FAIRYPROOF

ZED20 Network Bridged Tokens

AUDIT REPORT

Version 1.0.0

Serial No. 2024061600012024

Presented by Fairyproof

June 16, 2024

www.fairyproof.com

01. Introduction

This document includes the results of the audit performed by the Fairyproof team on the ZED20 Network Bridged Tokens project.

Audit Start Time:

June 15, 2024

Audit End Time:

June 16, 2024

Audited Source File's Address:

https://bscscan.com/token/0x567556A7493FB7a22d2fd158Dd4C766a98705f96

https://bscscan.com/token/0x4be35ec329343d7d9f548d42b0f8c17fffe07db4

https://bscscan.com/token/0xA4156cc61dc7796faA24278a0F9F229B15e298cb

https://bscscan.com/token/0x40f85D6040dF96ea14cD41142bcd244E14CF76f6

https://bscscan.com/token/0x68Db713779f7470c2FD43D3d06841D0192d44939

The goal of this audit is to review Zedx's solidity implementation for its Token Issuance 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 Zedx team for specified versions. Whenever the code, software, materials, settings, environment etc is changed, the comments of this audit will no longer apply.

— Disclaimer

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

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

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

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

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

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

— Methodology

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

The Fairyproof auditing process follows a routine series of steps:

1. Code Review, Including:

- Project Diagnosis

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

- Manual Code Review

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

- Specification Comparison

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

2. Testing and Automated Analysis, Including:

- Test Coverage Analysis

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

- Symbolic Execution

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

3. Best Practices Review

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

— Structure of the document

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

— Documentation

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

Website:https://zedscan.net/

Whitepaper:https://docs.zedscan.net/

Source Code:

https://bscscan.com/token/0x567556A7493FB7a22d2fd158Dd4C766a98705f96

https://bscscan.com/token/0x4be35ec329343d7d9f548d42b0f8c17fffe07db4

https://bscscan.com/token/0xA4156cc61dc7796faA24278a0F9F229B15e298cb

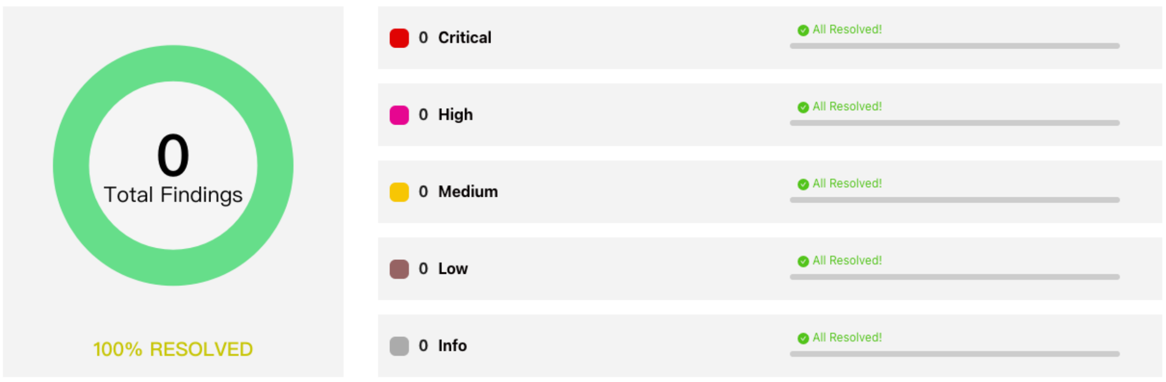
https://bscscan.com/token/0x40f85D6040dF96ea14cD41142bcd244E14CF76f6

https://bscscan.com/token/0x68Db713779f7470c2FD43D3d06841D0192d44939

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

— Comments from Auditor

Serial Number	Auditor	Audit Time	Result
2024061600012024	Fairyproof Security Team	Jun 15, 2024 - Jun 16, 2024	Passed



Summary:

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

02. About Fairyproof

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

03. Introduction to Zedx

Zedx is a decentralized blockchain-powered platform and technology stack whose goal is to enable genuine mass adoption of crypto payments and decentralized finance (DeFi).

The above description is quoted from relevant documents of Zedx.

04. Major functions of audited code

The audited code mainly implements a token issuance function. Here are the details:

USDT.z

- Blockchain: BSC
- Token Standard: ERC-20
- Token Address: 0x4BE35Ec329343d7d9F548d42B0F8c17FFfe07db4
- Token Name: Tether USD Bridged ZED20
- Token Symbol: USDT.z
- Decimals: 18
- Current Supply: 27,500,000,000
- Max Supply: 27,500,000,000

BTC.z

- Blockchain: BSC
- Token Standard: ERC-20
- Token Address: 0xA4156cc61dc7796faA24278a0F9F229B15e298cb
- Token Name: Bitcoin Bridged ZED20
- Token Symbol: BTC.z
- Decimals: 18
- Current Supply: 500,000
- Max Supply: 500,000

USDC.z

- Blockchain: BSC
- Token Standard: ERC-20
- Token Address: 0x40f85D6040dF96ea14cD41142bcd244E14CF76f6
- Token Name: USD Coin Bridged ZED20
- Token Symbol: USDC.z
- Decimals: 18
- Current Supply: 27,500,000,000
- Max Supply: 27,500,000,000

ETH.z

- Blockchain: BSC
- Token Standard: ERC-20
- Token Address: 0x68Db713779f7470c2FD43D3d06841D0192d44939
- Token Name: Ethereum Bridged ZED20
- Token Symbol: ETH.z
- Decimals: 18
- Current Supply: 10,000,000
- Max Supply: 10,000,000

Note:

The above four tokens are bridged tokens that are all minted on issuance. The token supplies are specified by the team.

WZEDX

- Blockchain: BSC
- Token Standard: ERC677
- Token Address: 0x567556A7493FB7a22d2fd158Dd4C766a98705f96
- Token Name: Wrapped ZEDXION from ZEDX
- Token Symbol: WZEDX
- Decimals: 18
- Current Supply: 14,425,603.7726
- Max Supply: No Cap
- Mintable: Yes
- Owner: 0xa3F37D00C47F4B0FDd5acE310632489A21779a6b

Note:

WZEDX is a bridged token which is minted by the bridge contract (owner). The max supply is unlimited and the owner cannot be revoked but can be changed.

