



Venus Vault

AUDIT REPORT

Version 1.0.0

Serial No. 2023051700012023

Presented by Fairyproof

May 17, 2023

01. Introduction

This document includes the results of the audit performed by the Fairproof 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:

```

1  contracts
2  |   Utils
3  |   |   Address.sol
4  |   |   ECDSA.sol
5  |   |   IBEP20.sol
6  |   |   SafeBEP20.sol
7  |   |   SafeCast.sol
8  |   |   SafeMath.sol
9  |   VRTVault
10 |   |   VRTVault.sol
11 |   |   VRTVaultProxy.sol
12 |   |   VRTVaultStorage.sol
13 |   XSVVault
14 |   |   XVSStore.sol
15 |   |   XSVVault.sol
16 |   |   XSVVaultErrorReporter.sol
17 |   |   XSVVaultProxy.sol
18 |   |   XSVVaultStorage.sol
19
20 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 Fairproof 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 Fairproof.

- 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 Fairproof.

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/>

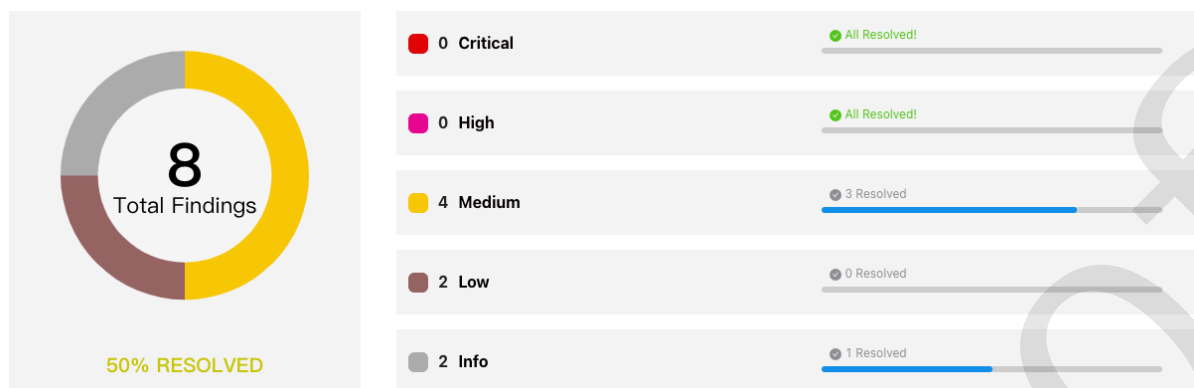
Whitepaper: <https://venus.io/Whitepaper.pdf>

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 Right | 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 Right

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 vxv 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 to 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 vxv 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 vxv is distributed as a reward token, safeXVSTransfer will update vxvBalance with the latest balance of the vxv token and vxvBalance will include the reward that is transferred into the vault but hasn't been updated. In this case the vxv 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's 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 `rewardTokens`s. However when these various `rewardTokens`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 stated 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 transferring it to a multi-sig wallet or DAO when necessary.

11. Appendices

11.1 Unit Test

1. VAIVault.js

```

1  const {
2    time,
3    loadFixture,
4  } = require("@nomicfoundation/hardhat-network-helpers");
5  const { expect } = require("chai");
6  const { ethers } = require("hardhat");
7
8  const zero_address = ethers.constants.AddressZero;
9
10 describe("VAIVault Unit Test", function () {
11   async function deployAndBindFixture() {
12     // get users;
13     const [Owner, Alice, Bob, ...users] = await ethers.getSigners();
14     // Deploy access_controller
15     const MockAccessControlManagerV5 = await ethers.getContractFactory("MockAccessControlManagerV5");
16     const access_controller = await MockAccessControlManagerV5.deploy();
17     // deploy VAIVault;
18     const VAIVault = await ethers.getContractFactory("VAIVault");
19     const vault = await VAIVault.deploy();
20     // deploy proxy
21     const VAIVaultProxy = await ethers.getContractFactory("VAIVaultProxy");
22     let proxy = await VAIVaultProxy.deploy();
23     // binding
24     await proxy._setPendingImplementation(vault.address);
25     await vault._become(proxy.address);
26     proxy = VAIVault.attach(proxy.address);
27     await proxy.setAccessControl(access_controller.address);
28     // deploy token
29     const MockERC20 = await ethers.getContractFactory("MockERC20");
30     const xvs = await MockERC20.deploy("XVS", "XVS", ethers.utils.parseEther("100000000"));
31     const vai = await MockERC20.deploy("VAI", "VAI", ethers.utils.parseEther("100000000"));
32     await proxy.setVenusInfo(xvs.address, vai.address);
33     // return
34     return {
35       Owner, Alice, Bob, users, proxy, vault, xvs, vai, access_controller
36     };
37   }
38
39   describe("Initial state check", function() {
40     it("VAIVaultStorage state check", async function() {
41       const {Owner, proxy, vault, xvs, vai, access_controller} = await loadFixture(deployAndBindFixture);
42       expect(await proxy.admin()).to.equal(Owner.address);

```

```

43     expect(await proxy.pendingAdmin()).to.equal(zero_address);
44     expect(await proxy.vaiVaultImplementation()).to.equal(vault.address);
45     expect(await proxy.pendingVAIVaultImplementation()).to.equal(zero_address);
46     expect(await proxy.xvs()).to.equal(xvs.address);
47     expect(await proxy.vai()).to.equal(vai.address);
48     expect(await proxy.xvsBalance()).to.equal(0);
49     expect(await proxy.accXVSPerShare()).to.equal(0);
50     expect(await proxy.pendingRewards()).to.equal(0);
51     expect(await proxy.vaultPaused()).to.equal(false);
52     expect(await proxy.accessControlManager()).to.equal(access_controller.address);
53   });
54 });
55
56 describe("Change admin and implement test", function() {
57   it("only admin can change admin or implement", async function() {
58     const {Alice, proxy, users} = await loadFixture(deployAndBindFixture);
59     const VAIVaultProxy = await ethers.getContractFactory("VAIVaultProxy");
60     let instance = VAIVaultProxy.attach(proxy.address);
61     await expect(instance.connect(Alice)._setPendingAdmin(users[0].address)).to.emit(
62       instance, "Failure"
63     ).withArgs(1, 2, 0);
64     expect(await proxy.pendingAdmin()).to.equal(zero_address);
65
66     await expect(instance.connect(Alice)._setPendingImplementation(users[0].address)).to.emit(
67       instance, "Failure"
68     ).withArgs(1, 3, 0);
69     expect(await proxy.pendingVAIVaultImplementation()).to.equal(zero_address);
70   });
71
72   it("only pending can accept", async function() {
73     const {Owner, Alice, Bob, proxy, vault} = await loadFixture(deployAndBindFixture);
74     const VAIVaultProxy = await ethers.getContractFactory("VAIVaultProxy");
75     let instance = VAIVaultProxy.attach(proxy.address);
76
77     // set pending
78     await expect(instance._setPendingAdmin(Alice.address)).to.emit(
79       instance, "NewPendingAdmin"
80     ).withArgs(zero_address, Alice.address);
81     expect(await proxy.pendingAdmin()).to.equal(Alice.address);
82     await expect(instance._setPendingImplementation(Bob.address)).to.emit(
83       instance, "NewPendingImplementation"
84     ).withArgs(zero_address, Bob.address);
85     expect(await proxy.pendingVAIVaultImplementation()).to.equal(Bob.address);
86
87     // accept without pending
88     await expect(instance._acceptAdmin()).to.emit(
89       instance, "Failure"
90     ).withArgs(1, 0, 0);
91     expect(await proxy.pendingAdmin()).to.equal(Alice.address);
92     await expect(instance._acceptImplementation()).to.emit(
93       instance, "Failure"
94     ).withArgs(1, 1, 0);
95     expect(await proxy.pendingVAIVaultImplementation()).to.equal(Bob.address);
96
97     // accept with pending
98     await expect(instance.connect(Alice)._acceptAdmin()).to.emit(
99       instance, "NewAdmin"
100    ).withArgs(Owner.address, Alice.address);
101
102    await expect(instance.connect(Bob)._acceptImplementation()).to.emit(
103      instance, "NewImplementation"
104    ).withArgs(vault.address, Bob.address);
105
106    // check final state
107    expect(await proxy.admin()).to.equal(Alice.address);
108    expect(await proxy.pendingAdmin()).to.equal(zero_address);
109    expect(await proxy.vaiVaultImplementation()).to.equal(Bob.address);
110    expect(await proxy.pendingVAIVaultImplementation()).to.equal(zero_address);
111  });
112 });
113
114 describe("Pause and resume test", function() {
115   it("pause and resume contract test", async function() {

```

