(bob_info.amount).to.equal(deposit_amount);
            expect(bob_info.rewardDebt).to.equal(ethers.constants.WeiPerEther);
            // check pending
            expect(await proxy.pendingXVS(Alice.address)).to.equal(ethers.constants.WeiPerEther);
            expect(await proxy.pendingXVS(Bob.address)).to.equal(0);
```

```
256
257             // bob deposit again
258             await proxy.connect(Bob).deposit(deposit_amount);
259             bob_info = await proxy.userInfo(Bob.address);
260             expect(bob_info.amount).to.equal(deposit_amount.mul(2));
261             expect(bob_info.rewardDebt).to.equal(ethers.constants.WeiPerEther.mul(2));
262
263             // bob claim ;
264             await proxy.functions["claim(address)"](Bob.address);
265             bob_info = await proxy.userInfo(Bob.address);
266             expect(bob_info.amount).to.equal(deposit_amount.mul(2));
267             expect(bob_info.rewardDebt).to.equal(ethers.constants.WeiPerEther.mul(2));
268
269             // rewards again;
270             await xvs.transfer(proxy.address,ethers.constants.WeiPerEther);
271             await proxy.updatePendingRewards();
272
273             let share = ethers.constants.WeiPerEther.div(100).add(ethers.constants.WeiPerEther.div(300));
274             let pending = share.mul(100);
275
276             // Alice withdraw
277             await expect(proxy.connect(Alice).withdraw(0)).to.emit(
278                 xvs,"Transfer"
279             ).withArgs(proxy.address,Alice.address,pending);
280             expect(await proxy.accXVSPerShare()).to.equal(share);
281
282             alice_info = await proxy.userInfo(Alice.address);
283             expect(alice_info.amount).to.equal(deposit_amount);
284             expect(alice_info.rewardDebt).to.equal(pending);
285
286             bob_info = await proxy.userInfo(Bob.address);
287             expect(bob_info.amount).to.equal(deposit_amount.mul(2));
288             expect(bob_info.rewardDebt).to.equal(ethers.constants.WeiPerEther.mul(2));
289
290             // withdraw should transfer all the xvs
291             let balance = await xvs.balanceOf(proxy.address);
292             await expect(proxy.connect(Bob).functions["claim()"]()).to.emit(
293                 xvs,"Transfer"
294             ).withArgs(proxy.address,Bob.address,balance.sub(100));
295         });
296     });
297
298
299     describe("Bug Demo", function () {
300         it("Rewards can convert to xvsBalance",async function () {
301             const {xvs,vai,Alice,Bob,proxy} = await loadFixture(deployAndBindFixture);
302             // prepare
303             await xvs.transfer(proxy.address,ethers.constants.WeiPerEther);
304             await proxy.updatePendingRewards();
305             await vai.transfer(Alice.address,ethers.utils.parseEther("10000"));
306             await vai.transfer(Bob.address,ethers.utils.parseEther("10000"));
307             await vai.connect(Alice).approve(proxy.address,ethers.constants.MaxInt256);
308             await vai.connect(Bob).approve(proxy.address,ethers.constants.MaxInt256);
309             // Alice deposit
310             let deposit_amount = ethers.utils.parseEther("100");
311             await expect(proxy.connect(Alice).deposit(deposit_amount)).to.emit(
312                 proxy,"Deposit"
313             ).withArgs(Alice.address,deposit_amount);
314             // Bob deposit before updatePendingRewards()
315             await proxy.connect(Bob).deposit(deposit_amount);
316             // transfer xvs to proxy;
317             await xvs.transfer(proxy.address,ethers.constants.WeiPerEther);
318             // Alice withdraw between transfer xvs and updatePendingRewards
319             await proxy.connect(Alice).withdraw(deposit_amount);
320             // updatePendingRewards
321             await proxy.updatePendingRewards();
322
323             let balance = await xvs.balanceOf(proxy.address);
324             expect(balance).to.equal(ethers.constants.WeiPerEther);
325             //
326             let alice_info = await proxy.userInfo(Alice.address);
327             expect(alice_info.amount).to.equal(0);
328             expect(alice_info.rewardDebt).to.equal(0);
```

```
            // dead coin
            let xvsBalance = await proxy.xvsBalance();
            expect(xvsBalance).to.equal(balance);

            let pending = await proxy.pendingRewards();
            expect(pending).to.equal(0);
            // Bob has no rewards
            let pending_bob = await proxy.pendingXVS(Bob.address);
            expect(pending_bob).to.equal(0);
        });
    });

    // describe("Recover owner demo", function() {
    //     it("Recover owner after burn owner", async function() {
    //         const {Alice,proxy,Bob} = await loadFixture(deployAndBindFixture);
    //         const VAIVaultProxy = await ethers.getContractFactory("VAIVaultProxy");
    //         let instance = VAIVaultProxy.attach(proxy.address);
    //         await instance._setPendingAdmin(Alice.address);
    //         expect(await proxy.pendingAdmin()).to.equal(Alice.address);
    //         await proxy.burnAdmin();
    //         expect(await proxy.admin()).to.equal(zero_address);
    //         await instance.connect(Alice)._acceptAdmin();
    //         expect(await proxy.admin()).to.equal(Alice.address);
    //         expect(await proxy.pendingAdmin()).to.equal(zero_address);
    //     });

    //     it("Recover owner after setNewAdmin", async function() {
    //         const {Alice,proxy,Bob} = await loadFixture(deployAndBindFixture);
    //         const VAIVaultProxy = await ethers.getContractFactory("VAIVaultProxy");
    //         let instance = VAIVaultProxy.attach(proxy.address);
    //         await instance._setPendingAdmin(Alice.address);
    //         expect(await proxy.pendingAdmin()).to.equal(Alice.address);
    //         await proxy.setNewAdmin(Bob.address);
    //         expect(await proxy.admin()).to.equal(Bob.address);
    //         await instance.connect(Alice)._acceptAdmin();
    //         expect(await proxy.admin()).to.equal(Alice.address);
    //         expect(await proxy.pendingAdmin()).to.equal(zero_address);
    //     });
    // });
});

```

2. VRTVault.js

```
const {
    time,
    mine,
    loadFixture,
} = require("@nomicfoundation/hardhat-network-helpers");
const { anyValue } = require("@nomicfoundation/hardhat-chai-matchers/withArgs");
const { expect } = require("chai");
const { ethers } = require("hardhat");


const zero_address = ethers.constants.AddressZero;

describe("VRTVault Unit Test", function () {
    async function deployAndInitializeFixture() {
        // get users;
        const [Owner, Alice,Bob,...users] = await ethers.getSigners();
        // Deploy access_controller
        const MockAccessControlManagerV5 = await ethers.getContractFactory("MockAccessControlManagerV5");
        const access_controller = await MockAccessControlManagerV5.deploy();
        // deploy VRTVault;
        const VRTVault = await ethers.getContractFactory("VRTVault");
        const vault = await VRTVault.deploy();
        // deploy token
        const MockERC20 = await ethers.getContractFactory("MockERC20");
        const vrt = await MockERC20.deploy("VRT","VRT",ethers.utils.parseEther("100000000"));
        // deploy proxy
```

```
        const VRTVaultProxy = await ethers.getContractFactory("VRTVaultProxy");
        let proxy = await VRTVaultProxy.deploy(
            vault.address,
            vrt.address,
            ethers.utils.parseEther("0.005")
        );
        proxy = VRTVault.attach(proxy.address);
        await proxy.setAccessControl(access_controller.address);
        return {
            Owner,Alice,Bob,users,proxy,vault,vrt
        };
    }

    describe("initial state unit test", function() {
        it("Check all state vars", async function() {
            const {Owner,proxy,vault,vrt} = await loadFixture(deployAndInitializeFixture);
            const VRTVaultProxy = await ethers.getContractFactory("VRTVaultProxy");
            const instance = VRTVaultProxy.attach(proxy.address);
            // instance
            expect(await instance.admin()).to.equal(Owner.address);
            expect(await instance.implementation()).to.equal(vault.address);
            expect(await instance.pendingAdmin()).to.equal(zero_address);
            expect(await instance.pendingImplementation()).to.equal(zero_address);
            // proxy
            expect(await proxy._notEntered()).to.equal(true);
            expect(await proxy.vaultPaused()).to.equal(false);
            expect(await proxy.vrt()).to.equal(vrt.address);
            expect(await proxy.interestRatePerBlock()).to.equal(ethers.utils.parseEther("0.005"));
            expect(await proxy.lastAccruingBlock()).to.equal(0);
        });
    });