The bridge contract is not covered by this audit.

05. Coverage of issues

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

- Access Control
- Admin Rights
- Arithmetic Precision
- Code Improvement
- Contract Upgrade/Migration
- Delete Trap
- Design Vulnerability
- DoS Attack
- EOA Call Trap
- Fake Deposit
- Function Visibility
- Gas Consumption
- Implementation Vulnerability
- Inappropriate Callback Function

- Injection Attack
- Integer Overflow/Underflow
- IsContract Trap
- Miner's Advantage
- Misc
- Price Manipulation
- Proxy selector clashing
- Pseudo Random Number
- Re-entrancy Attack
- Replay Attack
- Rollback Attack
- Shadow Variable
- Slot Conflict
- Token Issuance
- Tx.origin Authentication
- Uninitialized Storage Pointer

06. Severity level reference

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

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

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

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

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

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

07. Major areas that need attention

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

- Function Implementation

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

- Access Control

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

- Token Issuance & Transfer

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

- State Update

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

- Asset Security

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

- Miscellaneous

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

08. issues by severity

- N/A

09. Issue descriptions

- N/A

10. Recommendations to enhance the overall security

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

- N/A

11. Appendices

11.1 Unit Test

1. MockReceiver.sol

```
pragma solidity 0.7.5;


contract MockReceiver {
    event TokenReceived(address indexed from, uint value, bytes data);
    address public source;
    function onTokenTransfer(address from ,uint256 value ,bytes calldata data)
external {
        source = from;
        emit TokenReceived(from,value,data);
    }
}
```

2. TeamToken.t.js

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


  describe("Team Token Unit Test", function () {
    const config = {
        Mintable:false,
        Burnable:false,
        Pausable:false,
        Ownable:false,
        maxApprovalSkip:false,
    }
    const meta = {
        contractName:"TeamToken",
        tokenName:"Tether USD Bridged ZED20",
        tokenSymbol:"USDT.z",
        tokenDecimals:18,
        initSupply: ethers.parseEther("27500000000")
    }

    async function deployTokenFixture() {
        const [owner, alice,bob,...users] = await ethers.getSigners();
        const {contractName,tokenName,tokenSymbol,tokenDecimals,initSupply} =
meta;
        const StandardToken = await ethers.getContractFactory(contractName);
        const instance = await
StandardToken.deploy(tokenName,tokenSymbol,tokenDecimals,initSupply,owner.address
,alice.address);

        check_config(instance);
        return {owner,alice,bob,users,instance};
    }

    function check_config(instance) {
        let functions = instance.interface.fragments
            .filter(item => item.type === "function")
            .map(item => item.name);

        let mint_flag = functions.includes("mint");
        if(config.Mintable !== mint_flag) {
            throw("Invalid Mintable config");
        }

        let burn_flag = functions.includes("burn") &&
functions.includes("burnFrom");
        if(config.Burnable !== burn_flag) {
            throw("Invalid Burnable config");
        }

        let owner_flag = functions.includes("owner")
            && functions.includes("renounceOwnership")
            && functions.includes("transferOwnership");

        if(config.Ownable !== owner_flag) {
            throw("Invalid Ownable config");
        }

        let pause_flag = functions.includes("pause")
            && functions.includes("unpause")
            && functions.includes("paused");
        if(config.Pausable !== pause_flag) {
            throw("Invalid Pausable config");
        }

        if(config.Pausable && !config.Ownable) {
            throw("Please check the calling permission of Pausable");
        }

        if(config.Mintable && !config.Ownable) {
            throw("Please check the calling permission of Mintable");
        }
    }

    function convert(num) {
        return ethers.getBigInt(num);
    }

    describe("Metadata unit Test", function () {
        it("Metadata should be the same as expected", async function() {
            const {instance,owner} = await loadFixture(deployTokenFixture);
            expect(await instance.name()).eq(meta.tokenName,"TokenName does not
match");
            expect(await instance.symbol()).eq(meta.tokenSymbol,"TokenSymbol does
not match");
            expect(await
instance.decimals()).eq(meta.tokenDecimals,"tokenDecimals does not match");
```

```
            expect(await
instance.balanceOf(owner.address)).eq(meta.initSupply,"InitSupply does not
match");
            expect(await instance.totalSupply()).eq(meta.initSupply,"InitSupply
does not match");
        });
    });

    describe("Transfer unit test", function () {
        it("Token transfer should emit event and change balance", async
function() {
            const {instance,owner,alice,bob} = await
loadFixture(deployTokenFixture);
            await expect(instance.transfer(alice.address,1000)).to.be.emit(
                instance,"Transfer"
            ).withArgs(owner.address,alice.address,1000);
            expect(await instance.balanceOf(alice.address)).eq(1000,"Balance of
alice does not match");
            expect(await instance.balanceOf(owner.address)).eq(meta.initSupply -
convert(1000),"Balance of owner does not match");
            expect(await instance.totalSupply()).eq(meta.initSupply,"InitSupply
does not match");
            await instance.connect(alice).transfer(bob.address,400);
            expect(await instance.balanceOf(alice.address)).eq(600,"Balance of
alice does not match while transferring to bob");
            expect(await instance.balanceOf(bob.address)).eq(400,"Balance of bob
does not match");
        });

        it("Should be failed if sender doesn't have enough tokens", async () => {
            const {instance,alice} = await loadFixture(deployTokenFixture);
            await expect(instance.transfer(alice.address,meta.initSupply +
convert(1))).to.be.revertedWith(
                "ERC20: transfer amount exceeds balance"
            );
        });
    });

    describe("Approve unit test", function () {
        it("Approve should change state and emit event", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            expect(await
instance.allowance(alice.address,bob.address)).eq(0,"Allowance0 does not match");

            await
expect(instance.connect(alice).approve(bob.address,10000)).to.be.emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,10000);
            expect(await
instance.allowance(alice.address,bob.address)).eq(10000,"Allowance1 does not
match");

            await
expect(instance.connect(alice).increaseAllowance(bob.address,2000)).to.be.emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,12000);
            expect(await
instance.allowance(alice.address,bob.address)).eq(12000,"Allowance2 does not
match");

            await
expect(instance.connect(alice).decreaseAllowance(bob.address,3000)).to.be.emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,9000);
            expect(await
instance.allowance(alice.address,bob.address)).eq(9000,"Allowance3 does not
match");