```

116     const {Owner,Alice,proxy} = await loadFixture(deployAndBindFixture);
117     // only Owner
118     await expect(proxy.connect(Alice).pause()).to.revertedWith("Unauthorized");
119     await expect(proxy.connect(Alice).resume()).to.revertedWith("Unauthorized");
120     // pause emit event
121     await expect(proxy.pause()).to.emit(
122         proxy,"VaultPaused"
123     ).withArgs(Owner.address);
124     expect(await proxy.vaultPaused()).to.equal(true);
125     // pause twice should be failed
126     await expect(proxy.pause()).to.revertedWith("Vault is already paused");
127
128     // resume emit event
129     await expect(proxy.resume()).to.emit(
130         proxy,"VaultResumed"
131     ).withArgs(Owner.address);
132     expect(await proxy.vaultPaused()).to.equal(false);
133     // resume twice should be failed
134     await expect(proxy.resume()).to.revertedWith("Vault is not paused");
135 });
136 });
137
138 //
139 describe("setVenusInfo test", function() {
140     it("only admin can set info", async function() {
141         const {proxy,Alice,users} = await loadFixture(deployAndBindFixture);
142         await
143         expect(proxy.connect(Alice).setVenusInfo(users[0].address,users[1].address)).to.revertedWith("only admin
144         can");
145     });
146
147     // reset
148     it("VenusInfo can't be reset", async function() {
149         const {proxy,users} = await loadFixture(deployAndBindFixture);
150         await expect(proxy.setVenusInfo(users[0].address,users[1].address)).to.revertedWith("addresses
151         already set");
152     });
153 });
154
155 // describe("Get/Set Admin or Burn Admin test", function() {
156 //     // redundant
157 //     it("get and burn admin test", async function() {
158 //         const {Owner,proxy} = await loadFixture(deployAndBindFixture);
159 //         expect(await proxy.getAdmin()).to.equal(Owner.address);
160 //     });
161
162 //     it("only admin can burn admin", async function () {
163 //         const {Alice,proxy} = await loadFixture(deployAndBindFixture);
164 //         await expect(proxy.connect(Alice).burnAdmin()).to.revertedWith("only admin can");
165 //     });
166
167 //     // AdminTransferred => AdminTransferred
168 //     it("Burn admin should emit event and change state", async function() {
169 //         const {Owner,proxy} = await loadFixture(deployAndBindFixture);
170 //         await expect(proxy.burnAdmin()).to.emit(
171 //             proxy,"AdminTransferred"
172 //         ).withArgs(Owner.address,zero_address);
173 //     });
174
175 //     // redundant
176 //     it("Set only admin", async function() {
177 //         const {Alice,proxy} = await loadFixture(deployAndBindFixture);
178 //         await expect(proxy.connect(Alice).setNewAdmin(Alice.address)).to.revertedWith("only admin
179 //         can");
180 //     });
181
182 //     it("Set to zero address should be failed", async function() {
183 //         const {proxy} = await loadFixture(deployAndBindFixture);
184 //         await expect(proxy.setNewAdmin(zero_address)).to.revertedWith("new owner is the zero address");
185 //     });
186
187 //     // AdminTransferred => AdminTransferred
188 //     it("Set new admin should emit event and change state", async function() {

```

```

185 //      const {Owner,proxy,Alice} = await loadFixture(deployAndBindFixture);
186 //      await expect(proxy.setNewAdmin(Alice.address)).to.emit(
187 //        proxy,"AdminTransferred"
188 //      ).withArgs(Owner.address,Alice.address);
189 //    });
190 //  });
191
192 describe("setAccessControl test", function() {
193   it("Only admin can set", async function() {
194     const {Alice,proxy} = await loadFixture(deployAndBindFixture);
195     await expect(proxy.connect(Alice).setAccessControl(zero_address)).to.revertedWith("only admin
can");
196   });
197   it("Can not set to zero address", async function() {
198     const {proxy} = await loadFixture(deployAndBindFixture);
199     await expect(proxy.setAccessControl(zero_address)).to.revertedWith("invalid access control manager
address");
200   });
201   it("setAccessControl should change state and emit event", async function() {
202     const {proxy,access_controller,Bob} = await loadFixture(deployAndBindFixture);
203     await expect(proxy.setAccessControl(Bob.address)).to.emit(
204       proxy,"NewAccessControlManager"
205     ).withArgs(access_controller.address,Bob.address);
206     expect(await proxy.accessControlManager()).to.equal(Bob.address);
207   });
208 });
209
210 describe("updatePendingRewards test", function() {
211   it("updatePending should change records", async function() {
212     let value = ethers.constants.WeiPerEther;
213     const {xvs,proxy} = await loadFixture(deployAndBindFixture);
214     await xvs.transfer(proxy.address, value);
215     await proxy.updatePendingRewards();
216     expect(await proxy.xvsBalance()).to.equal(value);
217     expect(await proxy.pendingRewards()).to.equal(value);
218   });
219 });
220
221 //
222 describe("Deposit test", function () {
223   it("Deposit should be failed while paused", async function() {
224     const {Alice,proxy} = await loadFixture(deployAndBindFixture);
225     await proxy.pause();
226     await expect(proxy.connect(Alice).deposit(10000)).to.revertedWith("Vault is paused");
227   });
228
229   it("Deposit should change state and emit event", async function() {
230     const {xvs,vai,Alice,Bob,proxy} = await loadFixture(deployAndBindFixture);
231     // prepare
232     await xvs.transfer(proxy.address,ethers.constants.WeiPerEther);
233     await proxy.updatePendingRewards();
234     await vai.transfer(Alice.address,ethers.utils.parseEther("10000"));
235     await vai.transfer(Bob.address,ethers.utils.parseEther("10000"));
236     await vai.connect(Alice).approve(proxy.address,ethers.constants.MaxInt256);
237     await vai.connect(Bob).approve(proxy.address,ethers.constants.MaxInt256);
238     // Alice deposit
239     let deposit_amount = ethers.utils.parseEther("100");
240     await expect(proxy.connect(Alice).deposit(deposit_amount)).to.emit(
241       proxy,"Deposit"
242     ).withArgs(Alice.address,deposit_amount);
243     let alice_info = await proxy.userInfo(Alice.address);
244     expect(alice_info.amount).to.equal(deposit_amount);
245     expect(alice_info.rewardDebt).to.equal(0);
246     expect(await proxy.accXVSPerShare()).to.equal(0);
247     // Bob deposit
248     await proxy.connect(Bob).deposit(deposit_amount);
249     expect(await proxy.accXVSPerShare()).to.equal(ethers.constants.WeiPerEther.div(100));
250     let bob_info = await proxy.userInfo(Bob.address);
251     expect(bob_info.amount).to.equal(deposit_amount);
252     expect(bob_info.rewardDebt).to.equal(ethers.constants.WeiPerEther);
253     // check pending
254     expect(await proxy.pendingXVS(Alice.address)).to.equal(ethers.constants.WeiPerEther);
255     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);

```

```

329
330     // dead coin
331     let xvsBalance = await proxy.xvsBalance();
332     expect(xvsBalance).to.equal(balance);
333
334     let pending = await proxy.pendingRewards();
335     expect(pending).to.equal(0);
336     // Bob has no rewards
337     let pending_bob = await proxy.pendingXVS(Bob.address);
338     expect(pending_bob).to.equal(0);
339 });
340 });
341
342 // describe("Recover owner demo", function() {
343 //     it("Recover owner after burn owner", async function() {
344 //         const {Alice,proxy,Bob} = await loadFixture(deployAndBindFixture);
345 //         const VAIVaultProxy = await ethers.getContractFactory("VAIVaultProxy");
346 //         let instance = VAIVaultProxy.attach(proxy.address);
347 //         await instance._setPendingAdmin(Alice.address);
348 //         expect(await proxy.pendingAdmin()).to.equal(Alice.address);
349 //         await proxy.burnAdmin();
350 //         expect(await proxy.admin()).to.equal(zero_address);
351 //         await instance.connect(Alice)._acceptAdmin();
352 //         expect(await proxy.admin()).to.equal(Alice.address);
353 //         expect(await proxy.pendingAdmin()).to.equal(zero_address);
354 //     });
355
356 //     it("Recover owner after setNewAdmin", async function() {
357 //         const {Alice,proxy,Bob} = await loadFixture(deployAndBindFixture);
358 //         const VAIVaultProxy = await ethers.getContractFactory("VAIVaultProxy");
359 //         let instance = VAIVaultProxy.attach(proxy.address);
360 //         await instance._setPendingAdmin(Alice.address);
361 //         expect(await proxy.pendingAdmin()).to.equal(Alice.address);
362 //         await proxy.setNewAdmin(Bob.address);
363 //         expect(await proxy.admin()).to.equal(Bob.address);
364 //         await instance.connect(Alice)._acceptAdmin();
365 //         expect(await proxy.admin()).to.equal(Alice.address);
366 //         expect(await proxy.pendingAdmin()).to.equal(zero_address);
367 //     });
368 // });
369 });
370

```

2. VRTVault.js

```

1  const {
2      time,
3      mine,
4      loadFixture,
5  } = require("@nomicfoundation/hardhat-network-helpers");
6  const { anyValue } = require("@nomicfoundation/hardhat-chai-matchers/withArgs");
7  const { expect } = require("chai");
8  const { ethers } = require("hardhat");
9
10
11  const zero_address = ethers.constants.AddressZero;
12
13  describe("VRTVault Unit Test", function () {
14      async function deployAndInitializeFixture() {
15          // get users;
16          const [Owner, Alice,Bob,...users] = await ethers.getSigners();
17          // Deploy access_controller
18          const MockAccessControlManagerV5 = await ethers.getContractFactory("MockAccessControlManagerV5");
19          const access_controller = await MockAccessControlManagerV5.deploy();
20          // deploy VRTVault;
21          const VRTVault = await ethers.getContractFactory("VRTVault");
22          const vault = await VRTVault.deploy();
23          // deploy token
24          const MockERC20 = await ethers.getContractFactory("MockERC20");
25          const vrt = await MockERC20.deploy("VRT", "VRT",ethers.utils.parseEther("100000000"));
26          // deploy proxy

```

