MelosA NFT

AUDIT REPORT

Version 1.0.0

Serial No. 20220927000120027

Presented by Fairyproof

September 27, 2022

01. Introduction

This document includes the results of the audit performed by the Fairyproof team on the MelosA NFT project.

Audit Start Time:

September 15, 2022

Audit End Time:

September 27, 2022

Audited Source Files' Links:

https://rinkeby.etherscan.io/address/0xCB8fc9a2116462e1e248a274E1b97291A5c8b850#code

The goal of this audit is to review Melos's solidity implementation for its NFT 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 Melos 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

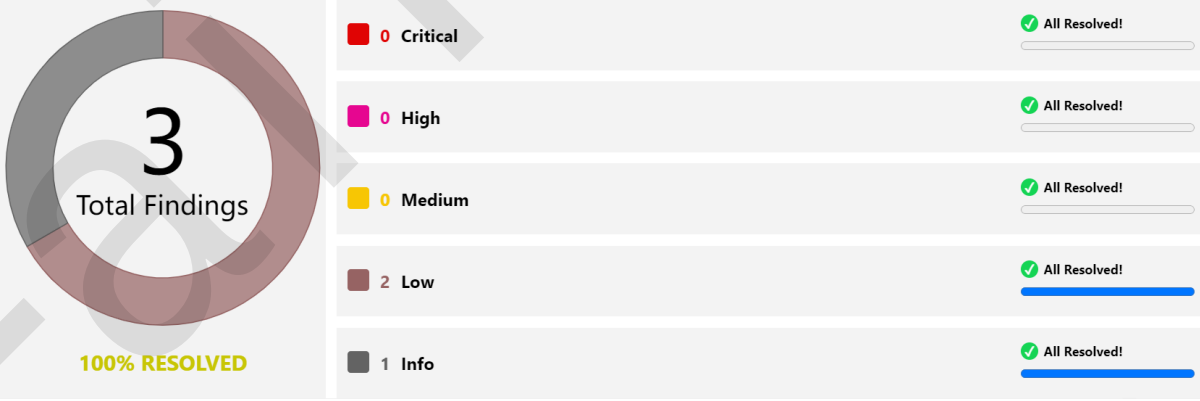
For this audit, we used the following sources of truth about how the NFT issuance system should work:

Contract Source Code

This was considered the specification, and when discrepancies arose with the actual code behavior, we consulted with the Melos team or reported an issue.

— Comments from Auditor

Serial Number	Auditor	Audit Time	Result
2022092700012027	Fairyproof Security Team	Sep 15, 2022 - Sep 27, 2022	Passed



Summary:

The Fairyproof security team used its auto analysis tools and manual work to audit the project. During the audit, two issues of low-severity and one issue of informational-severity were uncovered. The Melos team fixed all the 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. Major functions of audited code

The audited code implements an NFT issuance and here are the details:

NFT Issuance:

- Token Standard: ERC-721A
- Name: Melos A
- Symbol: MELOSA

NFT Purchase:

There is a whitelist. If it is enabled, only whitelisted addessses can mint an NFT. If it is disabled any address can mint an NFT.

The mint price is 0.06 ETHs.

04. Coverage of issues

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

- Re-entrancy Attack
- Replay Attack
- Reordering Attack
- Miner's Advantage
- Rollback Attack

- DDoS Attack
- Injection Attack
- Transaction Ordering Attack
- Race Condition
- Access Control
- Integer Overflow/Underflow
- Timestamp Attack
- Gas Consumption
- Inappropriate Callback Function
- Function Visibility
- Implementation Vulnerability
- Uninitialized Storage Pointer
- Arithmetic Precision
- Tx.origin
- Fake Deposit
- Shadow Variable
- Design Vulnerability
- Token Issuance
- Admin Rights
- Inappropriate Proxy Design
- Inappropriate Use of Slots
- Asset Security
- Contract Upgrade/Migration
- Code Improvement
- Misc

05. 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.