        });
    });

    describe("TransferFrom unit test", function () {
        it("Token transferFrom should emit event and change state", async
function() {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            const amount = 1000;
            await instance.approve(alice.address,amount * 10);
            await
expect(instance.connect(alice).transferFrom(owner.address,alice.address,amount)).
to.be.emit(
                instance,"Transfer"
            ).withArgs(owner.address,alice.address,amount);

            expect(await instance.balanceOf(alice.address)).eq(amount,"Balance of
alice does not match");
            expect(await instance.balanceOf(owner.address)).eq(meta.initSupply -
convert(amount),"Balance of owner does not match");
            expect(await instance.totalSupply()).eq(meta.initSupply,"InitSupply
does not match");
            expect(await
instance.allowance(owner.address,alice.address)).eq(amount * 9,"Allowance does
not match");
        });

        it("Should be failed if the sender doesn't have enough approval", async
() => {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            const amount = 1000;
            await instance.approve(alice.address,amount - 1);
            await
expect(instance.connect(alice).transferFrom(owner.address,alice.address,amount)).
to.be.revertedWith(
                "ERC20: transfer amount exceeds allowance"
            );
        });

        if(!meta.maxApprovalSkip) {
            return;
        }
        it("Maximum approval should not change while transferFrom", async () => {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            const amount = 1000;
            await instance.approve(alice.address,ethers.MaxUint256);
            await
instance.connect(alice).transferFrom(owner.address,alice.address,amount);
            expect(await
instance.allowance(owner.address,alice.address)).eq(ethers.MaxUint256,"Allowance
does not match");
        });


    });

    describe("Burnable unit test", function() {
        if(!config.Burnable) {
            return;
        }

        it("Burn should change state and emit event", async () => {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            await instance.transfer(alice.address,10000);

            await expect(instance.connect(alice).burn(4000)).to.emit(
                instance,"Transfer"
            ).withArgs(alice.address,ethers.ZeroAddress,4000);
            expect(await instance.balanceOf(alice.address)).eq(6000,"Balance of
alice does not match");
            expect(await instance.totalSupply()).eq(meta.initSupply -
convert(4000),"InitSupply does not match");
        });

        it("BurnFrom should change allowance", async () => {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            const amount = 1000;
            await instance.approve(alice.address,amount * 10);
            await
expect(instance.connect(alice).burnFrom(owner.address,amount)).to.be.emit(
                instance,"Transfer"
            ).withArgs(owner.address,ethers.ZeroAddress,amount);
            expect(await instance.balanceOf(owner.address)).eq(meta.initSupply -
convert(amount),"Balance of owner does not match");
            expect(await instance.totalSupply()).eq(meta.initSupply -
convert(amount),"InitSupply does not match");
            expect(await
instance.allowance(owner.address,alice.address)).eq(amount * 9,"Allowance does
not match");
        });

        it("Should be failed if burner doesn’t have enough approval", async () =>
{
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            const amount = 1000;
            await instance.approve(alice.address,amount - 1);
            await
expect(instance.connect(alice).burnFrom(owner.address,amount)).to.be.revertedWith
(
                "ERC20: insufficient allowance"
            );
        });

        it("Maximum approval should not change while BurnFrom", async () => {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            const amount = 1000;
            await instance.approve(alice.address,ethers.MaxUint256);
            await instance.connect(alice).burnFrom(owner.address,amount);
            expect(await
instance.allowance(owner.address,alice.address)).eq(ethers.MaxUint256,"Allowance
does not match");
        });
    });

    describe("Ownable unit test", function() {
        if(!config.Ownable) {
            return;
        }

        it("Renounce owner should change state and emit event", async () => {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            expect(await instance.owner()).eq(owner.address,"initial owner does
not match");

            await expect(instance.renounceOwnership()).to.be.emit(
                instance,"OwnershipTransferred"
            ).withArgs(owner.address,ethers.ZeroAddress);

            expect(await instance.owner()).eq(ethers.ZeroAddress,"owner should be
zero");
        });

        it("Change owner should change state and emit event", async () => {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            expect(await instance.owner()).eq(owner.address,"initial owner does
not match");

            await expect(instance.transferOwnership(alice.address)).to.be.emit(
                instance,"OwnershipTransferred"
            ).withArgs(owner.address,alice.address);

            expect(await instance.owner()).eq(alice.address,"owner does not
match");
        });

        it("only old owner can change or renounce owner", async () => {
            const {instance,bob,alice} = await loadFixture(deployTokenFixture);
            await
expect(instance.connect(alice).transferOwnership(bob.address)).to.be.revertedWith
(
                "Ownable: caller is not the owner"
            );
            await
expect(instance.connect(alice).renounceOwnership()).to.be.revertedWith(
                "Ownable: caller is not the owner"
            );
        });
    });


    describe("Mintable unit test", function() {
        if(!config.Mintable) {
            return;
        }

        it("Only owner can mint token", async () => {
            const {instance,bob,alice} = await loadFixture(deployTokenFixture);
            await
expect(instance.connect(alice).mint(bob.address,10000)).to.be.revertedWith(
                "Ownable: caller is not the owner"
            );
        });