```

27     const VRTVaultProxy = await ethers.getContractFactory("VRTVaultProxy");
28     let proxy = await VRTVaultProxy.deploy(
29         vault.address,
30         vrt.address,
31         ethers.utils.parseEther("0.005")
32     );
33     proxy = VRTVault.attach(proxy.address);
34     await proxy.setAccessControl(access_controller.address);
35     return {
36         Owner,Alice,Bob,users,proxy,vault,vrt
37     };
38 }
39
40 describe("initial state unit test", function() {
41     it("Check all state vars", async function() {
42         const {Owner,proxy,vault,vrt} = await loadFixture(deployAndInitializeFixture);
43         const VRTVaultProxy = await ethers.getContractFactory("VRTVaultProxy");
44         const instance = VRTVaultProxy.attach(proxy.address);
45         // instance
46         expect(await instance.admin()).to.equal(Owner.address);
47         expect(await instance.implementation()).to.equal(vault.address);
48         expect(await instance.pendingAdmin()).to.equal(zero_address);
49         expect(await instance.pendingImplementation()).to.equal(zero_address);
50         // proxy
51         expect(await proxy._notEntered()).to.equal(true);
52         expect(await proxy.vaultPaused()).to.equal(false);
53         expect(await proxy.vrt()).to.equal(vrt.address);
54         expect(await proxy.interestRatePerBlock()).to.equal(ethers.utils.parseEther("0.005"));
55         expect(await proxy.lastAccruingBlock()).to.equal(0);
56     });
57 });
58
59 describe("_setImplementation unit test", function() {
60     it("_setImplementation only by admin", async function() {
61         const {Alice,proxy,users} = await loadFixture(deployAndInitializeFixture);
62         const VRTVaultProxy = await ethers.getContractFactory("VRTVaultProxy");
63         const instance = VRTVaultProxy.attach(proxy.address);
64         await
65 expect(instance.connect(Alice)._setImplementation(users[0].address)).to.revertedWith("VRTVaultProxy::_setImplementation: admin only");
66     });
67
68     it("implementation_ can't be zero address", async function() {
69         const {proxy} = await loadFixture(deployAndInitializeFixture);
70         const VRTVaultProxy = await ethers.getContractFactory("VRTVaultProxy");
71         const instance = VRTVaultProxy.attach(proxy.address);
72         await
73 expect(instance._setImplementation(zero_address)).to.revertedWith("VRTVaultProxy::_setImplementation: invalid implementation address");
74     });
75
76     it("_setImplementation should change state and emit event", async function() {
77         const {proxy,vault,users} = await loadFixture(deployAndInitializeFixture);
78         const VRTVaultProxy = await ethers.getContractFactory("VRTVaultProxy");
79         const instance = VRTVaultProxy.attach(proxy.address);
80         expect(await instance._setImplementation(users[0].address)).to.emit(
81             instance,"NewImplementation"
82         ).withArgs(vault.address,users[0].address);
83         expect(await instance.implementation()).to.equal(users[0].address);
84     });
85
86     describe("initialize unit test", function(){
87         it("initialize twice should be failed", async function() {
88             const {proxy,vrt} = await loadFixture(deployAndInitializeFixture);
89             await
90 expect(proxy.initialize(vrt.address,ethers.utils.parseEther("0.005"))).to.revertedWith("Vault may only be initialized once");
91         });
92     });
93
94     describe("Pause and resume test", function() {
95         it("pause and resume contract test", async function() {

```



```

94     const {Owner,Alice,proxy} = await loadFixture(deployAndInitializeFixture);
95     // only Owner
96     await expect(proxy.connect(Alice).pause()).to.revertedWith("Unauthorized");
97     await expect(proxy.connect(Alice).resume()).to.revertedWith("Unauthorized");
98     // pause emit event
99     await expect(proxy.pause()).to.emit(
100         proxy,"VaultPaused"
101     ).withArgs(Owner.address);
102     expect(await proxy.vaultPaused()).to.equal(true);
103     // pause twice should be failed
104     await expect(proxy.pause()).to.revertedWith("Vault is already paused");
105
106     // resume emit event
107     await expect(proxy.resume()).to.emit(
108         proxy,"VaultResumed"
109     ).withArgs(Owner.address);
110     expect(await proxy.vaultPaused()).to.equal(false);
111     // resume twice should be failed
112     await expect(proxy.resume()).to.revertedWith("Vault is not paused");
113 });
114 });
115
116 describe("withdrawBep20 unit test", function() {
117     it("only admin can withdraw", async function() {
118         const {Owner,Alice,proxy} = await loadFixture(deployAndInitializeFixture);
119         await expect(proxy.connect(Alice).withdrawBep20(Owner.address,Alice.address,100)).to.revertedWith(
120             "Unauthorized"
121         );
122     });
123
124     // risk
125     it("Can withdraw vrt", async function() {
126         const {vrt,Alice,proxy} = await loadFixture(deployAndInitializeFixture);
127         await vrt.transfer(proxy.address,10000);
128         expect(await vrt.balanceOf(proxy.address)).to.equal(10000);
129         expect(await vrt.balanceOf(Alice.address)).to.equal(0);
130         await expect(proxy.withdrawBep20(vrt.address,Alice.address,10000)).to.emit(
131             proxy,"WithdrawToken"
132         ).withArgs(vrt.address,Alice.address,10000);
133         expect(await vrt.balanceOf(proxy.address)).to.equal(0);
134         expect(await vrt.balanceOf(Alice.address)).to.equal(10000);
135     });
136 });
137
138 describe("setLastAccruingBlock unit test", function() {
139     it("only admin can set", async function() {
140         const {Alice,proxy} = await loadFixture(deployAndInitializeFixture);
141         await expect(proxy.connect(Alice).setLastAccruingBlock(100)).to.revertedWith("Unauthorized");
142     });
143
144     it("Set should change state and emit event", async function() {
145         const {proxy} = await loadFixture(deployAndInitializeFixture);
146         let block = await time.latestBlock();
147         await expect(proxy.setLastAccruingBlock(block + 10)).to.emit(
148             proxy,"LastAccruingBlockChanged"
149         ).withArgs(0,block + 10);
150         expect(await proxy.lastAccruingBlock()).to.equal(block + 10);
151
152         await expect(proxy.setLastAccruingBlock(block + 8)).to.emit(
153             proxy,"LastAccruingBlockChanged"
154         ).withArgs(block + 10,block + 8);
155         expect(await proxy.lastAccruingBlock()).to.equal(block + 8);
156     });
157
158     it("Invalid block setting should be failed", async function(){
159         const {proxy} = await loadFixture(deployAndInitializeFixture);
160         let block = await time.latestBlock();
161         await expect(proxy.setLastAccruingBlock(block + 10)).to.emit(
162             proxy,"LastAccruingBlockChanged"
163         ).withArgs(0,block + 10);
164         expect(await proxy.lastAccruingBlock()).to.equal(block + 10);
165
166         await expect(proxy.setLastAccruingBlock(block + 8)).to.emit(

```



```

167         proxy, "LastAccruingBlockChanged"
168     ).withArgs(block + 10, block + 8);
169     expect(await proxy.lastAccruingBlock()).to.equal(block + 8);
170
171     await expect(proxy.setLastAccruingBlock(block - 1)).to.rejectedWith(
172         "Invalid _lastAccruingBlock interest have been accumulated"
173     );
174 });
175 });
176
177 describe("Deposit | Claim | Withdraw unit test", function() {
178     it("Deposit | Claim | Withdraw ", async function() {
179         // prepare
180         const {Alice, vrt, proxy} = await loadFixture(deployAndInitializeFixture);
181         await vrt.transfer(Alice.address, 10000);
182         await vrt.connect(Alice).approve(proxy.address, ethers.constants.MaxUint256);
183         let block = await time.latestBlock();
184         await proxy.setLastAccruingBlock(block + 10);
185         // first deposit
186         await expect(proxy.connect(Alice).deposit(1000)).to.emit(
187             proxy, "Deposit"
188         ).withArgs(Alice.address, 1000);
189         let info = await proxy.userInfo(Alice.address);
190         expect(info.userAddress).to.equal(Alice.address);
191         expect(info.accrualStartBlockNumber).to.equal(block + 2);
192         expect(info.totalPrincipalAmount).to.equal(1000);
193         expect(info.lastWithdrawnBlockNumber).to.equal(0); // unused
194
195         // deposit again;
196         await expect(proxy.connect(Alice).deposit(1000)).to.emit(
197             proxy, "Claim"
198         ).withArgs(Alice.address, 1000 * 0.005);
199
200         info = await proxy.userInfo(Alice.address);
201         expect(info.userAddress).to.equal(Alice.address);
202         expect(info.accrualStartBlockNumber).to.equal(block + 3);
203         expect(info.totalPrincipalAmount).to.equal(2000);
204         expect(info.lastWithdrawnBlockNumber).to.equal(0);
205
206         await mine(5);
207         expect(await proxy.getAccruedInterest(Alice.address)).to.equal(
208             2000 * 5 * 0.005
209         );
210
211         // claim
212         await expect(proxy.functions["claim(address)"](Alice.address)).to.emit(
213             proxy, "Claim"
214         ).withArgs(Alice.address, 2000 * 6 * 0.005);
215
216         // withdraw should be failed
217         await expect(proxy.connect(Alice).withdraw()).to.revertedWith("Failed to transfer VRT,
Insufficient VRT in Vault.");
218
219         let start_block = block + 2;
220         let end_block = block + 10;
221         let deposit_block = await time.latestBlock() + 1;
222
223         let interest = 0;
224         if (deposit_block >= end_block) {
225             interest = 1000 * 0.005 + 2000 * 7 * 0.005;
226         } else {
227             interest = 1000 * 0.005 + (deposit_block - start_block - 1) * 2000 * 0.005;
228         }
229         await vrt.transfer(proxy.address, interest);
230         // withdraw should be successful
231         await expect(proxy.connect(Alice).withdraw()).to.emit(
232             proxy, "Withdraw"
233         ).withArgs(Alice.address, anyValue, 2000, anyValue);
234         expect(await vrt.balanceOf(proxy.address)).to.equal(0);
235     });
236 });
237
238 });

```

3. XVSStore.js