    describe("_setImplementation unit test", function() {
        it("_setImplementation only by admin", async function() {
            const {Alice,proxy,users} = await loadFixture(deployAndInitializeFixture);
            const VRTVaultProxy = await ethers.getContractFactory("VRTVaultProxy");
            const instance = VRTVaultProxy.attach(proxy.address);
            await
expect(instance.connect(Alice)._setImplementation(users[0].address)).to.revertedWith("VRTVaultProxy::_setImple
mentation: admin only");
        });

        it("implementation_ can't be zero address", async function() {
            const {proxy} = await loadFixture(deployAndInitializeFixture);
            const VRTVaultProxy = await ethers.getContractFactory("VRTVaultProxy");
            const instance = VRTVaultProxy.attach(proxy.address);
            await
expect(instance._setImplementation(zero_address)).to.revertedWith("VRTVaultProxy::_setImplementation: invalid
implementation address");
        });

        it("_setImplementation should change state and emit event", async function() {
            const {proxy,vault,users} = await loadFixture(deployAndInitializeFixture);
            const VRTVaultProxy = await ethers.getContractFactory("VRTVaultProxy");
            const instance = VRTVaultProxy.attach(proxy.address);
            expect(await instance._setImplementation(users[0].address)).to.emit(
                instance,"NewImplementation"
            ).withArgs(vault.address,users[0].address);
            expect(await instance.implementation()).to.equal(users[0].address);
        });
    });

    describe("initialize unit test", function(){
        it("initialize twice should be failed", async function() {
            const {proxy,vrt} = await loadFixture(deployAndInitializeFixture);
            await
expect(proxy.initialize(vrt.address,ethers.utils.parseEther("0.005"))).to.revertedWith("Vault may only be
initialized once");
        });
    });

    describe("Pause and resume test", function() {
        it("pause  and resume contract test", async function() {
```

```
            const {Owner,Alice,proxy} = await loadFixture(deployAndInitializeFixture);
            // only Owner
            await expect(proxy.connect(Alice).pause()).to.revertedWith("Unauthorized");
            await expect(proxy.connect(Alice).resume()).to.revertedWith("Unauthorized");
            // pause emit event
            await expect(proxy.pause()).to.emit(
                proxy,"VaultPaused"
            ).withArgs(Owner.address);
            expect(await proxy.vaultPaused()).to.equal(true);
            // pause twice should be failed
            await expect(proxy.pause()).to.revertedWith("Vault is already paused");

            // resume emit event
            await expect(proxy.resume()).to.emit(
                proxy,"VaultResumed"
            ).withArgs(Owner.address);
            expect(await proxy.vaultPaused()).to.equal(false);
            // resume twice should be failed
            await expect(proxy.resume()).to.revertedWith("Vault is not paused");
        });
    });

    describe("withdrawBep20 unit test", function() {
        it("only admin can withdraw", async function() {
            const {Owner,Alice,proxy} = await loadFixture(deployAndInitializeFixture);
            await expect(proxy.connect(Alice).withdrawBep20(Owner.address,Alice.address,100)).to.revertedWith(
                "Unauthorized"
            );
        });

        // risk
        it("Can withdraw vrt", async function() {
            const {vrt,Alice,proxy} = await loadFixture(deployAndInitializeFixture);
            await vrt.transfer(proxy.address,10000);
            expect(await vrt.balanceOf(proxy.address)).to.equal(10000);
            expect(await vrt.balanceOf(Alice.address)).to.equal(0);
            await expect(proxy.withdrawBep20(vrt.address,Alice.address,10000)).to.emit(
                proxy,"WithdrawToken"
            ).withArgs(vrt.address,Alice.address,10000);
            expect(await vrt.balanceOf(proxy.address)).to.equal(0);
            expect(await vrt.balanceOf(Alice.address)).to.equal(10000);
        });
    });

    describe("setLastAccruingBlock unit test", function() {
        it("only admin can set", async function() {
            const {Alice,proxy} = await loadFixture(deployAndInitializeFixture);
            await expect(proxy.connect(Alice).setLastAccruingBlock(100)).to.revertedWith("Unauthorized");
        });

        it("Set should change state and emit event", async function() {
            const {proxy} = await loadFixture(deployAndInitializeFixture);
            let block = await time.latestBlock();
            await expect(proxy.setLastAccruingBlock(block + 10)).to.emit(
                proxy,"LastAccruingBlockChanged"
            ).withArgs(0,block + 10);
            expect(await proxy.lastAccruingBlock()).to.equal(block + 10);

            await expect(proxy.setLastAccruingBlock(block + 8)).to.emit(
                proxy,"LastAccruingBlockChanged"
            ).withArgs(block + 10,block + 8);
            expect(await proxy.lastAccruingBlock()).to.equal(block + 8);
        });

        it("Invalid block setting should be failed", async function(){
            const {proxy} = await loadFixture(deployAndInitializeFixture);
            let block = await time.latestBlock();
            await expect(proxy.setLastAccruingBlock(block + 10)).to.emit(
                proxy,"LastAccruingBlockChanged"
            ).withArgs(0,block + 10);
            expect(await proxy.lastAccruingBlock()).to.equal(block + 10);

            await expect(proxy.setLastAccruingBlock(block + 8)).to.emit(
```

```
                proxy,"LastAccruingBlockChanged"
            ).withArgs(block + 10,block + 8);
            expect(await proxy.lastAccruingBlock()).to.equal(block + 8);

            await expect(proxy.setLastAccruingBlock(block  - 1)).to.rejectedWith(
                "Invalid _lastAccruingBlock interest have been accumulated"
            );
        });
    });

    describe("Deposit | Claim | Withdraw unit test", function() {
        it("Deposit | Claim | Withdraw  ", async function(){
            // prepare
            const {Alice,vrt,proxy} = await loadFixture(deployAndInitializeFixture);
            await vrt.transfer(Alice.address,10000);
            await vrt.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
            let block = await time.latestBlock();
            await proxy.setLastAccruingBlock(block + 10);
            // first deposit
            await expect(proxy.connect(Alice).deposit(1000)).to.emit(
                proxy,"Deposit"
            ).withArgs(Alice.address,1000);
            let info = await proxy.userInfo(Alice.address);
            expect(info.userAddress).to.equal(Alice.address);
            expect(info.accrualStartBlockNumber).to.equal(block + 2);
            expect(info.totalPrincipalAmount).to.equal(1000);
            expect(info.lastWithdrawnBlockNumber).to.equal(0); // unused

            // deposit again;
            await expect(proxy.connect(Alice).deposit(1000)).to.emit(
                proxy,"Claim"
            ).withArgs(Alice.address,1000 * 0.005);

            info = await proxy.userInfo(Alice.address);
            expect(info.userAddress).to.equal(Alice.address);
            expect(info.accrualStartBlockNumber).to.equal(block + 3);
            expect(info.totalPrincipalAmount).to.equal(2000);
            expect(info.lastWithdrawnBlockNumber).to.equal(0);

            await mine(5);
            expect(await proxy.getAccruedInterest(Alice.address)).to.equal(
                2000 * 5 * 0.005
            );

            // claim
            await expect(proxy.functions["claim(address)"](Alice.address)).to.emit(
                proxy,"Claim"
            ).withArgs(Alice.address,2000 * 6 * 0.005);

            // withdraw should be failed
            await expect(proxy.connect(Alice).withdraw()).to.revertedWith("Failed to transfer VRT,
Insufficient VRT in Vault.");

            let start_block = block + 2;
            let end_block = block + 10;
            let deposit_block = await time.latestBlock() + 1;

            let interest = 0;
            if (deposit_block >= end_block) {
                interest = 1000 * 0.005 + 2000 * 7 * 0.005;
            } else {
                interest = 1000 * 0.005 + (deposit_block - start_block -1) * 2000 * 0.005;
            }
            await vrt.transfer(proxy.address,interest);
            // withdraw should be successful
            await expect(proxy.connect(Alice).withdraw()).to.emit(
                proxy,"Withdraw"
            ).withArgs(Alice.address,anyValue,2000,anyValue);
            expect(await vrt.balanceOf(proxy.address)).to.equal(0);
        });
    });