        it("mint token can change supply and balance", async () => {
            const {instance,alice} = await loadFixture(deployTokenFixture);
            await expect(instance.mint(alice.address,10000)).to.be.emit(
                instance,"Transfer"
            ).withArgs(ethers.ZeroAddress,alice.address,10000);
            expect(await instance.balanceOf(alice.address)).eq(10000,"Balance of
alice does not match");
            expect(await instance.totalSupply()).eq(meta.initSupply +
convert(10000),"TotalSupply does not match");
        });
    });

    describe("Pausable unit test", function() {
        if(!config.Pausable) {
            return;
        }

        it("Only owner can pause transfer", async () => {
            const {instance,alice} = await loadFixture(deployTokenFixture);
            await expect(instance.connect(alice).pause()).to.be.revertedWith(
                "Ownable: caller is not the owner"
            );

            await expect(instance.connect(alice).unpause()).to.be.revertedWith(
                "Ownable: caller is not the owner"
            );
        });

        it("Pause and unpause should change state and emit event", async () => {
            const {instance,owner} = await loadFixture(deployTokenFixture);
            expect(await instance.paused()).to.be.false;

            await expect(instance.pause()).to.be.emit(
                instance,"Paused"
            ).withArgs(owner.address);

            expect(await instance.paused()).to.be.true;
            await expect(instance.pause()).to.be.revertedWith("Pausable:
paused");
```

```
            await expect(instance.unpause()).to.be.emit(
                instance,"Unpaused"
            ).withArgs(owner.address);

            expect(await instance.paused()).to.be.false;
            await expect(instance.unpause()).to.be.revertedWith("Pausable: not
paused");
        });

        it("TokenTransfer should be failed while paused", async () => {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            await instance.pause();

            await
expect(instance.transfer(alice.address,10000)).to.be.revertedWith(
                "ERC20Pausable: token transfer while paused"
            );

            await instance.approve(alice.address,100000);
            await
expect(instance.connect(alice).transferFrom(owner.address,alice.address,1000))
                .to.be.revertedWith("ERC20Pausable: token transfer while
paused");
        });
    });

  });
```

3. WZEDX.t.js

```
const {
    loadFixture,
} = require("@nomicfoundation/hardhat-toolbox/network-helpers");
const { expect } = require("chai");
const { ethers } = require("hardhat");
const ethSigUtil = require('eth-sig-util');


  describe("WZEDX Token Unit Test", function () {

    async function deployTokenFixture() {
        const [owner, alice,bob,...users] = await ethers.getSigners();
        const PermittableToken = await
ethers.getContractFactory("PermittableToken");
        const impl = await PermittableToken.deploy("TokenB","TokenB",9,56);

        const TokenProxy = await ethers.getContractFactory("TokenProxy");
        let proxy = await
TokenProxy.deploy(impl.target,"TokenA","TokenA",18,31337,owner.address);
        let instance = PermittableToken.attach(proxy.target);

        const MockReceiver = await ethers.getContractFactory("MockReceiver");
        let receiver = await MockReceiver.deploy();

        const MintableToken = await ethers.getContractFactory("MintableToken");
        const mock_token = await MintableToken.deploy();

        return {owner,alice,bob,users,instance,proxy,impl,receiver,mock_token};
    }

    const EIP712Domain = [
        { name: 'name', type: 'string' },
        { name: 'version', type: 'string' },
        { name: 'chainId', type: 'uint256' },
        { name: 'verifyingContract', type: 'address' },
      ];

    async function domainSeparator (name, version, chainId, verifyingContract) {
        return '0x' + ethSigUtil.TypedDataUtils.hashStruct(
          'EIP712Domain',
          { name, version, chainId, verifyingContract },
          { EIP712Domain },
        ).toString('hex');
    }

    describe("Proxy unit test", function() {
        it("implementation unit test", async () => {
            const {proxy,impl} = await loadFixture(deployTokenFixture);
            expect(await proxy.implementation()).eq(impl.target);
        });

        it("getTokenProxyInterfacesVersion unit test", async () => {
            const {proxy} = await loadFixture(deployTokenFixture);
            const [major,minor,patch] = await
proxy.getTokenProxyInterfacesVersion();
            expect(major).eq(1);
            expect(minor).eq(0);
            expect(patch).eq(0);
        });

        it("getTokenInterfacesVersion unit test", async () => {
            const {instance} = await loadFixture(deployTokenFixture);
            const [major,minor,patch] = await
instance.getTokenInterfacesVersion();
            expect(major).eq(2);
            expect(minor).eq(5);
            expect(patch).eq(0);
        });

    });

    describe("Metadata unit test", function () {
        it("Metadata should be the same as expected", async function() {
            const {instance,impl,owner,alice} = await
loadFixture(deployTokenFixture);
            expect(await instance.name()).eq("TokenA","TokenName does not
match");
```

```
            expect(await instance.symbol()).eq("TokenA","TokenSymbol does not
match");
            expect(await instance.decimals()).eq(18,"TokenDecimals does not
match");
            expect(await instance.balanceOf(owner.address)).eq(0,"InitSupply does
not match");
            expect(await instance.totalSupply()).eq(0,"InitSupply does not
match");

            expect(await impl.name()).eq("TokenB","TokenName does not match");
            expect(await impl.symbol()).eq("TokenB","TokenSymbol does not
match");
            expect(await impl.decimals()).eq(9,"TokenDecimals does not match");
            expect(await impl.balanceOf(owner.address)).eq(0,"InitSupply does not
match");
            expect(await impl.totalSupply()).eq(0,"InitSupply does not match");
        });
    });

    describe("Ownable unit test", function() {
        it("Renounce owner should be reverted", async () => {
            const {instance,owner} = await loadFixture(deployTokenFixture);
            expect(await instance.owner()).eq(owner.address,"initial owner does
not match");
            await expect(instance.renounceOwnership()).to.be.reverted;
            expect(await instance.owner()).eq(owner.address,"owner should be
zero");
        });

        it("Change owner should change state and emit event", async () => {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            expect(await instance.owner()).eq(owner.address,"initial owner does
not match");

            await expect(instance.transferOwnership(alice.address)).to.be.emit(
                instance,"OwnershipTransferred"
            ).withArgs(owner.address,alice.address);