```

1  const {
2    loadFixture,
3  } = require("@nomicfoundation/hardhat-network-helpers");
4
5  const { expect } = require("chai");
6  const { ethers } = require("hardhat");
7
8  const zero_address = ethers.constants.AddressZero;
9
10 describe("XVSStore Unit Test", function () {
11   async function deployFixture() {
12     // get users;
13     const [Owner, Alice, Bob, ...users] = await ethers.getSigners();
14     // Deploy XVSStore
15     const XVSStore = await ethers.getContractFactory("XVSStore");
16     const store = await XVSStore.deploy();
17     // token
18     const MockERC20 = await ethers.getContractFactory("MockERC20");
19     const xvs = await MockERC20.deploy("XVS", "XVS", ethers.utils.parseEther("100000000"));
20     const vai = await MockERC20.deploy("VAI", "VAI", ethers.utils.parseEther("100000000"));
21     return {
22       store, Owner, Alice, Bob, xvs, vai, users
23     };
24   }
25
26   describe("Initial state check", function() {
27     it("Check all state after deploy", async function(){
28       const {store, Owner} = await loadFixture(deployFixture);
29       expect(await store.admin()).to.equal(Owner.address);
30       expect(await store.pendingAdmin()).to.equal(zero_address);
31       expect(await store.owner()).to.equal(zero_address);
32     });
33   });
34
35   describe("setPendingAdmin unit test", function() {
36     it("only admin can call it", async function() {
37       const {store, Alice} = await loadFixture(deployFixture);
38       await expect(store.connect(Alice).setPendingAdmin(Alice.address)).to.revertedWith("only admin
39 can");
40     });
41
42     it("setPendingAdmin should change state and emit event", async function() {
43       const {store, Alice, Bob} = await loadFixture(deployFixture);
44       await expect(store.setPendingAdmin(Alice.address)).to.emit(
45         store, "NewPendingAdmin"
46       ).withArgs(zero_address, Alice.address);
47       expect(await store.pendingAdmin()).to.equal(Alice.address);
48       // can set twice
49       await expect(store.setPendingAdmin(Bob.address)).to.emit(
50         store, "NewPendingAdmin"
51       ).withArgs(Alice.address, Bob.address);
52       expect(await store.pendingAdmin()).to.equal(Bob.address);
53     });
54   });
55
56   describe("acceptAdmin unit test", function() {
57     it("only pendingAdmin can call it", async function() {
58       const {store, Alice, Bob} = await loadFixture(deployFixture);
59       await store.setPendingAdmin(Alice.address);
60       await expect(store.connect(Bob).acceptAdmin()).to.revertedWith("only pending admin");
61     });
62
63     it("acceptAdmin should change state and emit event", async function() {
64       const {store, Alice, Owner} = await loadFixture(deployFixture);
65       await store.setPendingAdmin(Alice.address);
66       await expect(store.connect(Alice).acceptAdmin()).to.emit(
67         store, "AdminTransferred"

```

```

67         ).withArgs(Owner.address,Alice.address);
68         expect(await store.admin()).to.equal(Alice.address);
69         expect(await store.pendingAdmin()).to.equal(zero_address);
70     });
71 });
72
73 describe("setNewOwner", function() {
74     it("only admin can call it", async function(){
75         const {store,Alice} = await loadFixture(deployFixture);
76         await expect(store.connect(Alice).setNewOwner(Alice.address)).to.revertedWith("only admin can");
77     });
78     it("set to zero address should be failed", async function() {
79         const {store} = await loadFixture(deployFixture);
80         await expect(store.setNewOwner(zero_address)).to.revertedWith("new owner is the zero address");
81     });
82
83     it("set should change state and emit event", async function() {
84         const {store,Alice,Bob} = await loadFixture(deployFixture);
85         await expect(store.setNewOwner(Alice.address)).to.emit(
86             store,"OwnerTransferred"
87         ).withArgs(zero_address,Alice.address);
88         expect(await store.owner()).to.equal(Alice.address);
89         // owner can't call it
90         await expect(store.connect(Alice).setNewOwner(Bob.address)).to.revertedWith("only admin can");
91         // set again
92         await expect(store.setNewOwner(Bob.address)).to.emit(
93             store,"OwnerTransferred"
94         ).withArgs(Alice.address,Bob.address);
95         expect(await store.owner()).to.equal(Bob.address);
96     });
97 });
98
99 describe("setRewardToken unit test",function() {
100     it("Only owner or admin can call", async function() {
101         const {store,Alice,Bob,xvs} = await loadFixture(deployFixture);
102         await store.setNewOwner(Alice.address);
103         await expect(store.connect(Bob).setRewardToken(xvs.address,true)).to.revertedWith("only admin or
owner can");
104
105         await store.connect(Alice).setRewardToken(xvs.address,true);
106         expect(await store.rewardTokens(xvs.address)).to.equal(true);
107
108         await store.setRewardToken(xvs.address,false);
109         expect(await store.rewardTokens(xvs.address)).to.equal(false);
110     });
111 });
112
113 describe("emergencyRewardWithdraw unit test", function() {
114     it("only owner can call it", async function() {
115         const {store,Owner} = await loadFixture(deployFixture);
116         await expect(store.emergencyRewardWithdraw(Owner.address,100)).to.rejectedWith("only owner can");
117     });
118
119     it("call it will transfer token", async function() {
120         const {store,Alice,xvs} = await loadFixture(deployFixture);
121         await store.setNewOwner(Alice.address);
122         await xvs.transfer(store.address,10000);
123         expect(await xvs.balanceOf(store.address)).to.equal(10000);
124         expect(await xvs.balanceOf(Alice.address)).to.equal(0);
125         await expect(store.connect(Alice).emergencyRewardWithdraw(xvs.address,1000)).to.emit(
126             xvs,"Transfer"
127         ).withArgs(store.address,Alice.address,1000);
128
129         expect(await xvs.balanceOf(store.address)).to.equal(9000);
130         expect(await xvs.balanceOf(Alice.address)).to.equal(1000);
131     });
132 });
133
134 describe("safeRewardTransfer unit test", function() {
135     it("only owner can call it", async function() {
136         const {store,Owner,xvs} = await loadFixture(deployFixture);
137         await expect(store.safeRewardTransfer(xvs.address,Owner.address,100)).to.rejectedWith("only owner
can");

```

```

138     });
139
140     it("call should be failed while not set reward token", async function() {
141         const {store,Owner,xvs} = await loadFixture(deployFixture);
142         await store.setNewOwner(Owner.address);
143         await expect(store.safeRewardTransfer(xvs.address,Owner.address,100)).to.rejectedWith("only reward
token can");
144     });
145
146     it("Transfer rewards beyond balance test", async function() {
147         const {store,Owner,xvs,Alice} = await loadFixture(deployFixture);
148         await store.setNewOwner(Owner.address);
149         await store.setRewardToken(xvs.address,true);
150         await xvs.transfer(store.address,10000);
151         await expect(store.safeRewardTransfer(xvs.address,Alice.address,200000)).to.emit(
152             xvs,"Transfer"
153         ).withArgs(store.address,Alice.address,10000);
154         expect(await xvs.balanceOf(store.address)).to.equal(0);
155         expect(await xvs.balanceOf(Alice.address)).to.equal(10000);
156     });
157
158     it("Transfer rewards not beyond balance test", async function() {
159         const {store,Owner,xvs,Alice} = await loadFixture(deployFixture);
160         await store.setNewOwner(Owner.address);
161         await store.setRewardToken(xvs.address,true);
162         await xvs.transfer(store.address,10000);
163         await expect(store.safeRewardTransfer(xvs.address,Alice.address,2000)).to.emit(
164             xvs,"Transfer"
165         ).withArgs(store.address,Alice.address,2000);
166         expect(await xvs.balanceOf(store.address)).to.equal(8000);
167         expect(await xvs.balanceOf(Alice.address)).to.equal(2000);
168     });
169 });
170 });
171

```

4. XSVVault.js

```

1  const {
2      time,
3      loadFixture,
4  } = require("@nomicfoundation/hardhat-network-helpers");
5  const { anyValue } = require("@nomicfoundation/hardhat-chai-matchers/withArgs");
6  const { expect } = require("chai");
7  const { ethers } = require("hardhat");
8  const { describe } = require("node:test");
9
10 const zero_address = ethers.constants.AddressZero;
11 const GAIN = ethers.utils.parseUnits("1.0",12);
12
13 describe("XSVVault Unit Test", function () {
14     async function deployAndBindFixture() {
15         // get users;
16         const [Owner, Alice,Bob,...users] = await ethers.getSigners();
17         // Deploy access_controller
18         const MockAccessControlManagerV5 = await ethers.getContractFactory("MockAccessControlManagerV5");
19         const access_controller = await MockAccessControlManagerV5.deploy();
20         // deploy XSVVault;
21         const XSVVault = await ethers.getContractFactory("XSVVault");
22         const vault = await XSVVault.deploy();
23         // deploy proxy
24         const XSVVaultProxy = await ethers.getContractFactory("XSVVaultProxy");
25         let proxy = await XSVVaultProxy.deploy();
26         // deploy XVSStore
27         const XVSStore = await ethers.getContractFactory("XVSStore");
28         let store = await XVSStore.deploy();
29         await store.setNewOwner(proxy.address);
30         // binding
31         await proxy._setPendingImplementation(vault.address);
32         await vault._become(proxy.address);
33         proxy = XSVVault.attach(proxy.address);

```