});
```

```
239
```

3. XVSStore.js

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

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

const zero_address = ethers.constants.AddressZero;

describe("XVSStore Unit Test", function () {
    async function deployFixture() {
        // get users;
        const [Owner, Alice,Bob,...users] = await ethers.getSigners();
        // Deploy XVSStore
        const XVSStore = await ethers.getContractFactory("XVSStore");
        const store = await XVSStore.deploy();
        // token
        const MockERC20 = await ethers.getContractFactory("MockERC20");
        const xvs = await MockERC20.deploy("XVS","XVS",ethers.utils.parseEther("100000000"));
        const vai = await MockERC20.deploy("VAI","VAI",ethers.utils.parseEther("100000000"));
        return {
            store,Owner,Alice,Bob,xvs,vai,users
        };
    }

    describe("Initial state check", function() {
        it("Check all state after deploy", async function(){
            const {store,Owner} = await loadFixture(deployFixture);
            expect(await store.admin()).to.equal(Owner.address);
            expect(await store.pendingAdmin()).to.equal(zero_address);
            expect(await store.owner()).to.equal(zero_address);
        });
    });

    describe("setPendingAdmin unit test", function() {
        it("only admin can call it", async function() {
            const {store,Alice} = await loadFixture(deployFixture);
            await expect(store.connect(Alice).setPendingAdmin(Alice.address)).to.revertedWith("only admin
can");
        });

        it("setPendingAdmin should change state and emit event", async function() {
            const {store,Alice,Bob} = await loadFixture(deployFixture);
            await expect(store.setPendingAdmin(Alice.address)).to.emit(
                store,"NewPendingAdmin"
            ).withArgs(zero_address,Alice.address);
            expect(await store.pendingAdmin()).to.equal(Alice.address);
            // can set twice
            await expect(store.setPendingAdmin(Bob.address)).to.emit(
                store,"NewPendingAdmin"
            ).withArgs(Alice.address,Bob.address);
            expect(await store.pendingAdmin()).to.equal(Bob.address);
        });
    });

    describe("acceptAdmin unit test", function() {
        it("only pendingAdmin can call it", async function() {
            const {store,Alice,Bob} = await loadFixture(deployFixture);
            await store.setPendingAdmin(Alice.address);
            await expect(store.connect(Bob).acceptAdmin()).to.revertedWith("only pending admin");
        });

        it("acceptAdmin should change state and emit event", async function() {
            const {store,Alice,Owner} = await loadFixture(deployFixture);
            await store.setPendingAdmin(Alice.address);
            await expect(store.connect(Alice).acceptAdmin()).to.emit(
                store,"AdminTransferred"
```

```
            ).withArgs(Owner.address,Alice.address);
            expect(await store.admin()).to.equal(Alice.address);
            expect(await store.pendingAdmin()).to.equal(zero_address);
        });
    });

    describe("setNewOwner", function() {
        it("only admin can call it", async function(){
            const {store,Alice} = await loadFixture(deployFixture);
            await expect(store.connect(Alice).setNewOwner(Alice.address)).to.revertedWith("only admin can");
        });
        it("set to zero address should be failed", async function() {
            const {store} = await loadFixture(deployFixture);
            await expect(store.setNewOwner(zero_address)).to.revertedWith("new owner is the zero address");
        });

        it("set should change state and emit event", async function() {
            const {store,Alice,Bob} = await loadFixture(deployFixture);
            await expect(store.setNewOwner(Alice.address)).to.emit(
                store,"OwnerTransferred"
            ).withArgs(zero_address,Alice.address);
            expect(await store.owner()).to.equal(Alice.address);
            // owner can't call it
            await expect(store.connect(Alice).setNewOwner(Bob.address)).to.revertedWith("only admin can");
            // set again
            await expect(store.setNewOwner(Bob.address)).to.emit(
                store,"OwnerTransferred"
            ).withArgs(Alice.address,Bob.address);
            expect(await store.owner()).to.equal(Bob.address);
        });
    });

    describe("setRewardToken unit test",function() {
        it("Only owner or admin can call", async function() {
            const {store,Alice,Bob,xvs} = await loadFixture(deployFixture);
            await store.setNewOwner(Alice.address);
            await expect(store.connect(Bob).setRewardToken(xvs.address,true)).to.revertedWith("only admin or
owner can");

            await store.connect(Alice).setRewardToken(xvs.address,true);
            expect(await store.rewardTokens(xvs.address)).to.equal(true);

            await store.setRewardToken(xvs.address,false);
            expect(await store.rewardTokens(xvs.address)).to.equal(false);
        });
    });

    describe("emergencyRewardWithdraw unit test", function() {
        it("only owner can call it", async function() {
            const {store,Owner} = await loadFixture(deployFixture);
            await expect(store.emergencyRewardWithdraw(Owner.address,100)).to.rejectedWith("only owner can");
        });

        it("call it will transfer token", async function() {
            const {store,Alice,xvs} = await loadFixture(deployFixture);
            await store.setNewOwner(Alice.address);
            await xvs.transfer(store.address,10000);
            expect(await xvs.balanceOf(store.address)).to.equal(10000);
            expect(await xvs.balanceOf(Alice.address)).to.equal(0);
            await expect(store.connect(Alice).emergencyRewardWithdraw(xvs.address,1000)).to.emit(
                xvs,"Transfer"
            ).withArgs(store.address,Alice.address,1000);

            expect(await xvs.balanceOf(store.address)).to.equal(9000);
            expect(await xvs.balanceOf(Alice.address)).to.equal(1000);
        });
    });

    describe("safeRewardTransfer unit test", function() {
        it("only owner can call it", async function() {
            const {store,Owner,xvs} = await loadFixture(deployFixture);
            await expect(store.safeRewardTransfer(xvs.address,Owner.address,100)).to.rejectedWith("only owner
can");
```

```
        });

        it("call should be failed while not set reward token", async function() {
            const {store,Owner,xvs} = await loadFixture(deployFixture);
            await store.setNewOwner(Owner.address);
            await expect(store.safeRewardTransfer(xvs.address,Owner.address,100)).to.rejectedWith("only reward
token can");
        });

        it("Transfer rewards beyond balance test", async function() {
            const {store,Owner,xvs,Alice} = await loadFixture(deployFixture);
            await store.setNewOwner(Owner.address);
            await store.setRewardToken(xvs.address,true);
            await xvs.transfer(store.address,10000);
            await expect(store.safeRewardTransfer(xvs.address,Alice.address,200000)).to.emit(
                xvs,"Transfer"
            ).withArgs(store.address,Alice.address,10000);
            expect(await xvs.balanceOf(store.address)).to.equal(0);
            expect(await xvs.balanceOf(Alice.address)).to.equal(10000);
        });

        it("Transfer rewards not beyond balance test", async function() {
            const {store,Owner,xvs,Alice} = await loadFixture(deployFixture);
            await store.setNewOwner(Owner.address);
            await store.setRewardToken(xvs.address,true);
            await xvs.transfer(store.address,10000);
            await expect(store.safeRewardTransfer(xvs.address,Alice.address,2000)).to.emit(
                xvs,"Transfer"
            ).withArgs(store.address,Alice.address,2000);
            expect(await xvs.balanceOf(store.address)).to.equal(8000);
            expect(await xvs.balanceOf(Alice.address)).to.equal(2000);
        });
    });
});