            expect(await instance.owner()).eq(alice.address,"owner does not
match");
        });

        it("only old owner can change owner", async () => {
            const {instance,bob,alice} = await loadFixture(deployTokenFixture);
            await
expect(instance.connect(alice).transferOwnership(bob.address)).to.be.reverted;
        });
    });

    describe("Mintable unit test", function() {
        it("Finish mint always be reverted", async () => {
            const {instance,bob,alice} = await loadFixture(deployTokenFixture);
            expect(await instance.mintingFinished()).eq(false);
            await expect(instance.finishMinting()).to.be.reverted;
        });
        it("Only owner can mint token", async () => {
            const {instance,bob,alice} = await loadFixture(deployTokenFixture);
            await
expect(instance.connect(alice).mint(bob.address,10000)).to.be.reverted;
        });

        it("mint token can change supply and balance", async () => {
            const {instance,alice} = await loadFixture(deployTokenFixture);
            await expect(instance.mint(alice.address,10000)).to.be.emit(
                instance,"Transfer(address,address,uint256)"
            ).withArgs(ethers.ZeroAddress,alice.address,10000);
            expect(await instance.balanceOf(alice.address)).eq(10000,"Balance of
alice does not match");
            expect(await instance.totalSupply()).eq(10000,"TotalSupply does not
match");
        });
    });

    describe('BridgeContract unit test', () => {
        it("setBridgeContract unit test", async () => {
            const {instance,owner,alice,impl} = await
loadFixture(deployTokenFixture);
            expect(await instance.bridgeContract()).eq(owner.address);

            expect(await instance.isBridge(owner.address)).eq(true);
            expect(await instance.isBridge(alice.address)).eq(false);

            expect(await instance.isBridge(impl.target)).eq(false);

            await
expect(instance.setBridgeContract(alice.address)).to.be.reverted;
            await instance.setBridgeContract(impl.target);

            expect(await instance.bridgeContract()).eq(impl.target);
            expect(await instance.isBridge(owner.address)).eq(false);
            expect(await instance.isBridge(impl.target)).eq(true);
        });
     });

    describe("Burnable unit test", function() {
        it("Burn should change state and emit event", async () => {
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            await instance.mint(alice.address,10000);

            await expect(instance.connect(alice).burn(4000)).to.emit(
                instance,"Transfer(address,address,uint256)"
            ).withArgs(alice.address,ethers.ZeroAddress,4000);
            expect(await instance.balanceOf(alice.address)).eq(6000,"Balance of
alice does not match");
            expect(await instance.totalSupply()).eq(6000,"InitSupply does not
match");
        });

        it("Should be failed if burner doesn't have enough balance", async () =>
{
            const {instance,owner,alice} = await loadFixture(deployTokenFixture);
            const amount = 1000;
            await instance.mint(alice.address,amount - 1);
            await expect(instance.connect(alice).burn(amount)).to.be.reverted;
        });

        it("transferAndCall should be failed", async () => {
            const {instance,alice} = await loadFixture(deployTokenFixture);
            await
expect(instance.transferAndCall(instance.target,0,"0x12345678")).to.be.reverted;
            await
expect(instance.transferAndCall(ethers.ZeroAddress,0,"0x12345678")).to.be.reverte
d;
            await
expect(instance.transferAndCall(alice.address,10,"0x12345678")).to.be.reverted;
        });

        it("transferAndCall to eoa should be successful", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            await instance.mint(alice.address,10000);
            await
expect(instance.connect(alice).transferAndCall(bob.address,1000,"0x12345678")).to
.be.emit(
                instance,"Transfer(address,address,uint256,bytes)"
            ).withArgs(
                alice.address,bob.address,1000,"0x12345678"
            );

            expect(await instance.balanceOf(alice.address)).eq(9000,"Balance of
alice does not match");
            expect(await instance.balanceOf(bob.address)).eq(1000,"Balance of
alice does not match");
        });

        it("transferAndCall to contract should emit hooks", async () => {
            const {instance,owner,receiver} = await
loadFixture(deployTokenFixture);
            await
expect(instance.transferAndCall(receiver.target,0,"0x12345678")).to.be.emit(
                receiver,"TokenReceived"
            ).withArgs(owner.address,0,"0x12345678");
            expect(await receiver.source()).eq(owner.address);
        });

        it("transfer to bridge contract should emit hooks", async () => {
            const {instance,owner,receiver} = await
loadFixture(deployTokenFixture);
            await instance.mint(owner.address,10000);
            await instance.transfer(receiver.target,1000);
            expect(await instance.balanceOf(receiver.target)).eq(1000,"Balance of
alice does not match");
            expect(await receiver.source()).eq(ethers.ZeroAddress,"Source of
receiver does not match");

            await instance.setBridgeContract(receiver.target);
            await expect(instance.transfer(receiver.target,1000)).to.be.emit(
                receiver,"TokenReceived"
            ).withArgs(owner.address,1000,"0x");
            expect(await instance.balanceOf(receiver.target)).eq(2000,"Balance of
alice does not match");
            expect(await receiver.source()).eq(owner.address,"Source of receiver
does not match");
        });

        it("only owner can claim tokens", async () => {
            const {instance,owner,alice,mock_token} = await
loadFixture(deployTokenFixture);
            await mock_token.mint(instance.target,10000);
            await
expect(instance.connect(alice).claimTokens(mock_token.target,alice.address)).to.b
e.reverted;
            let addr = "0x" + "0".repeat(39) + "1";
            await
expect(instance.claimTokens(mock_token.target,addr)).to.be.emit(
                mock_token,"Transfer"
            ).withArgs(instance.target,addr,10000);

            // can't receive eth
            await expect(owner.sendTransaction({
                to:instance.target,
                value:ethers.parseEther("1.0")
            })).to.be.reverted;