```

34     await proxy.setAccessControl(access_controller.address);
35     // deploy token
36     const MockERC20 = await ethers.getContractFactory("MockERC20");
37     const xvs = await MockERC20.deploy("XVS", "XVS", ethers.utils.parseEther("100000000"));
38     const vai = await MockERC20.deploy("VAI", "VAI", ethers.utils.parseEther("100000000"));
39     const token_a = await MockERC20.deploy("TokenA", "TAT", ethers.utils.parseEther("100000000"));
40     const token_b = await MockERC20.deploy("TokenB", "TBT", ethers.utils.parseEther("100000000"));
41     await proxy.setXvsStore(xvs.address, store.address);
42     // return
43     return {
44         Owner, Alice, Bob, users, proxy, vault, xvs, vai, access_controller, store, token_a, token_b
45     };
46 }
47
48 describe("Initial State check", function() {
49     it("Check all state vars", async function() {
50         const {proxy, vault, Owner, xvs, store} = await loadFixture(deployAndBindFixture);
51         expect(await proxy.admin()).to.equal(Owner.address);
52         expect(await proxy.pendingAdmin()).to.equal(zero_address);
53         expect(await proxy.implementation()).to.equal(vault.address);
54         expect(await proxy.pendingXSVVaultImplementation()).to.equal(zero_address);
55         expect(await proxy.xvsStore()).to.equal(store.address);
56         expect(await proxy.xvsAddress()).to.equal(xvs.address);
57         expect(await proxy.vaultPaused()).to.equal(false);
58     });
59 });
60
61 describe("Pause and resume unit test", function() {
62     it("pause and resume contract test", async function() {
63         const {Owner, Alice, proxy} = await loadFixture(deployAndBindFixture);
64         // only Owner
65         await expect(proxy.connect(Alice).pause()).to.revertedWith("Unauthorized");
66         await expect(proxy.connect(Alice).resume()).to.revertedWith("Unauthorized");
67         // pause emit event
68         await expect(proxy.pause()).to.emit(
69             proxy, "VaultPaused"
70         ).withArgs(Owner.address);
71         expect(await proxy.vaultPaused()).to.equal(true);
72         // pause twice should be failed
73         await expect(proxy.pause()).to.revertedWith("Vault is already paused");
74
75         // resume emit event
76         await expect(proxy.resume()).to.emit(
77             proxy, "VaultResumed"
78         ).withArgs(Owner.address);
79         expect(await proxy.vaultPaused()).to.equal(false);
80         // resume twice should be failed
81         await expect(proxy.resume()).to.revertedWith("Vault is not paused");
82     });
83 });
84
85 describe("Change admin and implement test", function() {
86     it("only admin can change admin or implement", async function() {
87         const {Alice, proxy, users} = await loadFixture(deployAndBindFixture);
88         const XSVVaultProxy = await ethers.getContractFactory("XSVVaultProxy");
89         let instance = XSVVaultProxy.attach(proxy.address);
90         await expect(instance.connect(Alice)._setPendingAdmin(users[0].address)).to.emit(
91             instance, "Failure"
92         ).withArgs(1, 2, 0);
93         expect(await proxy.pendingAdmin()).to.equal(zero_address);
94
95         await expect(instance.connect(Alice)._setPendingImplementation(users[0].address)).to.emit(
96             instance, "Failure"
97         ).withArgs(1, 3, 0);
98         expect(await proxy.pendingXSVVaultImplementation()).to.equal(zero_address);
99     });
100
101     it("only pending can accept", async function() {
102         const {Owner, Alice, Bob, proxy, vault} = await loadFixture(deployAndBindFixture);
103         const XSVVaultProxy = await ethers.getContractFactory("XSVVaultProxy");
104         let instance = XSVVaultProxy.attach(proxy.address);
105
106         // set pending

```

```

107     await expect(instance._setPendingAdmin(Alice.address)).to.emit(
108         instance, "NewPendingAdmin"
109     ).withArgs(zero_address, Alice.address);
110     expect(await proxy.pendingAdmin()).to.equal(Alice.address);
111     await expect(instance._setPendingImplementation(Bob.address)).to.emit(
112         instance, "NewPendingImplementation"
113     ).withArgs(zero_address, Bob.address);
114     expect(await proxy.pendingXVSVaultImplementation()).to.equal(Bob.address);
115
116     // accept without pending
117     await expect(instance._acceptAdmin()).to.emit(
118         instance, "Failure"
119     ).withArgs(1, 0, 0);
120     expect(await proxy.pendingAdmin()).to.equal(Alice.address);
121     await expect(instance._acceptImplementation()).to.emit(
122         instance, "Failure"
123     ).withArgs(1, 1, 0);
124     expect(await proxy.pendingXVSVaultImplementation()).to.equal(Bob.address);
125
126     // accept with pending
127     await expect(instance.connect(Alice)._acceptAdmin()).to.emit(
128         instance, "NewAdmin"
129     ).withArgs(Owner.address, Alice.address);
130
131     await expect(instance.connect(Bob)._acceptImplementation()).to.emit(
132         instance, "NewImplementation"
133     ).withArgs(vault.address, Bob.address);
134
135     // check final state
136     expect(await proxy.admin()).to.equal(Alice.address);
137     expect(await proxy.pendingAdmin()).to.equal(zero_address);
138     expect(await proxy.implementation()).to.equal(Bob.address);
139     expect(await proxy.pendingXVSVaultImplementation()).to.equal(zero_address);
140 });
141 });
142
143 describe("Add pool unit test", function() {
144     it("only user with access_allowed can add pool", async function() {
145         const {Alice, token_a, xvs, proxy, vai} = await loadFixture(deployAndBindFixture);
146         await
147     expect(proxy.connect(Alice).add(xvs.address, 50, token_a.address, 50, 300)).to.revertedWith("Unauthorized");
148     });
149
150     it("Add pool should change state and emit event", async function() {
151         const {token_a, token_b, xvs, vai, proxy, store} = await loadFixture(deployAndBindFixture);
152         await expect(proxy.add(xvs.address, 50, token_a.address, 100, 300)).to.emit(
153             proxy, "PoolAdded"
154         ).withArgs(xvs.address, 0, token_a.address, 50, 100, 300);
155         let block = await time.latestBlock();
156         expect(await store.rewardTokens(xvs.address)).to.equal(true);
157         let {token, allocPoint, lastRewardBlock, accRewardPerShare, lockPeriod} = await
158     proxy.poolInfos(xvs.address, 0);
159         expect(token).to.equal(token_a.address);
160         expect(allocPoint).to.equal(50);
161         expect(lastRewardBlock).to.equal(block);
162         expect(accRewardPerShare).to.equal(0);
163         expect(lockPeriod).to.equal(300);
164
165         expect(await proxy.rewardTokenAmountsPerBlock(xvs.address)).to.equal(100);
166         expect(await proxy.totalAllocPoints(xvs.address)).to.equal(50);
167         expect(await proxy.poolLength(xvs.address)).to.equal(1);
168
169         // add same should be failed
170         await expect(proxy.add(xvs.address, 50, token_a.address, 100, 300)).to.revertedWith("Pool already
171     added");
172
173         // add token_b
174         await expect(proxy.add(xvs.address, 50, token_b.address, 200, 300)).to.emit(
175             proxy, "PoolAdded"
176         ).withArgs(xvs.address, 1, token_b.address, 50, 200, 300);
177
178         block = await time.latestBlock();
179         ({token, allocPoint, lastRewardBlock, accRewardPerShare, lockPeriod} = await
180     proxy.poolInfos(xvs.address, 1));

```

```

176     expect(token).to.equal(token_b.address);
177     expect(allocPoint).to.equal(50);
178     expect(lastRewardBlock).to.equal(block);
179     expect(accRewardPerShare).to.equal(0);
180     expect(lockPeriod).to.equal(300);
181
182     expect(await proxy.rewardTokenAmountsPerBlock(xvs.address)).to.equal(200);
183     expect(await proxy.totalAllocPoints(xvs.address)).to.equal(100);
184     expect(await proxy.poolLength(xvs.address)).to.equal(2);
185
186     // check token_a
187     ({token, allocPoint, lastRewardBlock, accRewardPerShare, lockPeriod} = await
proxy.poolInfos(xvs.address, 0));
188     expect(token).to.equal(token_a.address);
189     expect(allocPoint).to.equal(50);
190     expect(lastRewardBlock).to.equal(block);
191     expect(accRewardPerShare).to.equal(0);
192     expect(lockPeriod).to.equal(300);
193
194     // add another reward token;
195     await expect(proxy.add(token_a.address, 50, xvs.address, 100, 300)).to.emit(
proxy, "PoolAdded"
196     ).withArgs(token_a.address, 0, xvs.address, 50, 100, 300);
197
198     // add same deposit token
199     await expect(proxy.add(vai.address, 50, token_a.address, 100, 300)).to.revertedWith("Token exists in
other pool");
200
201     });
202   });
203
204   describe("Reset _allocPoint unit test", function() {
205     it("only user with access_allowed can reset pool", async function() {
206       const {Alice, xvs, proxy} = await loadFixture(deployAndBindFixture);
207       await expect(proxy.connect(Alice).set(xvs.address, 50, 300)).to.revertedWith("Unauthorized");
208     });
209     it("only valid pool can be reset", async function() {
210       const {xvs, proxy} = await loadFixture(deployAndBindFixture);
211       await expect(proxy.set(xvs.address, 50, 300)).to.revertedWith("vault: pool exists?");
212     });
213
214     it("Reset must update pool", async function() {
215       const {token_a, token_b, xvs, proxy} = await loadFixture(deployAndBindFixture);
216       await expect(proxy.add(xvs.address, 50, token_a.address, 100, 300)).to.emit(
proxy, "PoolAdded"
217       ).withArgs(xvs.address, 0, token_a.address, 50, 100, 300);
218       await expect(proxy.add(xvs.address, 50, token_b.address, 100, 300)).to.emit(
proxy, "PoolAdded"
219       ).withArgs(xvs.address, 1, token_b.address, 50, 100, 300);
220       // reset
221       expect(await proxy.totalAllocPoints(xvs.address)).to.equal(100);
222       await expect(proxy.set(xvs.address, 0, 100)).to.emit(
proxy, "PoolUpdated"
223       ).withArgs(xvs.address, 0, 50, 100);
224       // check state
225       let block = await time.latestBlock();
226       let {token, allocPoint, lastRewardBlock, accRewardPerShare, lockPeriod} = await
proxy.poolInfos(xvs.address, 0);
227       expect(token).to.equal(token_a.address);
228       expect(allocPoint).to.equal(100);
229       expect(lastRewardBlock).to.equal(block);
230       expect(accRewardPerShare).to.equal(0);
231       expect(lockPeriod).to.equal(300);
232       expect(await proxy.totalAllocPoints(xvs.address)).to.equal(150);
233     });
234   });
235
236   describe("setRewardAmountPerBlock unit test", function() {
237     it("only user with access_allowed can reset reward", async function() {
238       const {Alice, xvs, proxy} = await loadFixture(deployAndBindFixture);
239       await
expect(proxy.connect(Alice).setRewardAmountPerBlock(xvs.address, 300)).to.revertedWith("Unauthorized");
240     });
241   });
242
243
244

```