06. List of issues by severity

Index	Title	Issue/Risk	Severity	Status
FP-1	Input Parameters Not Validated	Implementation Vulnerability	Low	✓ Fixed
FP-2	Inappropriate Withdrawal Rights	Design Vulnerability	Low	✓ Fixed
FP-3	Redundant Code	Code Improvement	Info	✓ Fixed

07. Issue descriptions

[FP-1] Input Parameters Not Validated Low ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

In `MelosA.sol`, functions `setMaxMintPerTx`, `setMaxMintPerWallet`, `setMaxSale` and `setMaxTokenSupply` did not check the validility of the input parameters. So `currentMaxSale` could be set greater than `maxTokenSupply` and `maxMintPerTx` could be set greater than `maxMintPerWallet`.

It introduced a logic conflict.

Recommendation:

For functions `setMaxMintPerTx` and `setMaxMintPerWallet`,consider adding the following constraint: maxMintPerTx <= maxMintPerWallet.

For functions `setMaxSale` and `setMaxTokenSupply` , consider adding the the following constraint: currentMaxSale <= maxTokenSupply.

Update/Status:

It has been fixed by the Melos team.

[FP-2] Inappropriate Withdrawal Rights Low ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

In function `withdrawAll`, only `owner` could withdraw assets. When `owner`'s access control was compromised or transferred, the assets in the contract would be permanently locked and couldn't be withdrawn.

Recommendation:

Consider allowing beneficiary addresses to call this function as well.

Update/Status:

It has been fixed by the Melos team.

[FP-3] Redundant Code Info ✓ Fixed

Issue/Risk: Code Improvement

Description:

There was some redundant code:

- The `payable` modifier for `withdraw` was redundant.
- `mintReserve` could be replaced by `airdrop`. So `mintReserve` was redundant.
- Function `saleActive`: since the visibility of `saleEnabled` is public, `saleEnabled` rather than `saleActive` could be called externally.
- Modifier `saleIsOpen`: it was used to apply preconditions to the `mint` function. However `totalSupply <= maxTokenSupply` had checked that in the mint function. And in function `_mintElements`, `require(totalSupply() <= maxTokenSupply, "Melos A: Limit reached");` is adequate.

  So it was redundant and could be removed

Recommendation:

Consider removing all the above redundant code.

Update/Status:

It has been fixed by the Melos team.

08. 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 `owner`'s access control with great care and trasferring it to a multi-sig wallet or DAO when necessary.

09. Appendices

External Functional Check Points

File:MelosA.sol

contract: MelosA is ERC721A, ERC721AQueryable, Ownable

Index	Function	Visibility	Re-entrancy Check	Access Check	Unit Test	Notes
1	mint(uint256,uint256,bytes32[])	public	Passed	No Need	Passed	saleIsOpen
2	isPresaleListClaimed(address)	public	No Need	No Need	Passed	View
3	validClaim(address,uint256,bytes32[])	public	No Need	No Need	Passed	View
4	getPrice(uint256)	public	No Need	No Need	Passed	View
5	saleActive()	external	No Need	No Need	Passed	View
6	airdrop(Drop[])	public	No Need	onlyOwner	Passed	
7	mintReserve(uint256,address)	public	No Need	onlyOwner	Passed	redundant
8	setMaxSale(uint256)	external	No Need	onlyOwner	Passed	
9	setMaxTokenSupply(uint256)	external	No Need	onlyOwner	Passed	
10	setPrice(uint256)	external	No Need	onlyOwner	Passed	
11	setMaxMintPerWallet(uint256)	external	No Need	onlyOwner	Passed	
12	setMaxMintPerTx(uint256)	external	No Need	onlyOwner	Passed	
13	setBaseURI(string)	public	No Need	onlyOwner	Passed	
14	setMerkleRoot(bytes32)	public	No Need	onlyOwner	Passed	
15	toggleSale()	public	No Need	onlyOwner	Passed	
16	toggleWhitelist()	public	No Need	onlyOwner	Passed	
17	withdrawAll()	public	No Need	onlyOwner	Passed	