```

4. XVSVault.js

```
const {
    time,
    loadFixture,
} = require("@nomicfoundation/hardhat-network-helpers");
const { anyValue } = require("@nomicfoundation/hardhat-chai-matchers/withArgs");
const { expect } = require("chai");
const { ethers } = require("hardhat");
const { describe } = require("node:test");

const zero_address = ethers.constants.AddressZero;
const GAIN = ethers.utils.parseUnits("1.0",12);

describe("XVSVault Unit Test", function () {
    async function deployAndBindFixture() {
        // get users;
        const [Owner, Alice,Bob,...users] = await ethers.getSigners();
        // Deploy access_controller
        const MockAccessControlManagerV5 = await ethers.getContractFactory("MockAccessControlManagerV5");
        const access_controller = await MockAccessControlManagerV5.deploy();
        // deploy XVSVault;
        const XVSVault = await ethers.getContractFactory("XVSVault");
        const vault = await XVSVault.deploy();
        // deploy proxy
        const XVSVaultProxy = await ethers.getContractFactory("XVSVaultProxy");
        let proxy = await XVSVaultProxy.deploy();
        // deploy XVSStore
        const XVSStore = await ethers.getContractFactory("XVSStore");
        let store = await XVSStore.deploy();
        await store.setNewOwner(proxy.address);
        // binding
        await proxy._setPendingImplementation(vault.address);
        await vault._become(proxy.address);
        proxy = XVSVault.attach(proxy.address);
```

```
        await proxy.setAccessControl(access_controller.address);
        // deploy token
        const MockERC20 = await ethers.getContractFactory("MockERC20");
        const xvs = await MockERC20.deploy("XVS","XVS",ethers.utils.parseEther("100000000"));
        const vai = await MockERC20.deploy("VAI","VAI",ethers.utils.parseEther("100000000"));
        const token_a = await MockERC20.deploy("TokenA","TAT",ethers.utils.parseEther("100000000"));
        const token_b = await MockERC20.deploy("TokenB","TBT",ethers.utils.parseEther("100000000"));
        await proxy.setXvsStore(xvs.address,store.address);
        // return
        return {
            Owner,Alice,Bob,users,proxy,vault,xvs,vai,access_controller,store,token_a,token_b
        };
    }

    describe("Initial State check", function() {
        it("Check all state vars", async function() {
            const {proxy,vault,Owner,xvs,store} = await loadFixture(deployAndBindFixture);
            expect(await proxy.admin()).to.equal(Owner.address);
            expect(await proxy.pendingAdmin()).to.equal(zero_address);
            expect(await proxy.implementation()).to.equal(vault.address);
            expect(await proxy.pendingXVSVaultImplementation()).to.equal(zero_address);
            expect(await proxy.xvsStore()).to.equal(store.address);
            expect(await proxy.xvsAddress()).to.equal(xvs.address);
            expect(await proxy.vaultPaused()).to.equal(false);
        });
    });

    describe("Pause and resume unit test", function() {
        it("pause  and resume contract test", async function() {
            const {Owner,Alice,proxy} = await loadFixture(deployAndBindFixture);
            // only Owner
            await expect(proxy.connect(Alice).pause()).to.revertedWith("Unauthorized");
            await expect(proxy.connect(Alice).resume()).to.revertedWith("Unauthorized");
            // pause emit event
            await expect(proxy.pause()).to.emit(
                proxy,"VaultPaused"
            ).withArgs(Owner.address);
            expect(await proxy.vaultPaused()).to.equal(true);
            // pause twice should be failed
            await expect(proxy.pause()).to.revertedWith("Vault is already paused");

            // resume emit event
            await expect(proxy.resume()).to.emit(
                proxy,"VaultResumed"
            ).withArgs(Owner.address);
            expect(await proxy.vaultPaused()).to.equal(false);
            // resume twice should be failed
            await expect(proxy.resume()).to.revertedWith("Vault is not paused");
        });
    });

    describe("Change admin and implement test", function() {
        it("only admin can change admin or implement", async function() {
            const {Alice,proxy,users} = await loadFixture(deployAndBindFixture);
            const XVSVaultProxy = await ethers.getContractFactory("XVSVaultProxy");
            let instance = XVSVaultProxy.attach(proxy.address);
            await expect(instance.connect(Alice)._setPendingAdmin(users[0].address)).to.emit(
                instance,"Failure"
            ).withArgs(1,2,0);
            expect(await proxy.pendingAdmin()).to.equal(zero_address);

            await expect(instance.connect(Alice)._setPendingImplementation(users[0].address)).to.emit(
                instance,"Failure"
            ).withArgs(1,3,0);
            expect(await proxy.pendingXVSVaultImplementation()).to.equal(zero_address);
        });

        it("only pending can accept", async function() {
            const {Owner,Alice,Bob,proxy,vault} = await loadFixture(deployAndBindFixture);
            const XVSVaultProxy = await ethers.getContractFactory("XVSVaultProxy");
            let instance = XVSVaultProxy.attach(proxy.address);

            // set pending
```

```
            await expect(instance._setPendingAdmin(Alice.address)).to.emit(
                instance,"NewPendingAdmin"
            ).withArgs(zero_address,Alice.address);
            expect(await proxy.pendingAdmin()).to.equal(Alice.address);
            await expect(instance._setPendingImplementation(Bob.address)).to.emit(
                instance,"NewPendingImplementation"
            ).withArgs(zero_address,Bob.address);
            expect(await proxy.pendingXVSVaultImplementation()).to.equal(Bob.address);

            // accept without pending
            await expect(instance._acceptAdmin()).to.emit(
                instance,"Failure"
            ).withArgs(1,0,0);
            expect(await proxy.pendingAdmin()).to.equal(Alice.address);
            await expect(instance._acceptImplementation()).to.emit(
                instance,"Failure"
            ).withArgs(1,1,0);
            expect(await proxy.pendingXVSVaultImplementation()).to.equal(Bob.address);

            // accept with pending
            await expect(instance.connect(Alice)._acceptAdmin()).to.emit(
                instance,"NewAdmin"
            ).withArgs(Owner.address,Alice.address);

            await expect(instance.connect(Bob)._acceptImplementation()).to.emit(
                instance,"NewImplementation"
            ).withArgs(vault.address,Bob.address);

            // check final state
            expect(await proxy.admin()).to.equal(Alice.address);
            expect(await proxy.pendingAdmin()).to.equal(zero_address);
            expect(await proxy.implementation()).to.equal(Bob.address);
            expect(await proxy.pendingXVSVaultImplementation()).to.equal(zero_address);
        });
    });

    describe("Add pool unit test", function() {
        it("only user with access_allowed can add pool", async function() {
            const {Alice,token_a,xvs,proxy,vai} = await loadFixture(deployAndBindFixture);
            await
expect(proxy.connect(Alice).add(xvs.address,50,token_a.address,50,300)).to.revertedWith("Unauthorized");
        });

        it("Add pool should change state and emit event", async function() {
            const {token_a,token_b,xvs,vai,proxy,store} = await loadFixture(deployAndBindFixture);
            await expect(proxy.add(xvs.address,50,token_a.address,100,300)).to.emit(
                proxy,"PoolAdded"
            ).withArgs(xvs.address,0,token_a.address,50,100,300);
            let block = await time.latestBlock();
            expect(await store.rewardTokens(xvs.address)).to.equal(true);
            let {token,allocPoint,lastRewardBlock,accRewardPerShare,lockPeriod} = await
proxy.poolInfos(xvs.address,0);
            expect(token).to.equal(token_a.address);
            expect(allocPoint).to.equal(50);
            expect(lastRewardBlock).to.equal(block);
            expect(accRewardPerShare).to.equal(0);
            expect(lockPeriod).to.equal(300);

            expect(await proxy.rewardTokenAmountsPerBlock(xvs.address)).to.equal(100);
            expect(await proxy.totalAllocPoints(xvs.address)).to.equal(50);
            expect(await proxy.poolLength(xvs.address)).to.equal(1);

            // add same should be failed
            await expect(proxy.add(xvs.address,50,token_a.address,100,300)).to.revertedWith("Pool already
added");
            // add token_b
            await expect(proxy.add(xvs.address,50,token_b.address,200,300)).to.emit(
                proxy,"PoolAdded"
            ).withArgs(xvs.address,1,token_b.address,50,200,300);

            block = await time.latestBlock();
            ({token,allocPoint,lastRewardBlock,accRewardPerShare,lockPeriod} = await
proxy.poolInfos(xvs.address,1));
```

```
            expect(token).to.equal(token_b.address);
            expect(allocPoint).to.equal(50);
            expect(lastRewardBlock).to.equal(block);
            expect(accRewardPerShare).to.equal(0);
            expect(lockPeriod).to.equal(300);

            expect(await proxy.rewardTokenAmountsPerBlock(xvs.address)).to.equal(200);
            expect(await proxy.totalAllocPoints(xvs.address)).to.equal(100);
            expect(await proxy.poolLength(xvs.address)).to.equal(2);

            // check token_a
            ({token,allocPoint,lastRewardBlock,accRewardPerShare,lockPeriod} = await
proxy.poolInfos(xvs.address,0));
            expect(token).to.equal(token_a.address);
            expect(allocPoint).to.equal(50);
            expect(lastRewardBlock).to.equal(block);
            expect(accRewardPerShare).to.equal(0);
            expect(lockPeriod).to.equal(300);

            // add another reward token;
            await expect(proxy.add(token_a.address,50,xvs.address,100,300)).to.emit(
                proxy,"PoolAdded"
            ).withArgs(token_a.address,0,xvs.address,50,100,300);

            // add same deposit token
            await expect(proxy.add(vai.address,50,token_a.address,100,300)).to.revertedWith("Token exists in
other pool");
        });
    });

    describe("Reset _allocPoint unit test", function() {
        it("only user with access_allowed can reset pool", async function() {
            const {Alice,xvs,proxy} = await loadFixture(deployAndBindFixture);
            await expect(proxy.connect(Alice).set(xvs.address,50,300)).to.revertedWith("Unauthorized");
        });
        it("only valid pool can be reset", async function() {
            const {xvs,proxy} = await loadFixture(deployAndBindFixture);
            await expect(proxy.set(xvs.address,50,300)).to.revertedWith("vault: pool exists?");
        });

        it("Reset must update pool", async function() {
            const {token_a,token_b,xvs,proxy} = await loadFixture(deployAndBindFixture);
            await expect(proxy.add(xvs.address,50,token_a.address,100,300)).to.emit(
                proxy,"PoolAdded"
            ).withArgs(xvs.address,0,token_a.address,50,100,300);
            await expect(proxy.add(xvs.address,50,token_b.address,100,300)).to.emit(
                proxy,"PoolAdded"
            ).withArgs(xvs.address,1,token_b.address,50,100,300);
            // reset
            expect(await proxy.totalAllocPoints(xvs.address)).to.equal(100);
            await expect(proxy.set(xvs.address,0,100)).to.emit(
                proxy,"PoolUpdated"
            ).withArgs(xvs.address,0,50,100);
            // check state
            let block = await time.latestBlock();
            let {token,allocPoint,lastRewardBlock,accRewardPerShare,lockPeriod} = await
proxy.poolInfos(xvs.address,0);
            expect(token).to.equal(token_a.address);
            expect(allocPoint).to.equal(100);
            expect(lastRewardBlock).to.equal(block);
            expect(accRewardPerShare).to.equal(0);
            expect(lockPeriod).to.equal(300);
            expect(await proxy.totalAllocPoints(xvs.address)).to.equal(150);
        });
    });

    describe("setRewardAmountPerBlock unit test", function() {
        it("only user with access_allowed can reset reward", async function() {
            const {Alice,xvs,proxy} = await loadFixture(deployAndBindFixture);
            await
expect(proxy.connect(Alice).setRewardAmountPerBlock(xvs.address,300)).to.revertedWith("Unauthorized");
        });