```

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
267 expect(proxy.connect(Alice).setWithdrawalLockingPeriod(xvs.address,0,300)).to.revertedWith("Unauthorized");
268     });
269
270     it("Pool must be valid", async function() {
271         const {xvs,proxy} = await loadFixture(deployAndBindFixture);
272         await expect(proxy.setWithdrawalLockingPeriod(xvs.address,0,300)).to.revertedWith("vault: pool
273 exists?");
274     });
275
276     it("period must be greater than zero", async function() {
277         const {token_a,xvs,proxy} = await loadFixture(deployAndBindFixture);
278         await expect(proxy.add(xvs.address,50,token_a.address,100,300)).to.emit(
279             proxy,"PoolAdded"
280         ).withArgs(xvs.address,0,token_a.address,50,100,300);
281         await expect(proxy.setWithdrawalLockingPeriod(xvs.address,0,0)).to.revertedWith("Invalid new
282 locking period");
283     });
284
285     it("Reset should change state and emit event", async function() {
286         const {token_a,xvs,proxy} = await loadFixture(deployAndBindFixture);
287         await expect(proxy.add(xvs.address,50,token_a.address,100,300)).to.emit(
288             proxy,"PoolAdded"
289         ).withArgs(xvs.address,0,token_a.address,50,100,300);
290         await expect(proxy.setWithdrawalLockingPeriod(xvs.address,0,400)).to.emit(
291             proxy,"WithdrawalLockingPeriodUpdated"
292         ).withArgs(xvs.address,0,300,400);
293         let block = await time.latestBlock();
294         let info = await proxy.poolInfos(xvs.address,0);
295         expect(info.lastRewardBlock).to.equal(block -1);
296         expect(info.lockPeriod).to.equal(400);
297     });
298 });
299
300 describe("delegate unit test", function() {
301     it("Delegate should emit event", async function() {
302         const {Alice,Bob,proxy} = await loadFixture(deployAndBindFixture);
303         await expect(proxy.connect(Alice).delegate(Bob.address)).to.emit(
304             proxy,"DelegateChangedV2"
305         ).withArgs(Alice.address,zero_address,Bob.address);
306     });
307 });
308
309 describe("Deposit unit test", function() {
310     it("The pool must be valid", async function() {
311         const {xvs,proxy} = await loadFixture(deployAndBindFixture);
312         await expect(proxy.deposit(xvs.address,1,100)).to.revertedWith("vault: pool exists?");
313     });
314
315     it("Deposit should update pool and change user's state", async function() {
316         const {token_a,Alice,Bob,xvs,vai,proxy,store} = await loadFixture(deployAndBindFixture);
317         // prepare
318         await token_a.transfer(Alice.address,100000000);

```



```

315     await token_a.transfer(Bob.address,100000000);
316     await xvs.transfer(Alice.address,100000000);
317     await xvs.transfer(Bob.address,100000000);
318     await token_a.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
319     await xvs.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
320     await token_a.connect(Bob).approve(proxy.address,ethers.constants.MaxUint256);
321     await xvs.connect(Bob).approve(proxy.address,ethers.constants.MaxUint256);
322     await vai.transfer(store.address,ethers.utils.parseEther("100000000"));
323
324     let all_rewards = ethers.utils.parseEther("1");
325     // add pool
326     await expect(proxy.add(vai.address,50,token_a.address,all_rewards,300)).to.emit(
327       proxy,"PoolAdded"
328     ).withArgs(vai.address,0,token_a.address,50,all_rewards,300);
329     await expect(proxy.add(vai.address,100,xvs.address,all_rewards,300)).to.emit(
330       proxy,"PoolAdded"
331     ).withArgs(vai.address,1,xvs.address,100,all_rewards,300);
332     let block = await time.latestBlock();
333     // Alice deposit token_a;
334     await expect(proxy.connect(Alice).deposit(vai.address,0,10000)).to.emit(
335       proxy,"Deposit"
336     ).withArgs(Alice.address,vai.address,0,10000);
337
338     let alice_info = await proxy.getUserInfo(vai.address,0,Alice.address);
339     expect(alice_info.amount).equal(10000);
340     expect(alice_info.rewardDebt).equal(0);
341     expect(alice_info.pendingWithdrawals).equal(0);
342     let pool_info = await proxy.poolInfos(vai.address,0);
343     expect(pool_info.lastRewardBlock).to.eq(block + 1);
344     expect(pool_info.accRewardPerShare).to.eq(0);
345     // Bob deposit token_a;
346     await expect(proxy.connect(Bob).deposit(vai.address,0,10000)).to.emit(
347       proxy,"Deposit"
348     ).withArgs(Bob.address,vai.address,0,10000);
349
350     pool_info = await proxy.poolInfos(vai.address,0);
351     expect(pool_info.lastRewardBlock).to.eq(block + 2);
352     let reward_pershare = all_rewards.mul(50).div(150).mul(GAIN).div(10000);
353     expect(pool_info.accRewardPerShare).to.eq(reward_pershare);
354     // Alice deposit again;
355     let offset = all_rewards.mul(50).div(150).mul(GAIN).div(20000);
356     reward_pershare = reward_pershare.add(offset);
357     let pending = reward_pershare.mul(10000).div(GAIN);
358     // expect(await proxy.pendingReward(vai.address,0,Alice.address)).to.equal(pending);
359     // claim or deposit should transfer reward
360     await expect(proxy.connect(Alice).claim(Alice.address,vai.address,0)).to.emit(
361       vai,"Transfer"
362     ).withArgs(store.address,Alice.address,pending);
363     await expect(proxy.connect(Alice).deposit(vai.address,0,10000)).to.emit(
364       vai,"Transfer"
365     ).withArgs(store.address,Alice.address,anyValue);
366   });
367 });
368
369
370
371 describe("Claim unit test", function() {
372   it("The pool must be valid", async function() {
373     const {Alice,xvs,proxy} = await loadFixture(deployAndBindFixture);
374     await expect(proxy.claim(Alice.address,xvs.address,1)).to.revertedWith("vault: pool exists?");
375   });
376 });
377
378
379
380 describe("requestWithdrawal unit test", function() {
381   it("The pool must be valid", async function() {
382     const {xvs,proxy} = await loadFixture(deployAndBindFixture);
383     await expect(proxy.requestWithdrawal(xvs.address,1,100)).to.revertedWith("vault: pool exists?");
384   });
385
386   it("requestWithdrawal should emit event and change state", async function() {
387     const {token_a,Alice,Bob,xvs,vai,proxy,store} = await loadFixture(deployAndBindFixture);

```