        });
    });

    describe("Approve unit test", function () {
        it("Approve should change state and emit event", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            expect(await
instance.allowance(alice.address,bob.address)).eq(0,"Allowance0 does not match");

            await
expect(instance.connect(alice).approve(bob.address,10000)).to.be.emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,10000);
            expect(await
instance.allowance(alice.address,bob.address)).eq(10000,"Allowance1 does not
match");

            await
expect(instance.connect(alice).increaseAllowance(bob.address,2000)).to.be.emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,12000);
            expect(await
instance.allowance(alice.address,bob.address)).eq(12000,"Allowance2 does not
match");

            await
expect(instance.connect(alice).decreaseAllowance(bob.address,3000)).to.be.emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,9000);
            expect(await
instance.allowance(alice.address,bob.address)).eq(9000,"Allowance3 does not
match");

        });
    });

    describe("PermittableToken unit test", function() {
        const chain_id = 31337;

        it("DOMAIN_SEPARATOR unit test", async () => {
            const {instance} = await loadFixture(deployTokenFixture);
            let separator = await
domainSeparator("TokenA","1",31337,instance.target);
            expect(await instance.DOMAIN_SEPARATOR()).eq(separator);
        });

        it("transferFrom without max approvals", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            await instance.mint(alice.address,1000000);
            await instance.connect(alice).approve(bob.address,10000);
            await
expect(instance.connect(bob).transferFrom(alice.address,bob.address,100)).to.be.e
mit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,9900);
            expect(await
instance.allowance(alice.address,bob.address)).eq(9900,"Allowance does not
match");
            expect(await instance.balanceOf(alice.address)).eq(1000000 - 100);
            expect(await instance.balanceOf(bob.address)).eq(100);

            await
expect(instance.connect(bob).transferFrom(alice.address,bob.address,10000)).to.be
.reverted;
            await
expect(instance.connect(bob).transferFrom(alice.address,bob.address,9900)).to.be.
emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,0);
        });

        it("transferFrom with max approvals", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            await instance.mint(alice.address,1000000);
            await instance.connect(alice).approve(bob.address,ethers.MaxUint256);
            await
expect(instance.connect(bob).transferFrom(alice.address,bob.address,100)).to.be.e
mit(
                instance,"Transfer(address,address,uint256)"
            ).withArgs(alice.address,bob.address,100);
            expect(await
instance.allowance(alice.address,bob.address)).eq(ethers.MaxUint256,"Allowance
does not match");
        });
        it("permit with deadline test", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            const domain = {
                name:"TokenA",
                version:"1",
                chainId:chain_id,
                verifyingContract:instance.target,
            };
            const types ={
                "Permit":[
                    {name:"owner", type:"address"},
                    {name:"spender", type:"address"},
                    {name:"value", type:"uint256"},
                    {name:"nonce", type:"uint256"},
                    {name:"deadline", type:"uint256"},
                ]
            };
            const params = {
                owner:alice.address,
                spender:bob.address,
                value:100000,
                nonce:0,
                deadline: 9876543210
            };
            let signature = await alice.signTypedData(domain,types,params);
            const {v,r,s} = ethers.Signature.from(signature);
            await
expect(instance["permit(address,address,uint256,uint256,uint8,bytes32,bytes32)"](
                params.owner,
                params.spender,
                params.value,
                params.deadline,
                v,
                r,
                s
            )).to.be.emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,100000);
            expect(await
instance.allowance(alice.address,bob.address)).eq(100000,"Allowance does not
match");


            // permit twice should be failed
            await
expect(instance["permit(address,address,uint256,uint256,uint8,bytes32,bytes32)"](
                params.owner,
                params.spender,
                params.value,
                params.deadline,
                v,
                r,
                s
            )).to.be.reverted;
        });
        it("permit with expiry", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            let block = await ethers.provider.getBlockNumber();
            const {timestamp} = await ethers.provider.getBlock(block);
            const domain = {
                name:"TokenA",
                version:"1",
                chainId:chain_id,
                verifyingContract:instance.target,
            };
            const types ={
                "Permit":[
                    {name:"holder", type:"address"},
                    {name:"spender", type:"address"},
                    {name:"nonce", type:"uint256"},
                    {name:"expiry", type:"uint256"},
                    {name:"allowed", type:"bool"},
                ]
            };
            const params = {
                holder:alice.address,
                spender:bob.address,
                nonce:0,
                expiry: timestamp + 5,
                allowed: true
            };
            let signature = await alice.signTypedData(domain,types,params);
            const {v,r,s} = ethers.Signature.from(signature);

            await
expect(instance["permit(address,address,uint256,uint256,bool,uint8,bytes32,bytes3
2)"](
                params.holder,
                params.spender,
                params.nonce,
                params.expiry,
                params.allowed,
                v,
                r,
                s
            )).to.be.emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,ethers.MaxUint256);

            expect(await
instance.expirations(alice.address,bob.address)).eq(timestamp + 5);

            // transfer from
            await
instance.connect(bob).transferFrom(alice.address,bob.address,0);

            await ethers.provider.send("evm_mine",[timestamp + 7]);

            await
expect(instance.connect(bob).transferFrom(alice.address,bob.address,0)).to.be.rev
erted;
```

```
            // reset the approval to transferFrom
            await instance.connect(alice).approve(bob.address,ethers.MaxUint256 -
ethers.getBigInt(1));

            await
instance.connect(bob).transferFrom(alice.address,bob.address,0);
            // expirations not change
            expect(await
instance.expirations(alice.address,bob.address)).eq(timestamp + 5);

        });

        it("permit with expiry and reset by approve", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            let block = await ethers.provider.getBlockNumber();
            const {timestamp} = await ethers.provider.getBlock(block);
            const domain = {
                name:"TokenA",
                version:"1",
                chainId:chain_id,
                verifyingContract:instance.target,
            };
            const types ={
                "Permit":[
                    {name:"holder", type:"address"},
                    {name:"spender", type:"address"},
                    {name:"nonce", type:"uint256"},
                    {name:"expiry", type:"uint256"},
                    {name:"allowed", type:"bool"},
                ]
            };
            const params = {
                holder:alice.address,
                spender:bob.address,
                nonce:0,
                expiry: timestamp + 5,
                allowed: true
            };
            let signature = await alice.signTypedData(domain,types,params);
            const {v,r,s} = ethers.Signature.from(signature);

            await
expect(instance["permit(address,address,uint256,uint256,bool,uint8,bytes32,bytes3
2)"](
                params.holder,
                params.spender,
                params.nonce,
                params.expiry,
                params.allowed,
                v,
                r,
                s
            )).to.be.emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,ethers.MaxUint256);
            expect(await
instance.expirations(alice.address,bob.address)).eq(timestamp + 5);