```

```
245         it("set should update pool and change state", async function() {
246             const {token_a,xvs,proxy} = await loadFixture(deployAndBindFixture);
247             await expect(proxy.add(xvs.address,50,token_a.address,100,300)).to.emit(
248                 proxy,"PoolAdded"
249             ).withArgs(xvs.address,0,token_a.address,50,100,300);
250 
251             expect(await proxy.rewardTokenAmountsPerBlock(xvs.address)).to.equal(100);
252             await expect(proxy.setRewardAmountPerBlock(xvs.address,200)).to.emit(
253                 proxy,"RewardAmountUpdated"
254             ).withArgs(xvs.address,100,200);
255             // check state after set
256             expect(await proxy.rewardTokenAmountsPerBlock(xvs.address)).to.equal(200);
257             let block = await time.latestBlock();
258             let info = await proxy.poolInfos(xvs.address,0);
259             expect(info.lastRewardBlock).to.equal(block);
260         });
261     });
262 
263     describe("setWithdrawalLockingPeriod unit test", function() {
264         it("only user with access_allowed can reset locking period", async function() {
265             const {Alice,xvs,proxy} = await loadFixture(deployAndBindFixture);
266             await
expect(proxy.connect(Alice).setWithdrawalLockingPeriod(xvs.address,0,300)).to.revertedWith("Unauthorized");
267         });
268 
269         it("Pool must be valid", async function() {
270             const {xvs,proxy} = await loadFixture(deployAndBindFixture);
271             await expect(proxy.setWithdrawalLockingPeriod(xvs.address,0,300)).to.revertedWith("vault: pool
exists?");
272         });
273 
274         it("period must be greater than zero", async function() {
275             const {token_a,xvs,proxy} = await loadFixture(deployAndBindFixture);
276             await expect(proxy.add(xvs.address,50,token_a.address,100,300)).to.emit(
277                 proxy,"PoolAdded"
278             ).withArgs(xvs.address,0,token_a.address,50,100,300);
279             await expect(proxy.setWithdrawalLockingPeriod(xvs.address,0,0)).to.revertedWith("Invalid new
locking period");
280         });
281 
282         it("Reset should change state and emit event", async function() {
283             const {token_a,xvs,proxy} = await loadFixture(deployAndBindFixture);
284             await expect(proxy.add(xvs.address,50,token_a.address,100,300)).to.emit(
285                 proxy,"PoolAdded"
286             ).withArgs(xvs.address,0,token_a.address,50,100,300);
287             await expect(proxy.setWithdrawalLockingPeriod(xvs.address,0,400)).to.emit(
288                 proxy,"WithdrawalLockingPeriodUpdated"
289             ).withArgs(xvs.address,0,300,400);
290             let block = await time.latestBlock();
291             let info = await proxy.poolInfos(xvs.address,0);
292             expect(info.lastRewardBlock).to.equal(block -1);
293             expect(info.lockPeriod).to.equal(400);
294         });
295     });
296 
297     describe("delegate unit test", function() {
298         it("Delegate should emit event", async function() {
299             const {Alice,Bob,proxy} = await loadFixture(deployAndBindFixture);
300             await expect(proxy.connect(Alice).delegate(Bob.address)).to.emit(
301                 proxy,"DelegateChangedV2"
302             ).withArgs(Alice.address,zero_address,Bob.address);
303         });
304     });
305 
306     describe("Deposit unit test", function() {
307         it("The pool must be valid", async function() {
308             const {xvs,proxy} = await loadFixture(deployAndBindFixture);
309             await expect(proxy.deposit(xvs.address,1,100)).to.revertedWith("vault: pool exists?");
310         });
311         it("Deposit should update pool and change user's state", async function() {
312             const {token_a,Alice,Bob,xvs,vai,proxy,store} = await loadFixture(deployAndBindFixture);
313             // prepare
314             await token_a.transfer(Alice.address,100000000);
```

```
            await token_a.transfer(Bob.address,100000000);
            await xvs.transfer(Alice.address,100000000);
            await xvs.transfer(Bob.address,100000000);
            await token_a.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
            await xvs.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
            await token_a.connect(Bob).approve(proxy.address,ethers.constants.MaxUint256);
            await xvs.connect(Bob).approve(proxy.address,ethers.constants.MaxUint256);
            await vai.transfer(store.address,ethers.utils.parseEther("100000000"))

            let all_rewards = ethers.utils.parseEther("1");
            // add pool
            await expect(proxy.add(vai.address,50,token_a.address,all_rewards,300)).to.emit(
                proxy,"PoolAdded"
            ).withArgs(vai.address,0,token_a.address,50,all_rewards,300);
            await expect(proxy.add(vai.address,100,xvs.address,all_rewards,300)).to.emit(
                proxy,"PoolAdded"
            ).withArgs(vai.address,1,xvs.address,100,all_rewards,300);
            let block = await time.latestBlock();
            // Alice deposit token_a;
            await expect(proxy.connect(Alice).deposit(vai.address,0,10000)).to.emit(
                proxy,"Deposit"
            ).withArgs(Alice.address,vai.address,0,10000);

            let alice_info = await proxy.getUserInfo(vai.address,0,Alice.address);
            expect(alice_info.amount).equal(10000);
            expect(alice_info.rewardDebt).equal(0);
            expect(alice_info.pendingWithdrawals).equal(0);
            let pool_info = await proxy.poolInfos(vai.address,0);
            expect(pool_info.lastRewardBlock).to.eq(block + 1);
            expect(pool_info.accRewardPerShare).to.eq(0);
            // Bob deposit token_a;
            await expect(proxy.connect(Bob).deposit(vai.address,0,10000)).to.emit(
                proxy,"Deposit"
            ).withArgs(Bob.address,vai.address,0,10000);

            pool_info = await proxy.poolInfos(vai.address,0);
            expect(pool_info.lastRewardBlock).to.eq(block + 2);
            let reward_pershare = all_rewards.mul(50).div(150).mul(GAIN).div(10000);
            expect(pool_info.accRewardPerShare).to.eq(reward_pershare);
            // Alice deposit again;
            let offset = all_rewards.mul(50).div(150).mul(GAIN).div(20000);
            reward_pershare = reward_pershare.add(offset);
            let pending = reward_pershare.mul(10000).div(GAIN);
            // expect(await proxy.pendingReward(vai.address,0,Alice.address)).to.equal(pending);
            // claim or deposit should transfer reward
            await expect(proxy.connect(Alice).claim(Alice.address,vai.address,0)).to.emit(
                vai,"Transfer"
            ).withArgs(store.address,Alice.address,pending);
            await expect(proxy.connect(Alice).deposit(vai.address,0,10000)).to.emit(
                vai,"Transfer"
            ).withArgs(store.address,Alice.address,anyValue);
        });
    });


    describe("Claim unit test", function() {
        it("The pool must be valid", async function() {
            const {Alice,xvs,proxy} = await loadFixture(deployAndBindFixture);
            await expect(proxy.claim(Alice.address,xvs.address,1)).to.revertedWith("vault: pool exists?");
        });
    });



    describe("requestWithdrawal unit test", function() {
        it("The pool must be valid", async function() {
            const {xvs,proxy} = await loadFixture(deployAndBindFixture);
            await expect(proxy.requestWithdrawal(xvs.address,1,100)).to.revertedWith("vault: pool exists?");
        });

        it("requestWithdrawal should emit event and change state", async function() {
            const {token_a,Alice,Bob,xvs,vai,proxy,store} = await loadFixture(deployAndBindFixture);
```

```
            // prepare
            await token_a.transfer(Alice.address,100000000);
            await token_a.transfer(Bob.address,100000000);
            await xvs.transfer(Alice.address,100000000);
            await xvs.transfer(Bob.address,100000000);
            await token_a.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
            await xvs.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
            await token_a.connect(Bob).approve(proxy.address,ethers.constants.MaxUint256);
            await xvs.connect(Bob).approve(proxy.address,ethers.constants.MaxUint256);
            await vai.transfer(store.address,ethers.utils.parseEther("100000000"))
            let all_rewards = ethers.utils.parseEther("1");
            // add pool
            await expect(proxy.add(vai.address,50,token_a.address,all_rewards,300)).to.emit(
                proxy,"PoolAdded"
            ).withArgs(vai.address,0,token_a.address,50,all_rewards,300);
            await proxy.connect(Alice).deposit(vai.address,0,10000);
            time.increase(9);
            await proxy.connect(Alice).deposit(vai.address,0,20000);
            time.increase(9);
            // requestWithdrawal
            let amount = 5000;

            await expect(proxy.connect(Alice).requestWithdrawal(vai.address,0,amount)).to.emit(
                proxy,"RequestedWithdrawal"
            ).withArgs(Alice.address,vai.address,0,5000);
            expect(await proxy.getRequestedAmount(vai.address,0,Alice.address)).to.equal(5000);
            await time.increase(9);
            await  proxy.connect(Alice).requestWithdrawal(vai.address,0,6000);
            await time.increase(9);
            await proxy.setWithdrawalLockingPeriod(vai.address,0,200);
            await proxy.connect(Alice).requestWithdrawal(vai.address,0,3000);
            await time.increase(9);
            await proxy.setWithdrawalLockingPeriod(vai.address,0,250);
            await proxy.connect(Alice).requestWithdrawal(vai.address,0,2000);
            // check sort
            let requests = await proxy.getWithdrawalRequests(vai.address,0,Alice.address);
            expect(requests.length).to.equal(4);
            expect(requests[0].amount).to.equal(6000);
            expect(requests[1].amount).to.equal(5000);
            expect(requests[2].amount).to.equal(2000);
            expect(requests[3].amount).to.equal(3000);


            expect(await proxy.getRequestedAmount(vai.address,0,Alice.address)).to.equal(6000 + 5000 + 3000 +
2000);
            expect(await proxy.pendingWithdrawalsBeforeUpgrade(vai.address,0,Alice.address)).to.equal(0);

            let user_info = await proxy.getUserInfo(vai.address,0,Alice.address);
            expect(user_info.amount).to.equal(30000);
            expect(user_info.pendingWithdrawals).to.equal(16000);
            expect(await proxy.getRequestedAmount(vai.address,0,Alice.address)).to.equal(16000);
            await time.increase(253);
            let can_withdraw = await proxy.getEligibleWithdrawalAmount(vai.address,0,Alice.address);
            expect(can_withdraw).to.equal(5000);

            // exec withdraw
            await expect(proxy.connect(Alice).executeWithdrawal(vai.address,0)).to.emit(
                proxy,"ExecutedWithdrawal"
            ).withArgs(Alice.address,vai.address,0,5000);

            requests = await proxy.getWithdrawalRequests(vai.address,0,Alice.address);
            expect(requests.length).to.equal(2);

            user_info = await proxy.getUserInfo(vai.address,0,Alice.address);
            expect(user_info.amount).to.equal(25000);
            expect(user_info.pendingWithdrawals).to.equal(11000);
        });
    });

    describe("Deposit | withdraw with delegates", function() {
        it("delegate before deposit", async function() {
            const {Alice,Bob,xvs,proxy,store,users} = await loadFixture(deployAndBindFixture);
            // delegate first;
```

```
            await proxy.connect(Alice).delegate(Bob.address);

            await xvs.transfer(Alice.address,10000000);
            await xvs.transfer(store.address,ethers.utils.parseEther("10000"));
            await xvs.connect(Alice).approve(proxy.address,ethers.constants.MaxInt256);
            // add pool
            await proxy.add(xvs.address,100,xvs.address,ethers.utils.parseEther("1.0"),200);

            // deposit
            await expect(proxy.connect(Alice).deposit(xvs.address,0,10000)).to.emit(
                proxy,"DelegateVotesChangedV2"
            ).withArgs(Bob.address,0,10000);
            await expect(proxy.connect(Alice).deposit(xvs.address,0,10000)).to.emit(
                proxy,"DelegateVotesChangedV2"
            ).withArgs(Bob.address,10000,20000);
            let block = await time.latestBlock();
            expect(await proxy.getCurrentVotes(Bob.address)).to.equal(20000);
            expect(await proxy.getCurrentVotes(Alice.address)).to.equal(0);
            expect(await proxy.getPriorVotes(Bob.address,block -1)).to.equal(10000);
            // withdraw
            await proxy.connect(Alice).requestWithdrawal(xvs.address,0,2000);
            expect(await proxy.getPriorVotes(Bob.address,block)).to.equal(20000);
            expect(await proxy.getCurrentVotes(Bob.address)).to.equal(18000);
        });

        it("Delegate after deposit", async function() {
            const {Alice,Bob,xvs,proxy,store,users} = await loadFixture(deployAndBindFixture);
            await xvs.transfer(Alice.address,10000000);
            await xvs.transfer(store.address,ethers.utils.parseEther("10000"));
            await xvs.connect(Alice).approve(proxy.address,ethers.constants.MaxInt256);
            // add pool
            await proxy.add(xvs.address,100,xvs.address,ethers.utils.parseEther("1.0"),200);
            // deposit
            await proxy.connect(Alice).deposit(xvs.address,0,10000);
            expect(await proxy.getCurrentVotes(Alice.address)).to.equal(0);
            // delegate
            await proxy.connect(Alice).delegate(Bob.address);
            expect(await proxy.getCurrentVotes(Alice.address)).to.equal(0);
            expect(await proxy.getCurrentVotes(Bob.address)).to.equal(10000);
            // withdraw
            let block = await time.latestBlock();
            await proxy.connect(Alice).requestWithdrawal(xvs.address,0,2000);
            expect(await proxy.getPriorVotes(Bob.address,block)).to.equal(10000);
            expect(await proxy.getCurrentVotes(Bob.address)).to.equal(8000);
        });
    });
});