Unit Test

```
const { MerkleTree } = require('merkletreejs')
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("MelosA Unit Test", function () {
    const sha256 = ethers.utils.keccak256;
    const price = ethers.utils.parseEther("0.06");
    const maxTokenSupply = 5000;
    const currentMaxSale = 5000;
    const maxMintPerTx = 5;
    const maxMintPerWallet = 10;
    const baseURI = "http://www.test.com/";
    let instance;
    let owner,otherAccount,leaves,tree,root;

    async function deployInstance() {
        const [owner, ...otherAccount] = await ethers.getSigners();
        const leaves = otherAccount.slice(5,15).map((user,index) =>
ethers.utils.solidityKeccak256(
            ["address","string"],[user.address,"" + (index + 10)]
        ));
        const tree = new MerkleTree(leaves, sha256,{sortPairs:true});
        const root = tree.getHexRoot();
        const MelosA = await ethers.getContractFactory("MelosA");
        const instance = await MelosA.deploy(baseURI);
        await instance.deployed();
        return {owner,otherAccount,leaves,tree,root,instance};
    }

    beforeEach(async () => {
        ({owner,otherAccount,leaves,tree,root,instance} = await deployInstance());
    });

    describe("Deploy test", function() {
        it("Init state check", async () => {
            expect(await instance.maxTokenSupply()).to.be.equal(maxTokenSupply);
            expect(await instance.baseTokenURI()).to.be.equal(baseURI);
            expect(await
instance.merkleRoot()).to.be.equal(ethers.constants.HashZero);
            expect(await instance.saleEnabled()).to.be.false;
            expect(await instance.whitelistEnabled()).to.be.false;
            expect(await instance.currentPrice()).to.be.equal(price);
            expect(await instance.currentMaxSale()).to.be.equal(currentMaxSale);
            expect(await instance.maxMintPerTx()).to.be.equal(maxMintPerTx);
            expect(await
instance.maxMintPerWallet()).to.be.equal(maxMintPerWallet);
        });
    });

    describe("validClaim test", function() {
        beforeEach(async () => {
            await instance.setMerkleRoot(root);
        });

        it("normal node should pass the check", async () => {
```

```
            let user = otherAccount.slice(5,15)[0].address;
            let leaf = leaves[0];
            let proof = tree.getHexProof(leaf);
            expect(await instance.validClaim(user,10,proof)).to.be.true;
        });

        it("incorrect amount,proof or user should return false", async () => {
            let user = otherAccount.slice(5,15)[0].address;
            let leaf = leaves[0];
            let proof = tree.getHexProof(leaf);
            // incorrect amount
            expect(await instance.validClaim(user,11,proof)).to.be.false;
            // incorrect user
            user = otherAccount.slice(5,15)[1].address;
            expect(await instance.validClaim(user,10,proof)).to.be.false;
            // incorrect proof
            user = otherAccount.slice(5,15)[0].address;
            leaf = leaves[1];
            proof = tree.getHexProof(leaf);
            expect(await instance.validClaim(user,10,proof)).to.be.false;
        });
    });

    describe("getPrice test", function() {
        it("Can get the correct fee", async () => {
            for(let i=1;i<3;i++) {
                let n = i;
                expect(await instance.getPrice(n)).to.be.equal(price.mul(n));
            }
        });
    });

    describe("toggleWhitelist, setBaseURI and toggleSale test", () => {
        it("toggleWhitelist should change state", async () => {
            expect(await instance.whitelistEnabled()).to.be.false;
            await instance.toggleWhitelist();
            expect(await instance.whitelistEnabled()).to.be.true;
            await instance.toggleWhitelist();
            expect(await instance.whitelistEnabled()).to.be.false;
        });

        it("toggleSale should change state", async () => {
            expect(await instance.saleEnabled()).to.be.false;
            await instance.toggleSale();
            expect(await instance.saleEnabled()).to.be.true;
            await instance.toggleSale();
            expect(await instance.saleEnabled()).to.be.false;
        });

        it("setBaseURI should change state", async () => {
            expect(await instance.baseTokenURI()).to.be.equal(baseURI);
            await instance.setBaseURI("http://www.example.com/");
            expect(await
instance.baseTokenURI()).to.be.equal("http://www.example.com/");
            await instance.setBaseURI(baseURI);
```

```
            expect(await instance.baseTokenURI()).to.be.equal(baseURI);
        });
    });

    describe("onlyOwner and setMerkleRoot test", function() {
        it("set without owner should be failed", async () => {
            await expect(instance.connect(otherAccount[0]).setMerkleRoot(root))
                .to.be.revertedWith("Ownable: caller is not the owner");