            // transfer from
            await
instance.connect(bob).transferFrom(alice.address,bob.address,0);

            await ethers.provider.send("evm_mine",[timestamp + 7]);

            await
expect(instance.connect(bob).transferFrom(alice.address,bob.address,0)).to.be.rev
erted;

            // reset the approval to transferFrom
            await instance.connect(alice).approve(bob.address,ethers.MaxUint256);
            // expirations is reset to zero
            expect(await instance.expirations(alice.address,bob.address)).eq(0);

            await
instance.connect(bob).transferFrom(alice.address,bob.address,0);

        });

        it("permit with expiry and reset by permit", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            let block = await ethers.provider.getBlockNumber();
            const {timestamp} = await ethers.provider.getBlock(block);
            const domain = {
                name:"TokenA",
                version:"1",
                chainId:chain_id,
                verifyingContract:instance.target,
            };
            const types ={
                "Permit":[
                    {name:"holder", type:"address"},
                    {name:"spender", type:"address"},
                    {name:"nonce", type:"uint256"},
                    {name:"expiry", type:"uint256"},
                    {name:"allowed", type:"bool"},
                ]
            };
            const params = {
                holder:alice.address,
                spender:bob.address,
                nonce:0,
                expiry: timestamp + 5,
                allowed: true
            };
            let signature = await alice.signTypedData(domain,types,params);
            const {v,r,s} = ethers.Signature.from(signature);

            await
expect(instance["permit(address,address,uint256,uint256,bool,uint8,bytes32,bytes3
2)"](
                params.holder,
                params.spender,
                params.nonce,
                params.expiry,
                params.allowed,
                v,
                r,
                s
            )).to.be.emit(
                instance,"Approval"
            ).withArgs(alice.address,bob.address,ethers.MaxUint256);

            expect(await
instance.expirations(alice.address,bob.address)).eq(timestamp + 5);

            // transfer from
            await
instance.connect(bob).transferFrom(alice.address,bob.address,0);

            await ethers.provider.send("evm_mine",[timestamp + 7]);

            await
expect(instance.connect(bob).transferFrom(alice.address,bob.address,0)).to.be.rev
erted;

            // reset the approval to transferFrom
            {
                const domain = {
                    name:"TokenA",
                    version:"1",
                    chainId:chain_id,
                    verifyingContract:instance.target,
                };
                const types ={
                    "Permit":[
                        {name:"owner", type:"address"},
                        {name:"spender", type:"address"},
                        {name:"value", type:"uint256"},
                        {name:"nonce", type:"uint256"},
                        {name:"deadline", type:"uint256"},
                    ]
                };
                const params = {
                    owner:alice.address,
                    spender:bob.address,
                    value:ethers.MaxUint256,
                    nonce:1,
                    deadline: 9876543210
                };
                let signature = await alice.signTypedData(domain,types,params);
                const {v,r,s} = ethers.Signature.from(signature);
                await
expect(instance["permit(address,address,uint256,uint256,uint8,bytes32,bytes32)"](
                    params.owner,
                    params.spender,
                    params.value,
                    params.deadline,
                    v,
                    r,
                    s
                )).to.be.emit(
                    instance,"Approval"
                ).withArgs(alice.address,bob.address,params.value);
            }
            // expirations is reset to zero
            expect(await instance.expirations(alice.address,bob.address)).eq(0);
            // transfer from successfully
            await
instance.connect(bob).transferFrom(alice.address,bob.address,0);
        });

        it("Alias method push test", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            await instance.mint(alice.address,10000);
            await
expect(instance.connect(alice).push(bob.address,100)).to.be.emit(
                instance,"Transfer(address,address,uint256)"
            ).withArgs(alice.address,bob.address,100);
            expect(await instance.balanceOf(alice.address)).eq(10000 -
100,"Balance of alice does not match");
            expect(await instance.balanceOf(bob.address)).eq(100,"Balance of bob
does not match");
            expect(await instance.totalSupply()).eq(10000);
        });

        it("Alias method pull test", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            await instance.mint(alice.address,10000);
            await instance.connect(alice).approve(bob.address,300);
            await
expect(instance.connect(bob).pull(alice.address,100)).to.be.emit(
                instance,"Transfer(address,address,uint256)"
            ).withArgs(alice.address,bob.address,100);
            expect(await instance.balanceOf(alice.address)).eq(10000 -
100,"Balance of alice does not match");
            expect(await instance.balanceOf(bob.address)).eq(100,"Balance of bob
does not match");
            expect(await instance.totalSupply()).eq(10000);
            expect(await
instance.allowance(alice.address,bob.address)).to.eq(200);
        });

        it("Alias method move test", async () => {
            const {instance,alice,bob,users} = await
loadFixture(deployTokenFixture);
            await instance.mint(alice.address,10000);
            await instance.connect(alice).approve(bob.address,300);
            await
expect(instance.connect(bob).move(alice.address,users[0].address,100)).to.be.emit
(
                instance,"Transfer(address,address,uint256)"
            ).withArgs(alice.address,users[0].address,100);
```

```
            expect(await instance.balanceOf(alice.address)).eq(10000 -
100,"Balance of alice does not match");
            expect(await instance.balanceOf(bob.address)).eq(0,"Balance of bob
does not match");
            expect(await instance.balanceOf(users[0].address)).eq(100,"Balance of
users[0] does not match");
            expect(await instance.totalSupply()).eq(10000);
            expect(await
instance.allowance(alice.address,bob.address)).to.eq(200);
        });
    });