```

5. MockUpgrade.js

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

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

describe("MockSlot Upgrade Unit Test", function () {
    async function deployFixture() {
        const CustomMinERC1967Proxy = await ethers.getContractFactory("CustomMinERC1967Proxy");
        const MockSlot = await ethers.getContractFactory("MockSlot");
        const old_impl = await MockSlot.deploy();
        let instance = await CustomMinERC1967Proxy.deploy(old_impl.address);
        instance = MockSlot.attach(instance.address);
        const MockSlotNew = await ethers.getContractFactory("MockSlotNew");
        return {
            instance,MockSlotNew,CustomMinERC1967Proxy
        };
```

```
    }

    it("Upgrade should not change slot", async () => {
        let {instance,MockSlotNew,CustomMinERC1967Proxy} = await loadFixture(deployFixture);
        let param = {
            amount:600,
            lockedUntil:9876543210
        };
        await instance.setRequest(param,5);
        let request = await instance.getRequest(5);
        expect(request[0].amount).to.eq(param.amount);
        expect(request[0].lockedUntil).to.eq(param.lockedUntil);
        // calculate slot
        let start = ethers.utils.solidityKeccak256(["uint256","uint256"],[5,0]);
        let middle = ethers.utils.solidityKeccak256(["bytes32"],[start]);
        let next = ethers.BigNumber.from(middle).add(1);

        let info_0 = await getStorageAt(instance.address,middle);
        let amount  = ethers.BigNumber.from(info_0);
        let info_1 = await getStorageAt(instance.address,next);
        let lockedUntil = ethers.BigNumber.from(info_1);
        expect(amount).to.eq(param.amount);
        expect(lockedUntil).to.eq(param.lockedUntil);

        // upgrade
        let new_impl = await MockSlotNew.deploy();
        instance = CustomMinERC1967Proxy.attach(instance.address);
        await instance.updateTo(new_impl.address);
        instance = MockSlotNew.attach(instance.address);
        // check slot
        info_0 = await getStorageAt(instance.address,middle);
        amount  = ethers.BigNumber.from(info_0);
        info_1 = await getStorageAt(instance.address,next);
        lockedUntil = ethers.BigNumber.from(info_1);
        expect(amount).to.eq(param.amount);
        expect(lockedUntil).to.eq(param.lockedUntil);
        let info_2 = await getStorageAt(instance.address,next.add(1));
        expect(info_2).to.eq(ethers.constants.HashZero);

        // check state
        request = await instance.getRequest(5);
        expect(request[0].amount).to.eq(param.amount);
        expect(request[0].lockedUntil).to.eq(param.lockedUntil);
        expect(request[0].afterUpgrade).to.eq(0);