        });

        it("set with owner should be successful", async () => {
            expect(await
instance.merkleRoot()).to.be.equal(ethers.constants.HashZero);
            await instance.setMerkleRoot(root);
            expect(await instance.merkleRoot()).to.be.equal(root);
        });
    });

    describe("setMaxSale,setMaxTokenSupply,setPrice,setMaxMintPerWallet and
setMaxMintPerTx test", function() {
        it("setMaxSale should change state", async () => {
            expect(await instance.currentMaxSale()).to.be.equal(currentMaxSale);
            await instance.setMaxSale(currentMaxSale - 1);
            expect(await instance.currentMaxSale()).to.be.equal(currentMaxSale -
1);
            await expect(instance.setMaxSale(maxTokenSupply +
1)).to.be.revertedWith("MelosA: currentMaxSale cannot be higher than
maxTokenSupply");
        });

        it("setMaxTokenSupply should change state", async () => {
            expect(await instance.maxTokenSupply()).to.be.equal(maxTokenSupply);
            await instance.setMaxTokenSupply(maxTokenSupply + 1);
            expect(await instance.maxTokenSupply()).to.be.equal(maxTokenSupply +
1);
            await expect(instance.setMaxTokenSupply(currentMaxSale -
1)).to.be.revertedWith("MelosA: maxTokenSupply cannot be lower than
currentMaxSale");
        });

        it("setPrice should change state", async () => {
            expect(await instance.currentPrice()).to.be.equal(price);
            await instance.setPrice(price.mul(2));
            expect(await instance.currentPrice()).to.be.equal(price.mul(2));
        });

        it("setMaxMintPerWallet should change state", async () => {
            expect(await
instance.maxMintPerWallet()).to.be.equal(maxMintPerWallet);
            await instance.setMaxMintPerWallet(maxMintPerWallet + 1);
            expect(await instance.maxMintPerWallet()).to.be.equal(maxMintPerWallet
+ 1);
            await expect(instance.setMaxMintPerWallet(maxMintPerTx -
1)).to.be.revertedWith("MelosA: MaxMintPerWallet cannot be lower than
maxMintPerTx");
```

```
        });

        it("setMaxMintPerTx should change state", async () => {
            expect(await instance.maxMintPerTx()).to.be.equal(maxMintPerTx);
            await instance.setMaxMintPerTx(maxMintPerTx + 1);
            expect(await instance.maxMintPerTx()).to.be.equal(maxMintPerTx + 1);
            await expect(instance.setMaxMintPerTx(maxMintPerWallet +
1)).to.be.revertedWith("MelosA: MaxMintPerTx cannot be higher than
maxMintPerWallet");
        });
    });

    describe("airdrop test", function() {
        let user;
        beforeEach(() => {
            user = otherAccount[0].address;
        });

        it("airdrop should add total supply", async () => {
            expect(await instance.totalSupply()).to.be.equal(0);
            expect(await instance.balanceOf(user)).to.be.equal(0);
            expect(await instance.mintCount(user)).to.be.equal(0);
            let args = [
                {
                    to:user,
                    amount: 10
                }
            ];
            await instance.airdrop(args);
            expect(await instance.totalSupply()).to.be.equal(10);
            expect(await instance.balanceOf(user)).to.be.equal(10);
            expect(await instance.mintCount(user)).to.be.equal(0);
        });

        it("beyond max supply should be failed", async () => {
            await instance.setMaxSale(10);
            await instance.setMaxTokenSupply(10);
            expect(await instance.totalSupply()).to.be.equal(0);
            expect(await instance.balanceOf(user)).to.be.equal(0);
            let args = [
                {
                    to:user,
                    amount: 10
                }
            ];
            await instance.airdrop(args);
            expect(await instance.totalSupply()).to.be.equal(10);
            expect(await instance.balanceOf(user)).to.be.equal(10);
            await expect(instance.airdrop(args)).to.be.revertedWith("MelosA: Limit
reached");
        });
    });

    describe("mint test",function(){
        it("should be failed while beyond maxMintPerWallet", async () => {
```

```
            await expect(instance.mint(maxMintPerWallet + 1,1,
[])).to.be.revertedWith("MelosA: Max wallet limit");
        });

        it("should be failed while beyond maxMintPerTx", async () => {
            await expect(instance.mint(maxMintPerTx + 1,1,
[])).to.be.revertedWith("MelosA: Max mint for tx limit");
        });

        it("it should be failed while sale is disabled", async () => {
            await expect(instance.mint(1,1,[])).to.be.revertedWith("MelosA: Sale
is not active");
        });

        it("Insufficient ethers should be failed", async () => {
            await instance.toggleSale();
            await expect(instance.mint(1,1,[],{
                value:price.div(2)
            })).to.be.revertedWith("MelosA: Value below price");
        });