    //  Can be used maliciously
    describe("Transfer zero demo", function() {
        it("transfer zero amounts can be successful", async () => {
            const {instance,alice,bob} = await loadFixture(deployTokenFixture);
            await
expect(instance.connect(alice).transfer(bob.address,0)).to.be.emit(
                instance,"Transfer(address,address,uint256)"
            ).withArgs(alice.address,bob.address,0);
        });
    });
});
```

4. UnitTestOutput

```
  Team Token Unit Test
    Metadata unit Test
      ✓ Metadata should be the same as expected (1056ms)
    Transfer unit test
      ✓ Token transfer should emit event and change balance
      ✓ Should be failed if sender doesn’t have enough tokens
    Approve unit test
      ✓ Approve should change state and emit event
    TransferFrom unit test
      ✓ Token transferFrom should emit event and change state
      ✓ Should be failed if the sender doesn't have enough approval

  WZEDX Token Unit Test
    Proxy unit test
      ✓ implementation unit test (61ms)
      ✓ getTokenProxyInterfacesVersion unit test
      ✓ getTokenInterfacesVersion unit test
    Metadata unit test
      ✓ Metadata should be the same as expected
    Ownable unit test
      ✓ Renounce owner should be reverted
      ✓ Change owner should change state and emit event
      ✓ only old owner can change owner
    Mintable unit test
```

```
      ✓ Finish mint always be reverted
      ✓ Only owner can mint token
      ✓ mint token can change supply and balance
    BridgeContract unit test
      ✓ setBridgeContract unit test
    Burnable unit test
      ✓ Burn should change state and emit event
      ✓ Should be failed if burner doesn't have enough balance
      ✓ transferAndCall should be failed
      ✓ transferAndCall to eoa should be successful
      ✓ transferAndCall to contract should emit hooks
      ✓ transfer to bridge contract should emit hooks
      ✓ only owner can claim tokens
    Approve unit test
      ✓ Approve should change state and emit event
    PermittableToken unit test
      ✓ DOMAIN_SEPARATOR unit test
      ✓ transferFrom without max approvals
      ✓ transferFrom with max approvals
      ✓ permit with deadline test
      ✓ permit with expiry
      ✓ permit with expiry and reset by approve
      ✓ permit with expiry and reset by permit
      ✓ Alias method push test
      ✓ Alias method pull test
      ✓ Alias method move test
    Transfer zero demo
      ✓ transfer zero amounts can be successful


  36 passing (2s)
```

11.2 External Functions Check Points

1. TeamToken.sol_output.md

File: contracts/TeamToken.sol

contract: TeamToken is ERC20

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

Index	Function	StateMutability	Modifier	Param Check	IsUserInterface	Unit Test	Miscellaneous
1	name()	view				Passed	
2	symbol()	view				Passed	
3	decimals()	view				Passed	
4	totalSupply()	view				Passed	

Index	Function	StateMutability	Modifier	Param Check	IsUserInterface	Unit Test	Miscellaneous
5	balanceOf(address)	view				Passed	
6	transfer(address,uint256)			Yes	Yes	Passed	
7	allowance(address,address)	view				Passed	
8	approve(address,uint256)				Yes	Passed	
9	transferFrom(address,address,uint256)			Yes	Yes	Passed	
10	increaseAllowance(address,uint256)				Yes	Passed	
11	decreaseAllowance(address,uint256)				Yes	Passed	

2. TokenProxy.sol_output.md

File: contracts/TokenProxy.sol

contract: TokenProxy is Proxy

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

Index	Function	StateMutability	Modifier	Param Check	IsUserInterface	Unit Test	Miscellaneous
1	implementation()	view				Passed	
2	getTokenProxyInterfacesVersion()	pure				Passed	
3	fallback()	payable				Passed	

File: contracts/PermittableToken.sol

contract: PermittableToken is ERC677BridgeToken

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

Index	Function	StateMutability	Modifier	Param Check	IsUserInterface	Unit Test	Miscellaneous
1	transferFrom(address,address,uint256)			Yes	Yes	Passed	
2	approve(address,uint256)				Yes	Passed	
3	increaseAllowance(address,uint256)				Yes	Passed	
4	push(address,uint256)				Yes	Passed	
5	pull(address,uint256)				Yes	Passed	
6	move(address,address,uint256)				Yes	Passed	
7	permit(address,address,uint256,uint256,bool,uint8,bytes32,bytes32)			Yes	Yes	Passed	
8	permit(address,address,uint256,uint256,uint8,bytes32,bytes32)			Yes	Yes	Passed	
9	bridgeContract()	view				Passed	
10	setBridgeContract(address)		`onlyOwner`		No	Passed	
11	transferAndCall(address,uint256,bytes)		`validRecipient(_to)`	Yes	Yes	Passed	
12	getTokenInterfacesVersion()	pure				Passed	
13	transfer(address,uint256)			Yes	Yes	Passed	
14	isBridge(address)	view				Passed	
15	finishMinting()				No	Passed	Always Reverted
16	renounceOwnership()		`onlyOwner`		No	Passed	Always Reverted
17	claimTokens(address,address)		`onlyOwner`		No	Passed	

Index	Function	StateMutability	Modifier	Param Check	IsUserInterface	Unit Test	Miscellaneous
18	decreaseAllowance(address,uint256)				Yes	Passed	
19	burn(uint256)				Yes	Passed	
20	totalSupply()	view				Passed	
21	balanceOf(address)	view				Passed	
22	mint(address,uint256)		`hasMintPermission`, `canMint`		No	Passed	
23	allowance(address,address)	view				Passed	
24	increaseApproval(address,uint256)				Yes	Passed	
25	decreaseApproval(address,uint256)				Yes	Passed	
26	transferOwnership(address)		`onlyOwner`		No	Passed	

FAIRYPROOF

https://medium.com/@FairyproofT

https://twitter.com/FairyproofT

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

https://t.me/Fairyproof_tech

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