        // add new struct
        await instance.setRequest({
            amount:600,
            lockedUntil:9876543210,
            afterUpgrade:1
        },5);
        // check old slot
        info_0 = await getStorageAt(instance.address,middle);
        amount  = ethers.BigNumber.from(info_0);
        info_1 = await getStorageAt(instance.address,next);
        lockedUntil = ethers.BigNumber.from(info_1);
        expect(amount).to.eq(param.amount);
        expect(lockedUntil).to.eq(param.lockedUntil);
        // check new slot
        info_0 = await getStorageAt(instance.address,next.add(1));
        amount  = ethers.BigNumber.from(info_0);
        info_1 = await getStorageAt(instance.address,next.add(2));
        lockedUntil = ethers.BigNumber.from("0x" + info_1.substring(34));
        let afterUpgrade = ethers.BigNumber.from(info_1.substring(0,34));
        expect(amount).to.eq(param.amount);
        expect(lockedUntil).to.eq(param.lockedUntil);
        expect(afterUpgrade).to.eq(1);

        // check state
        request = await instance.getRequest(5);
        expect(request[0].amount).to.eq(param.amount);
        expect(request[0].lockedUntil).to.eq(param.lockedUntil);
        expect(request[0].afterUpgrade).to.eq(0);
```

```
        expect(request[1].amount).to.eq(param.amount);
        expect(request[1].lockedUntil).to.eq(param.lockedUntil);
        expect(request[1].afterUpgrade).to.eq(1);

        let length_info = await getStorageAt(instance.address,start);
        let length = ethers.BigNumber.from(length_info);
        expect(length).to.eq(2);
    });
});
```

6. Upgrade.js

```
const {
    time,
    loadFixture,
} = require("@nomicfoundation/hardhat-network-helpers");
const { anyValue } = require("@nomicfoundation/hardhat-chai-matchers/withArgs");
const { expect } = require("chai");
const { ethers } = require("hardhat");

const zero_address = ethers.constants.AddressZero;
const GAIN = ethers.utils.parseUnits("1.0",12);

describe("XVSVault Upgrade Unit Test", function () {
    async function deployAndBindFixture() {
        // get users;
        const [Owner, Alice,Bob,...users] = await ethers.getSigners();
        // Deploy old_impl
        const XVSVaultOld = await ethers.getContractFactory("XVSVaultOld");
        const vault = await XVSVaultOld.deploy();
        // deploy proxy
        const XVSVaultProxy = await ethers.getContractFactory("XVSVaultProxy");
        let proxy = await XVSVaultProxy.deploy();
        // deploy XVSStore
        const XVSStore = await ethers.getContractFactory("XVSStore");
        let store = await XVSStore.deploy();
        await store.setNewOwner(proxy.address);
        // binding
        await proxy._setPendingImplementation(vault.address);
        await vault._become(proxy.address);
        proxy = XVSVaultOld.attach(proxy.address);
        // deploy token
        const MockERC20 = await ethers.getContractFactory("MockERC20");
        const xvs = await MockERC20.deploy("XVS","XVS",ethers.utils.parseEther("100000000"));
        const vai = await MockERC20.deploy("VAI","VAI",ethers.utils.parseEther("100000000"));
        const token_a = await MockERC20.deploy("TokenA","TAT",ethers.utils.parseEther("100000000"));
        const token_b = await MockERC20.deploy("TokenB","TBT",ethers.utils.parseEther("100000000"));
        await proxy.setXvsStore(xvs.address,store.address);
        // return
        return {
            Owner,Alice,Bob,users,proxy,vault,xvs,vai,store,token_a,token_b
        };
    }

    it("Add a pool and deposit", async function() {
        let {token_a,Alice,Bob,xvs,vai,proxy,store} = await loadFixture(deployAndBindFixture);
        // prepare
        await token_a.transfer(Alice.address,100000000);
        await token_a.transfer(Bob.address,100000000);
        await xvs.transfer(Alice.address,100000000);
        await xvs.transfer(Bob.address,100000000);
        await token_a.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
        await xvs.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
        await token_a.connect(Bob).approve(proxy.address,ethers.constants.MaxUint256);
        await xvs.connect(Bob).approve(proxy.address,ethers.constants.MaxUint256);
        await vai.transfer(store.address,ethers.utils.parseEther("100000000"))

        let all_rewards = ethers.utils.parseEther("1");
        // add pool
```

```
        await expect(proxy.add(vai.address,50,token_a.address,all_rewards,300)).to.emit(
            proxy,"PoolAdded"
        ).withArgs(vai.address,0,token_a.address,50,all_rewards,300);
        await expect(proxy.add(vai.address,100,xvs.address,all_rewards,300)).to.emit(
            proxy,"PoolAdded"
        ).withArgs(vai.address,1,xvs.address,100,all_rewards,300);

        let block = await time.latestBlock();
        // Alice deposit token_a;
        await expect(proxy.connect(Alice).deposit(vai.address,0,10000)).to.emit(
            proxy,"Deposit"
        ).withArgs(Alice.address,vai.address,0,10000);
        let alice_info = await proxy.getUserInfo(vai.address,0,Alice.address);
        expect(alice_info.amount).equal(10000);
        expect(alice_info.rewardDebt).equal(0);
        expect(alice_info.pendingWithdrawals).equal(0);
        let pool_info = await proxy.poolInfos(vai.address,0);
        expect(pool_info.lastRewardBlock).to.eq(block + 1);
        expect(pool_info.accRewardPerShare).to.eq(0);
        await time.increase(9);
        await proxy.connect(Alice).requestWithdrawal(vai.address,0,6000);
        await time.increase(9);
        let requests = await proxy.getWithdrawalRequests(vai.address,0,Alice.address);
        console.log(requests);

        // upgrade
        const XVSVaultNew = await ethers.getContractFactory("XVSVault");
        const vault = await XVSVaultNew.deploy();
        const XVSVaultProxy = await ethers.getContractFactory("XVSVaultProxy");
        proxy = XVSVaultProxy.attach(proxy.address);
        await proxy._setPendingImplementation(vault.address);
        await vault._become(proxy.address);
        proxy = XVSVaultNew.attach(proxy.address);

        requests = await proxy.getWithdrawalRequests(vai.address,0,Alice.address);
        console.log(requests);
    });
});

```

7. UnitTestOutput

```
✔ Check all state vars (1050ms)
  ✔ pause  and resume contract test (202ms)
  ✔ only admin can change admin or implement (64ms)
  ✔ only pending can accept (171ms)
  ✔ only user with access_allowed can add pool
  ✔ Add pool should change state and emit event (475ms)
  ✔ only user with access_allowed can reset pool
  ✔ only valid pool can be reset
  ✔ Reset must update pool (247ms)
  ✔ only user with access_allowed can reset reward
  ✔ set should update pool and change state (174ms)
  ✔ only user with access_allowed can reset locking period
  ✔ Pool must be valid
  ✔ period must be greater than zero (116ms)
  ✔ Reset should change state and emit event (129ms)
  ✔ Delegate should emit event
  ✔ The pool must be valid (38ms)
  ✔ Deposit should update pool and change user's state (649ms)
  ✔ The pool must be valid
  ✔ The pool must be valid
  ✔ requestWithdrawal should emit event and change state (1188ms)
  ✔ delegate before deposit (550ms)
  ✔ Delegate after deposit (446ms)
  MockSlot Upgrade Unit Test
    ✔ Upgrade should not change slot (209ms)

  XVSVault Upgrade Unit Test
[
  [
```

```
    BigNumber { value: "6000" },
    BigNumber { value: "1684223604" },
    amount: BigNumber { value: "6000" },
    lockedUntil: BigNumber { value: "1684223604" }
  ]
]
[
  [
    BigNumber { value: "6000" },
    BigNumber { value: "1684223604" },
    BigNumber { value: "0" },
    amount: BigNumber { value: "6000" },
    lockedUntil: BigNumber { value: "1684223604" },
    afterUpgrade: BigNumber { value: "0" }
  ]
]
    ✔ Add a pool and deposit (968ms)

  VAIVault Unit Test
    Initial state check
      ✔ VAIVaultStorage state check (406ms)
    Change admin and implement test
      ✔ only admin can change admin or implement (60ms)
      ✔ only pending can accept (186ms)
    Pause and resume test
      ✔ pause  and resume contract test (229ms)
    setVenusInfo test
      ✔ only admin can set info
      ✔ VenusInfo can't be reset
    setAccessControl test
      ✔ Only admin can set
      ✔ Can not set to zero address
      ✔ setAccessControl should change state and emit event (46ms)
    updatePendingRewards test
      ✔ updatePending should change records (77ms)
    Deposit test
      ✔ Deposit should be failed while paused (55ms)
      ✔ Deposit should change state and emit event (660ms)
    Bug Demo
      ✔ Rewards can convert to xvsBalance (409ms)