        it("mint while white list is disabled should change state", async () => {
            expect(await instance.whitelistEnabled()).to.be.false;
            await instance.toggleSale();
            let user = otherAccount[0];
            expect(await instance.totalSupply()).to.be.equal(0);
            expect(await instance.balanceOf(user.address)).to.be.equal(0);
            expect(await instance.mintCount(user.address)).to.be.equal(0);
            await instance.connect(user).mint(3,1,[],{
                value:price.mul(3)
            });
            expect(await instance.totalSupply()).to.be.equal(3);
            expect(await instance.balanceOf(user.address)).to.be.equal(3);
            expect(await instance.mintCount(user.address)).to.be.equal(3);
            expect(await instance.isPresaleListClaimed(user.address)).to.be.false;
        });

        it("merkle root not set should be faild", async() => {
            await instance.toggleSale();
            await instance.toggleWhitelist();
            await expect(instance.mint(1,1,[],{
                value:price
            })).to.be.revertedWith("MelosA: merkle root not set");
        });

        it("beyond max pre-sale amount should be faild", async() => {
            await instance.toggleSale();
            await instance.toggleWhitelist();
            await instance.setMerkleRoot(root);
            await expect(instance.mint(3,1,[],{
                value:price.mul(3)
            })).to.be.revertedWith("MelosA: can only claim less than or equal to
the max amount");
        });
```

```
        it("invalid proof should be failed", async () => {
            let leaf = leaves[1];
            let user = otherAccount.slice(5,15)[0];
            let proof = tree.getHexProof(leaf);
            await instance.toggleSale();
            await instance.toggleWhitelist();
            await instance.setMerkleRoot(root);
            await expect(instance.connect(user).mint(1,10,proof,{
                value:price
            })).to.be.revertedWith("MelosA: invalid proof");
        });

        it("pre-sale mint should change state", async () => {
            let leaf = leaves[0];
            let user = otherAccount.slice(5,15)[0];
            let proof = tree.getHexProof(leaf);
            await instance.toggleSale();
            await instance.toggleWhitelist();
            await instance.setMerkleRoot(root);
            await instance.connect(user).mint(3,10,proof,{
                value:price.mul(3)
            });
            expect(await
ethers.provider.getBalance(instance.address)).to.be.equal(price.mul(3));
            expect(await instance.isPresaleListClaimed(user.address)).to.be.true;
            expect(await instance.totalSupply()).to.be.equal(3);
            expect(await instance.balanceOf(user.address)).to.be.equal(3);
            expect(await instance.mintCount(user.address)).to.be.equal(3);
        })

        it("Replay with same proof should be failed", async () => {
            let leaf = leaves[0];
            let user = otherAccount.slice(5,15)[0];
            let proof = tree.getHexProof(leaf);
            await instance.toggleSale();
            await instance.toggleWhitelist();
            await instance.setMerkleRoot(root);
            await instance.connect(user).mint(3,10,proof,{
                value:price.mul(3)
            });
            await expect(instance.connect(user).mint(3,10,proof,{
                value:price.mul(3)
            })).to.be.revertedWith("MelosA: account already claimed");
        });
    });

    describe("withdrawAll test", () => {
        it("withdarw should be failed while the balace of contract is zero", async
() => {
            expect(await
ethers.provider.getBalance(instance.address)).to.be.equal(0);
            await expect(instance.withdrawAll()).to.be.revertedWith("MelosA:
Balance should be above 0");
        });
```

```
        it("withdraw should change state", async () => {
            const DEV_ADDRESS = "0xea7Ce518713bc6Aec484C2Dc22b359621ed560F5";
            let leaf = leaves[0];
            let user = otherAccount.slice(5,15)[0];
            let proof = tree.getHexProof(leaf);
            await instance.toggleSale();
            await instance.toggleWhitelist();
            await instance.setMerkleRoot(root);
            await instance.connect(user).mint(3,10,proof,{
                value:price.mul(3)
            });
            expect(await ethers.provider.getBalance(DEV_ADDRESS)).to.be.equal(0);
            expect(await
ethers.provider.getBalance(instance.address)).to.be.equal(price.mul(3));
            await
expect(instance.connect(otherAccount[1]).withdrawAll()).to.be.revertedWith("no
permission");
            await instance.withdrawAll();
            expect(await
ethers.provider.getBalance(DEV_ADDRESS)).to.be.equal(price.mul(3));
            expect(await
ethers.provider.getBalance(instance.address)).to.be.equal(0);
        });
    });
});
```