```

388 // prepare
389 await token_a.transfer(Alice.address,100000000);
390 await token_a.transfer(Bob.address,100000000);
391 await xvs.transfer(Alice.address,100000000);
392 await xvs.transfer(Bob.address,100000000);
393 await token_a.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
394 await xvs.connect(Alice).approve(proxy.address,ethers.constants.MaxUint256);
395 await token_a.connect(Bob).approve(proxy.address,ethers.constants.MaxUint256);
396 await xvs.connect(Bob).approve(proxy.address,ethers.constants.MaxUint256);
397 await vai.transfer(store.address,ethers.utils.parseEther("100000000"))
398 let all_rewards = ethers.utils.parseEther("1");
399 // add pool
400 await expect(proxy.add(vai.address,50,token_a.address,all_rewards,300)).to.emit(
401   proxy,"PoolAdded"
402 ).withArgs(vai.address,0,token_a.address,50,all_rewards,300);
403 await proxy.connect(Alice).deposit(vai.address,0,10000);
404 time.increase(9);
405 await proxy.connect(Alice).deposit(vai.address,0,20000);
406 time.increase(9);
407 // requestWithdrawal
408 let amount = 5000;
409
410 await expect(proxy.connect(Alice).requestWithdrawal(vai.address,0,amount)).to.emit(
411   proxy,"RequestedWithdrawal"
412 ).withArgs(Alice.address,vai.address,0,5000);
413 expect(await proxy.getRequestedAmount(vai.address,0,Alice.address)).to.equal(5000);
414 await time.increase(9);
415 await proxy.connect(Alice).requestWithdrawal(vai.address,0,6000);
416 await time.increase(9);
417 await proxy.setWithdrawalLockingPeriod(vai.address,0,200);
418 await proxy.connect(Alice).requestWithdrawal(vai.address,0,3000);
419 await time.increase(9);
420 await proxy.setWithdrawalLockingPeriod(vai.address,0,250);
421 await proxy.connect(Alice).requestWithdrawal(vai.address,0,2000);
422 // check sort
423 let requests = await proxy.getWithdrawalRequests(vai.address,0,Alice.address);
424 expect(requests.length).to.equal(4);
425 expect(requests[0].amount).to.equal(6000);
426 expect(requests[1].amount).to.equal(5000);
427 expect(requests[2].amount).to.equal(2000);
428 expect(requests[3].amount).to.equal(3000);
429
430
431 expect(await proxy.getRequestedAmount(vai.address,0,Alice.address)).to.equal(6000 + 5000 + 3000 +
2000);
432 expect(await proxy.pendingWithdrawalsBeforeUpgrade(vai.address,0,Alice.address)).to.equal(0);
433
434 let user_info = await proxy.getUserInfo(vai.address,0,Alice.address);
435 expect(user_info.amount).to.equal(30000);
436 expect(user_info.pendingWithdrawals).to.equal(16000);
437 expect(await proxy.getRequestedAmount(vai.address,0,Alice.address)).to.equal(16000);
438 await time.increase(253);
439 let can_withdraw = await proxy.getEligibleWithdrawalAmount(vai.address,0,Alice.address);
440 expect(can_withdraw).to.equal(5000);
441
442 // exec withdraw
443 await expect(proxy.connect(Alice).executeWithdrawal(vai.address,0)).to.emit(
444   proxy,"ExecutedWithdrawal"
445 ).withArgs(Alice.address,vai.address,0,5000);
446
447 requests = await proxy.getWithdrawalRequests(vai.address,0,Alice.address);
448 expect(requests.length).to.equal(2);
449
450 user_info = await proxy.getUserInfo(vai.address,0,Alice.address);
451 expect(user_info.amount).to.equal(25000);
452 expect(user_info.pendingWithdrawals).to.equal(11000);
453 });
454 });
455
456 describe("Deposit | withdraw with delegates", function() {
457   it("delegate before deposit", async function() {
458     const {Alice,Bob,xvs,proxy,store,users} = await loadFixture(deployAndBindFixture);
459     // delegate first;

```

```

460         await proxy.connect(Alice).delegate(Bob.address);
461
462         await xvs.transfer(Alice.address,10000000);
463         await xvs.transfer(store.address,ethers.utils.parseEther("10000"));
464         await xvs.connect(Alice).approve(proxy.address,ethers.constants.MaxInt256);
465         // add pool
466         await proxy.add(xvs.address,100,xvs.address,ethers.utils.parseEther("1.0"),200);
467
468         // deposit
469         await expect(proxy.connect(Alice).deposit(xvs.address,0,10000)).to.emit(
470             proxy,"DelegateVotesChangedV2"
471         ).withArgs(Bob.address,0,10000);
472         await expect(proxy.connect(Alice).deposit(xvs.address,0,10000)).to.emit(
473             proxy,"DelegateVotesChangedV2"
474         ).withArgs(Bob.address,10000,20000);
475         let block = await time.latestBlock();
476         expect(await proxy.getCurrentVotes(Bob.address)).to.equal(20000);
477         expect(await proxy.getCurrentVotes(Alice.address)).to.equal(0);
478         expect(await proxy.getPriorVotes(Bob.address,block-1)).to.equal(10000);
479         // withdraw
480         await proxy.connect(Alice).requestWithdrawal(xvs.address,0,2000);
481         expect(await proxy.getPriorVotes(Bob.address,block)).to.equal(20000);
482         expect(await proxy.getCurrentVotes(Bob.address)).to.equal(18000);
483     });
484
485     it("Delegate after deposit", async function() {
486         const {Alice,Bob,xvs,proxy,store,users} = await loadFixture(deployAndBindFixture);
487         await xvs.transfer(Alice.address,10000000);
488         await xvs.transfer(store.address,ethers.utils.parseEther("10000"));
489         await xvs.connect(Alice).approve(proxy.address,ethers.constants.MaxInt256);
490         // add pool
491         await proxy.add(xvs.address,100,xvs.address,ethers.utils.parseEther("1.0"),200);
492         // deposit
493         await proxy.connect(Alice).deposit(xvs.address,0,10000);
494         expect(await proxy.getCurrentVotes(Alice.address)).to.equal(0);
495         // delegate
496         await proxy.connect(Alice).delegate(Bob.address);
497         expect(await proxy.getCurrentVotes(Alice.address)).to.equal(0);
498         expect(await proxy.getCurrentVotes(Bob.address)).to.equal(10000);
499         // withdraw
500         let block = await time.latestBlock();
501         await proxy.connect(Alice).requestWithdrawal(xvs.address,0,2000);
502         expect(await proxy.getPriorVotes(Bob.address,block)).to.equal(10000);
503         expect(await proxy.getCurrentVotes(Bob.address)).to.equal(8000);
504     });
505 });
506 });
507

```

5. MockUpgrade.js

```

1  const {
2      time,
3      getStorageAt,
4      loadFixture,
5  } = require("@nomicfoundation/hardhat-network-helpers");
6
7  const { expect } = require("chai");
8  const { ethers } = require("hardhat");
9
10 describe("MockSlot Upgrade Unit Test", function () {
11     async function deployFixture() {
12         const CustomMinERC1967Proxy = await ethers.getContractFactory("CustomMinERC1967Proxy");
13         const MockSlot = await ethers.getContractFactory("MockSlot");
14         const old_impl = await MockSlot.deploy();
15         let instance = await CustomMinERC1967Proxy.deploy(old_impl.address);
16         instance = MockSlot.attach(instance.address);
17         const MockSlotNew = await ethers.getContractFactory("MockSlotNew");
18         return {
19             instance,MockSlotNew,CustomMinERC1967Proxy
20         };
21     };
22

```

```

21   }
22
23   it("Upgrade should not change slot", async () => {
24     let {instance, MockSlotNew, CustomMinERC1967Proxy} = await loadFixture(deployFixture);
25     let param = {
26       amount: 600,
27       lockedUntil: 9876543210
28     };
29     await instance.setRequest(param, 5);
30     let request = await instance.getRequest(5);
31     expect(request[0].amount).to.eq(param.amount);
32     expect(request[0].lockedUntil).to.eq(param.lockedUntil);
33     // calculate slot
34     let start = ethers.utils.solidityKeccak256(["uint256", "uint256"], [5, 0]);
35     let middle = ethers.utils.solidityKeccak256(["bytes32"], [start]);
36     let next = ethers.BigNumber.from(middle).add(1);
37
38     let info_0 = await getStorageAt(instance.address, middle);
39     let amount = ethers.BigNumber.from(info_0);
40     let info_1 = await getStorageAt(instance.address, next);
41     let lockedUntil = ethers.BigNumber.from(info_1);
42     expect(amount).to.eq(param.amount);
43     expect(lockedUntil).to.eq(param.lockedUntil);
44
45     // upgrade
46     let new_impl = await MockSlotNew.deploy();
47     instance = CustomMinERC1967Proxy.attach(instance.address);
48     await instance.updateTo(new_impl.address);
49     instance = MockSlotNew.attach(instance.address);
50     // check slot
51     info_0 = await getStorageAt(instance.address, middle);
52     amount = ethers.BigNumber.from(info_0);
53     info_1 = await getStorageAt(instance.address, next);
54     lockedUntil = ethers.BigNumber.from(info_1);
55     expect(amount).to.eq(param.amount);
56     expect(lockedUntil).to.eq(param.lockedUntil);
57     let info_2 = await getStorageAt(instance.address, next.add(1));
58     expect(info_2).to.eq(ethers.constants.HashZero);
59
60     // check state
61     request = await instance.getRequest(5);
62     expect(request[0].amount).to.eq(param.amount);
63     expect(request[0].lockedUntil).to.eq(param.lockedUntil);
64     expect(request[0].afterUpgrade).to.eq(0);
65
66     // add new struct
67     await instance.setRequest({
68       amount: 600,
69       lockedUntil: 9876543210,
70       afterUpgrade: 1
71     }, 5);
72     // check old slot
73     info_0 = await getStorageAt(instance.address, middle);
74     amount = ethers.BigNumber.from(info_0);
75     info_1 = await getStorageAt(instance.address, next);
76     lockedUntil = ethers.BigNumber.from(info_1);
77     expect(amount).to.eq(param.amount);
78     expect(lockedUntil).to.eq(param.lockedUntil);
79     // check new slot
80     info_0 = await getStorageAt(instance.address, next.add(1));
81     amount = ethers.BigNumber.from(info_0);
82     info_1 = await getStorageAt(instance.address, next.add(2));
83     lockedUntil = ethers.BigNumber.from("0x" + info_1.substring(34));
84     let afterUpgrade = ethers.BigNumber.from(info_1.substring(0, 34));
85     expect(amount).to.eq(param.amount);
86     expect(lockedUntil).to.eq(param.lockedUntil);
87     expect(afterUpgrade).to.eq(1);
88
89     // check state
90     request = await instance.getRequest(5);
91     expect(request[0].amount).to.eq(param.amount);
92     expect(request[0].lockedUntil).to.eq(param.lockedUntil);
93     expect(request[0].afterUpgrade).to.eq(0);

```

```

94
95     expect(request[1].amount).to.eq(param.amount);
96     expect(request[1].lockedUntil).to.eq(param.lockedUntil);
97     expect(request[1].afterUpgrade).to.eq(1);
98
99     let length_info = await getStorageAt(instance.address, start);
100     let length = ethers.BigNumber.from(length_info);
101     expect(length).to.eq(2);
102   });
103 });
104

```

6. Upgrade.js