  VRTVault Unit Test
    initial state unit test
      ✔ Check all state vars (370ms)
    _setImplementation unit test
      ✔ _setImplementation only by admin
      ✔ implementation_ can't be zero address
      ✔ _setImplementation should change state and emit event
    initialize unit test
      ✔ initialize twice should be failed
    Pause and resume test
      ✔ pause  and resume contract test (156ms)
    withdrawBep20 unit test
      ✔ only admin can withdraw (39ms)
      ✔ Can withdraw vrt (86ms)
    setLastAccruingBlock unit test
      ✔ only admin can set
      ✔ Set should change state and emit event (74ms)
      ✔ Invalid block setting should be failed (108ms)
    Deposit | Claim | Withdraw unit test
      ✔ Deposit | Claim | Withdraw   (362ms)

  XVSStore Unit Test
    Initial state check
      ✔ Check all state after deploy (206ms)
    setPendingAdmin unit test
      ✔ only admin can call it
      ✔ setPendingAdmin should change state and emit event (46ms)
    acceptAdmin unit test
      ✔ only pendingAdmin can call it
      ✔ acceptAdmin should change state and emit event (44ms)
    setNewOwner
      ✔ only admin can call it
```

```
      ✔ set to zero address should be failed
      ✔ set should change state and emit event (57ms)
    setRewardToken unit test
      ✔ Only owner or admin can call (64ms)
    emergencyRewardWithdraw unit test
      ✔ only owner can call it
      ✔ call it will transfer token (92ms)
    safeRewardTransfer unit test
      ✔ only owner can call it
      ✔ call should be failed while not set reward token
      ✔ Transfer rewards beyond balance test (97ms)
      ✔ Transfer rewards not beyond balance test (81ms)


  65 passing (11s)

```

11.2 External Functions Check Points

1. All_Proxy_output.md

File: contracts/Vault/VAIVaultProxy.sol

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

contract: VAIVaultProxy is VAIVaultAdminStorage, VAIVaultErrorReporter

Index	Function	Visibility	StateMutability	Permission Check	IsUserInterface	Unit Test	Notes
1	_setPendingImplementation(address)	public		onlyAdmin		Passed	
2	_acceptImplementation()	public		onlyPendingImplement		Passed	
3	_setPendingAdmin(address)	public		onlyAdmin		Passed	
4	_acceptAdmin()	public		onlyPendingAdmin		Passed	

File: contracts/VRTVault/VRTVaultProxy.sol

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

contract: VRTVaultProxy is VRTVaultAdminStorage

Index	Function	Visibility	StateMutability	Permission Check	IsUserInterface	Unit Test	Notes
1	_setImplementation(address)	public		onlyAdmin		Passed	
2	_setPendingImplementation(address)	public		onlyAdmin		Passed	
3	_acceptImplementation()	public		onlyPendingImplement		Passed	
4	_setPendingAdmin(address)	public		onlyAdmin		Passed	
5	_acceptAdmin()	public		onlyPendingAdmin		Passed	

File: contracts/XVSVault/XVSVaultProxy.sol

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

contract: XVSVaultProxy is XVSVaultAdminStorage, XVSVaultErrorReporter

Index	Function	Visibility	StateMutability	Permission Check	IsUserInterface	Unit Test	Notes
1	_setPendingImplementation(address)	public		onlyAdmin		Passed	
2	_acceptImplementation()	public		onlyPendingImplement		Passed	
3	_setPendingAdmin(address)	public		onlyAdmin		Passed	
4	_acceptAdmin()	public		onlyPendingAdmin		Passed	

2. VAIVault.sol_output.md

File: contracts/Vault/VAIVault.sol

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

contract: VAIVault is VAIVaultStorage, AccessControlledV5

Index	Function	Visibility	StateMutability	Permission Check	IsUserInterface	Unit Test	Notes
1	pause()	external		_checkAccessAllowed("pause()")		Passed	
2	resume()	external		_checkAccessAllowed("resume()")		Passed	
3	deposit(uint256)	external			Yes	Passed	
4	withdraw(uint256)	external			Yes	Passed	
5	claim()	external			Yes	Skip	
6	claim(address)	external			Yes	Skip	
7	pendingXVS(address)	public	view			Passed	
8	updatePendingRewards()	external				Passed	Public
9	_become(IVAIVaultProxy)	external		onlyAdmin		Passed	
10	setVenusInfo(address,address)	external		onlyAdmin		Passed	
11	setAccessControl(address)	external		onlyAdmin		Passed	

3. VRTVault.sol_output.md

File: contracts/VRTVault/VRTVault.sol

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

contract: VRTVault is VRTVaultStorage, AccessControlledV5

Index	Function	Visibility	StateMutability	Permission Check	IsUserInterface	Unit Test	Notes
1	initialize(address,uint256)	public		onlyAdmin		Passed	onlyOnce
2	pause()	external		_checkAccessAllowed("pause()")		Passed	
3	resume()	external		_checkAccessAllowed("resume()")		Passed	
4	deposit(uint256)	external			True	Passed	
5	getAccruedInterest(address)	public	view			Passed	
6	claim()	external			True	Passed	nonReentrant,isActive
7	claim(address)	external			True	Passed	nonReentrant,isActive
8	withdraw()	external			True	Passed	nonReentrant,isActive
9	withdrawBep20(address,address,uint256)	external		_checkAccessAllowed("withdrawBep20(address,address,uint256)")		Passed	Excessive authority
10	setLastAccruingBlock(uint256)	external		_checkAccessAllowed("setLastAccruingBlock(uint256)")		Passed	
11	getBlockNumber()	public	view			Passed	
12	_become(IVRTVaultProxy)	external		onlyAdmin		Passed	
13	setAccessControl(address)	external		onlyAdmin		Passed	

4. XVSStore.sol_output.md

File: contracts/XVSVault/XVSStore.sol

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

contract: XVSStore

Index	Function	Visibility	StateMutability	Permission Check	IsUserInterface	Unit Test	Notes
1	safeRewardTransfer(address,address,uint256)	external		onlyOwner		Passed	
2	setPendingAdmin(address)	external		onlyAdmin		Passed	
3	acceptAdmin()	external		onlyPendingAdmin		Passed	
4	setNewOwner(address)	external		onlyAdmin		Passed	
5	setRewardToken(address,bool)	external		admin || owner		Passed	
6	emergencyRewardWithdraw(address,uint256)	external		onlyOwner		Passed	

5. XVSVault.sol_output.md

File: contracts/XVSVault/XVSVault.sol

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

contract: XVSVault is XVSVaultStorage, ECDSA, AccessControlledV5

Index	Function	Visibility	StateMutability	Permission Check	IsUserInterface	Unit Test	Notes
1	pause()	external		_checkAccessAllowed("pause()")		Passed	
2	resume()	external		_checkAccessAllowed("resume()")		Passed	
3	poolLength(address)	external	view			Passed	
4	add(address,uint256,IBEP20,uint256,uint256)	external		_checkAccessAllowed("add(address,uint256,address,uint256,uint256)")		Passed	massUpd
5	set(address,uint256,uint256)	external		_checkAccessAllowed("set(address,uint256,uint256)")		Passed	massUpd
6	setRewardAmountPerBlock(address,uint256)	external		_checkAccessAllowed("setRewardAmountPerBlock(address,uint256)")		Passed	massUpd
7	setWithdrawalLockingPeriod(address,uint256,uint256)	external		_checkAccessAllowed("setWithdrawalLockingPeriod(address,uint256,uint256)")		Passed	
8	deposit(address,uint256,uint256)	external			True	Passed	nonReent
9	claim(address,address,uint256)	external			True	Passed	nonReent
10	executeWithdrawal(address,uint256)	external			True	Passed	nonReent
11	requestWithdrawal(address,uint256,uint256)	external			True	Passed	nonReent
12	getEligibleWithdrawalAmount(address,uint256,address)	external	view			Passed	
13	getRequestedAmount(address,uint256,address)	external	view			Passed	
14	getWithdrawalRequests(address,uint256,address)	external	view			Passed	
15	pendingReward(address,uint256,address)	external	view			Passed	
16	updatePool(address,uint256)	external				Passed	Public
17	getUserInfo(address,uint256,address)	external	view			Passed	
18	pendingWithdrawalsBeforeUpgrade(address,uint256,address)	public	view			Passed	
19	delegate(address)	external			True	Passed	
20	delegateBySig(address,uint,uint,uint8,bytes32,bytes32)	external			True	Skip	
21	getCurrentVotes(address)	external	view			Passed	
22	getPriorVotes(address,uint256)	external	view			Passed	
23	_become(IXVSVaultProxy)	external		onlyAdmin		Passed	
24	setXvsStore(address,address)	external		onlyAdmin		Passed	
25	setAccessControl(address)	external		onlyAdmin		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