Output:

```
MelosA Unit Test
    Deploy test
      ✓ Init state check (106ms)
    validClaim test
      ✓ normal node should pass the check
      ✓ incorrect amount,proof or user should return false (47ms)
    getPrice test
      ✓ Can get the correct fee
    toggleWhitelist, setBaseURI and toggleSale test
      ✓ toggleWhitelist should change state (42ms)
      ✓ toggleSale should change state (43ms)
      ✓ setBaseURI should change state (45ms)
    onlyOwner and setMerkleRoot test
      ✓ set without owner should be failed (44ms)
      ✓ set with owner should be successful
    setMaxSale,setMaxTokenSupply,setPrice,setMaxMintPerWallet and setMaxMintPerTx
test
      ✓ setMaxSale should change state
      ✓ setMaxTokenSupply should change state
      ✓ setPrice should change state
      ✓ setMaxMintPerWallet should change state
      ✓ setMaxMintPerTx should change state
    airdrop test
      ✓ airdrop should add total supply (60ms)
      ✓ beyond max supply should be failed (85ms)
```

```
  mint test
    ✓ should be failed while beyond maxMintPerWallet
    ✓ should be failed while beyond maxMintPerTx
    ✓ it should be failed while sale is disabled
    ✓ Insufficient ethers should be failed
    ✓ mint while white list is disabled should change state (135ms)
    ✓ merkle root not set should be faild
    ✓ beyond max pre-sale amount should be faild (40ms)
    ✓ invalid proof should be failed (51ms)
    ✓ pre-sale mint should change state (86ms)
    ✓ Replay with same proof should be failed (75ms)
  withdrawAll test
    ✓ withdarw should be failed while the balace of contract is zero
    ✓ withdraw should change state (90ms)


28 passing (4s)
```

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