```

1  const {
2    time,
3    loadFixture,
4  } = require("@nomicfoundation/hardhat-network-helpers");
5  const { anyValue } = require("@nomicfoundation/hardhat-chai-matchers/withArgs");
6  const { expect } = require("chai");
7  const { ethers } = require("hardhat");
8
9  const zero_address = ethers.constants.AddressZero;
10 const GAIN = ethers.utils.parseUnits("1.0", 12);
11
12 describe("XVSVault Upgrade Unit Test", function () {
13   async function deployAndBindFixture() {
14     // get users;
15     const [Owner, Alice, Bob, ...users] = await ethers.getSigners();
16     // Deploy old_impl
17     const XVSVaultOld = await ethers.getContractFactory("XVSVaultOld");
18     const vault = await XVSVaultOld.deploy();
19     // deploy proxy
20     const XVSVaultProxy = await ethers.getContractFactory("XVSVaultProxy");
21     let proxy = await XVSVaultProxy.deploy();
22     // deploy XVSSStore
23     const XVSSStore = await ethers.getContractFactory("XVSSStore");
24     let store = await XVSSStore.deploy();
25     await store.setNewOwner(proxy.address);
26     // binding
27     await proxy._setPendingImplementation(vault.address);
28     await vault._become(proxy.address);
29     proxy = XVSVaultOld.attach(proxy.address);
30     // deploy token
31     const MockERC20 = await ethers.getContractFactory("MockERC20");
32     const xvs = await MockERC20.deploy("XVS", "XVS", ethers.utils.parseEther("100000000"));
33     const vai = await MockERC20.deploy("VAI", "VAI", ethers.utils.parseEther("100000000"));
34     const token_a = await MockERC20.deploy("TokenA", "TAT", ethers.utils.parseEther("100000000"));
35     const token_b = await MockERC20.deploy("TokenB", "TBT", ethers.utils.parseEther("100000000"));
36     await proxy.setXvsStore(xvs.address, store.address);
37     // return
38     return {
39       Owner, Alice, Bob, users, proxy, vault, xvs, vai, store, token_a, token_b
40     };
41   }
42
43   it("Add a pool and deposit", async function() {
44     let {token_a, Alice, Bob, xvs, vai, proxy, store} = await loadFixture(deployAndBindFixture);
45     // prepare
46     await token_a.transfer(Alice.address, 100000000);
47     await token_a.transfer(Bob.address, 100000000);
48     await xvs.transfer(Alice.address, 100000000);
49     await xvs.transfer(Bob.address, 100000000);
50     await token_a.connect(Alice).approve(proxy.address, ethers.constants.MaxUint256);
51     await xvs.connect(Alice).approve(proxy.address, ethers.constants.MaxUint256);
52     await token_a.connect(Bob).approve(proxy.address, ethers.constants.MaxUint256);
53     await xvs.connect(Bob).approve(proxy.address, ethers.constants.MaxUint256);
54     await vai.transfer(store.address, ethers.utils.parseEther("100000000"));
55
56     let all_rewards = ethers.utils.parseEther("1");
57     // add pool

```

```

58     await expect(proxy.add(vai.address,50,token_a.address,all_rewards,300)).to.emit(
59       proxy, "PoolAdded"
60     ).withArgs(vai.address,0,token_a.address,50,all_rewards,300);
61     await expect(proxy.add(vai.address,100,xvs.address,all_rewards,300)).to.emit(
62       proxy, "PoolAdded"
63     ).withArgs(vai.address,1,xvs.address,100,all_rewards,300);
64
65     let block = await time.latestBlock();
66     // Alice deposit token_a;
67     await expect(proxy.connect(Alice).deposit(vai.address,0,10000)).to.emit(
68       proxy, "Deposit"
69     ).withArgs(Alice.address,vai.address,0,10000);
70     let alice_info = await proxy.getUserInfo(vai.address,0,Alice.address);
71     expect(alice_info.amount).equal(10000);
72     expect(alice_info.rewardDebt).equal(0);
73     expect(alice_info.pendingWithdrawals).equal(0);
74     let pool_info = await proxy.poolInfos(vai.address,0);
75     expect(pool_info.lastRewardBlock).to.eq(block + 1);
76     expect(pool_info.accRewardPerShare).to.eq(0);
77     await time.increase(9);
78     await proxy.connect(Alice).requestWithdrawal(vai.address,0,6000);
79     await time.increase(9);
80     let requests = await proxy.getWithdrawalRequests(vai.address,0,Alice.address);
81     console.log(requests);
82
83     // upgrade
84     const XSVVaultNew = await ethers.getContractFactory("XSVVault");
85     const vault = await XSVVaultNew.deploy();
86     const XSVVaultProxy = await ethers.getContractFactory("XSVVaultProxy");
87     proxy = XSVVaultProxy.attach(proxy.address);
88     await proxy._setPendingImplementation(vault.address);
89     await vault._become(proxy.address);
90     proxy = XSVVaultNew.attach(proxy.address);
91
92     requests = await proxy.getWithdrawalRequests(vai.address,0,Alice.address);
93     console.log(requests);
94   });
95 });
96

```

7. UnitTestOutput

```

1  ✓ Check all state vars (1050ms)
2    ✓ pause and resume contract test (202ms)
3    ✓ only admin can change admin or implement (64ms)
4    ✓ only pending can accept (171ms)
5    ✓ only user with access_allowed can add pool
6    ✓ Add pool should change state and emit event (475ms)
7    ✓ only user with access_allowed can reset pool
8    ✓ only valid pool can be reset
9    ✓ Reset must update pool (247ms)
10   ✓ only user with access_allowed can reset reward
11   ✓ set should update pool and change state (174ms)
12   ✓ only user with access_allowed can reset locking period
13   ✓ Pool must be valid
14   ✓ period must be greater than zero (116ms)
15   ✓ Reset should change state and emit event (129ms)
16   ✓ Delegate should emit event
17   ✓ The pool must be valid (38ms)
18   ✓ Deposit should update pool and change user's state (649ms)
19   ✓ The pool must be valid
20   ✓ The pool must be valid
21   ✓ requestWithdrawal should emit event and change state (1188ms)
22   ✓ delegate before deposit (550ms)
23   ✓ Delegate after deposit (446ms)
24   MockSlot Upgrade Unit Test
25     ✓ Upgrade should not change slot (209ms)
26
27   XSVVault Upgrade Unit Test
28   [
29   [

```

```

30     BigNumber { value: "6000" },
31     BigNumber { value: "1684223604" },
32     amount: BigNumber { value: "6000" },
33     lockedUntil: BigNumber { value: "1684223604" }
34   ]
35 ]
36 [
37   [
38     BigNumber { value: "6000" },
39     BigNumber { value: "1684223604" },
40     BigNumber { value: "0" },
41     amount: BigNumber { value: "6000" },
42     lockedUntil: BigNumber { value: "1684223604" },
43     afterUpgrade: BigNumber { value: "0" }
44   ]
45 ]
46   ✓ Add a pool and deposit (968ms)
47
48 VAIVault Unit Test
49   Initial state check
50     ✓ VAIVaultStorage state check (406ms)
51   Change admin and implement test
52     ✓ only admin can change admin or implement (60ms)
53     ✓ only pending can accept (186ms)
54   Pause and resume test
55     ✓ pause and resume contract test (229ms)
56   setVenusInfo test
57     ✓ only admin can set info
58     ✓ VenusInfo can't be reset
59   setAccessControl test
60     ✓ Only admin can set
61     ✓ Can not set to zero address
62     ✓ setAccessControl should change state and emit event (46ms)
63   updatePendingRewards test
64     ✓ updatePending should change records (77ms)
65   Deposit test
66     ✓ Deposit should be failed while paused (55ms)
67     ✓ Deposit should change state and emit event (660ms)
68   Bug Demo
69     ✓ Rewards can convert to xvsBalance (409ms)
70
71 VRTVault Unit Test
72   initial state unit test
73     ✓ Check all state vars (370ms)
74   _setImplementation unit test
75     ✓ _setImplementation only by admin
76     ✓ implementation_ can't be zero address
77     ✓ _setImplementation should change state and emit event
78   initialize unit test
79     ✓ initialize twice should be failed
80   Pause and resume test
81     ✓ pause and resume contract test (156ms)
82   withdrawBep20 unit test
83     ✓ only admin can withdraw (39ms)
84     ✓ Can withdraw vrt (86ms)
85   setLastAccruingBlock unit test
86     ✓ only admin can set
87     ✓ Set should change state and emit event (74ms)
88     ✓ Invalid block setting should be failed (108ms)
89   Deposit | Claim | Withdraw unit test
90     ✓ Deposit | Claim | Withdraw (362ms)
91
92 XVStore Unit Test
93   Initial state check
94     ✓ Check all state after deploy (206ms)
95   setPendingAdmin unit test
96     ✓ only admin can call it
97     ✓ setPendingAdmin should change state and emit event (46ms)
98   acceptAdmin unit test
99     ✓ only pendingAdmin can call it
100     ✓ acceptAdmin should change state and emit event (44ms)
101   setNewOwner
102     ✓ only admin can call it

```



```

103      ✓ set to zero address should be failed
104      ✓ set should change state and emit event (57ms)
105    setRewardToken unit test
106      ✓ Only owner or admin can call (64ms)
107    emergencyRewardWithdraw unit test
108      ✓ only owner can call it
109      ✓ call it will transfer token (92ms)
110    safeRewardTransfer unit test
111      ✓ only owner can call it
112      ✓ call should be failed while not set reward token
113      ✓ Transfer rewards beyond balance test (97ms)
114      ✓ Transfer rewards not beyond balance test (81ms)
115
116
117    65 passing (11s)
118

```

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,BEP20,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,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 | |



<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>

