**Step 1. Create a new npm project**

Navigate to a directory you wish to create this tutorial folder in and run the commands:

|  |
| --- |
| mkdir upgradable-erc20 && cd upgradable-erc20 |

then run

|  |
| --- |
| npm init -y |

**Step 2. Setup Hardhat**

Run the following commands:

|  |
| --- |
| npm install -–save-dev hardhat |

then run

|  |
| --- |
| npx hardhat |

-Choose: “Create an empty hardhat.config.js” from the options given.

**Step 3. Install Hardhat Upgrades, Hardhat Ethers, Ethers, and Hardhat Etherscan.**

Run the following commands:

|  |
| --- |
| npm install --save-dev @openzeppelin/hardhat-upgrades |

then run

|  |
| --- |
| npm install --save-dev @nomicfoundation/hardhat-toolbox |

then install openzeppelin/contracts

|  |
| --- |
| npm install --save-dev @openzeppelin/contracts |

then install openzeppelin/contracts-upgradeable

|  |
| --- |
| npm install --save-dev @openzeppelin/contracts-upgradeable |

Then install dotenv

npm install --save-dev dotenv

**Step 4. Create and configure hardhat.config.js**

In the directory upgradable, open hardhat.config.js and paste the code below.

Your hardhat.config.js may be very different depending on your preferences and network settings.

Example hardhat.config.js file:

|  |
| --- |
| require("@nomicfoundation/hardhat-toolbox");  require("@nomiclabs/hardhat-etherscan");  require("@openzeppelin/hardhat-upgrades");  require("dotenv").config();  module.exports = {    solidity: "0.8.18",    settings: {      optimizer: {        enabled: true,        runs: 200,      },    },    networks: {      hardhat: {        chainId: 1337,      },      sepolia: {        url: process.env.INFURA\_SEPOLIA\_URL,        accounts: [process.env.WALLET\_PRIVATE\_KEY],      },      goerli: {        url: process.env.INFURA\_GOERLI\_URL,        accounts: [process.env.WALLET\_PRIVATE\_KEY],      },    },    etherscan: {      apiKey: process.env.ETHERSCAN\_KEY,    },  }; |

**Step 4. Create and Configure .env**

Create a env file to store your Deployer Wallet Private Key and API key. Paste the below code into it.

Note - Never upload or share your Deployer Wallet Private Key or API key with anyone.

Do not let anyone know your Private Key.

It is highly suggested that you use a separate test wallet for your deployer.

To get your Private key, it depends on the wallet you have, you will need to find tutorials for that via google.

You can get an Infura API key at <https://app.infura.io/dashboard>

You can get an Etherscan API key at <https://etherscan.io/myapikey>

Example .env file:

|  |
| --- |
|  |

**Step 6. Create the ERC20TokenV1.sol upgradeable contract.**

In the directory upgradable create a “contracts” directory.

Create ERC20TokenV1.sol in the “contracts” directory.

Paste the below text into the ERC20TokenV1.sol contract.

|  |
| --- |
| // contracts/ ERC20TokenV1.sol // SPDX-License-Identifier: MIT  pragma solidity ^0.8.6;  import "@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol";  import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";  contract ERC20TokenV1 is Initializable, ERC20Upgradeable {      address public owner;      function initialize(uint \_totalSupply) public initializer {          \_\_ERC20\_init("ERC20 Token", "TKN");          \_mint(msg.sender, \_totalSupply);          owner = msg.sender;      }  } |

**Step 7. Create tests the Contract**

Install the testing package, chai with the command

**npm install chai**

In the directory upgradable create a “test” directory.

In the directory “test” create the file ERC20TokenV1.js

Paste the below text into the ERC20TokenV1.js file

|  |
| --- |
| // test/ERC20TokenV1.js const { expect } = require("chai");  const { ethers, upgrades } = require("hardhat");  let tokenV1;  let deployer;  const tokenName = "ERC20 Token";  const tokenSymbol = "TKN";  const amountToEther = (amount) => {    return ethers.utils.parseEther(amount.toString());  };  const amount = amountToEther(1000000);  describe("ERC20TokenV1 Test", () => {    before(async () => {      const TokenV1 = await ethers.getContractFactory("ERC20TokenV1");      tokenV1 = await upgrades.deployProxy(TokenV1, [amount]);      await tokenV1.deployed();      const account = await ethers.getSigners();      deployer = account[0];    });    describe("Phase 1: ERC20 contract with a pre-minting of 1 million tokens", () => {      it("1. Token should have name", async () => {        const name = await tokenV1.name();        expect(name).to.be.equal(tokenName);      });      it("2. Token should have symbol", async () => {        const symbol = await tokenV1.symbol();        expect(symbol).to.be.equal(tokenSymbol);      });      it("3. Token should have 1 million tokens", async () => {        const totalSupply = await tokenV1.totalSupply();        expect(totalSupply.toString()).to.be.equal(amount.toString());      });      it("4. Deployer should have balance of 10000 tokens", async () => {        const balance = await tokenV1.balanceOf(deployer.address);        expect(balance.toString()).to.be.equal(amount.toString());      });      it("5. Deployer should be token owner", async () => {        const owner = await tokenV1.owner();        expect(owner).to.be.equal(deployer.address);      });    });  }); |

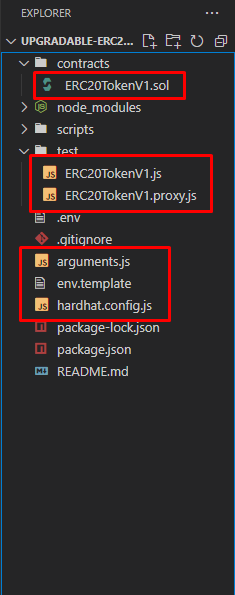
In the directory “test” create the file ERC20TokenV1.proxy.js

Paste the below text into the ERC20TokenV1.proxy.js file

|  |
| --- |
| // test/ERC20TokenV1.proxy.js const { expect } = require("chai");  const { ethers, upgrades } = require("hardhat");  let tokenV1;  let deployer;  const tokenName = "ERC20 Token";  const tokenSymbol = "TKN";  const amountToEther = (amount) => {    return ethers.utils.parseEther(amount.toString());  };  const amount = amountToEther(1000000);  describe("ERC20TokenV1 Test", () => {    before(async () => {      const TokenV1 = await ethers.getContractFactory("ERC20TokenV1");      tokenV1 = await upgrades.deployProxy(TokenV1, [amount]);      const account = await ethers.getSigners();      deployer = account[0];    });    describe("Phase 1: ERC20 contract with a pre-minting of 1 million tokens", () => {      it("1. Token should have name", async () => {        const name = await tokenV1.name();        expect(name).to.be.equal(tokenName);      });      it("2. Token should have symbol", async () => {        const symbol = await tokenV1.symbol();        expect(symbol).to.be.equal(tokenSymbol);      });      it("3. Token should have 1 million tokens", async () => {        const totalSupply = await tokenV1.totalSupply();        expect(totalSupply.toString()).to.be.equal(amount.toString());      });      it("4. Deployer should have balance of 10000 tokens", async () => {        const balance = await tokenV1.balanceOf(deployer.address);        expect(balance.toString()).to.be.equal(amount.toString());      });      it("5. Deployer should be token owner", async () => {        const owner = await tokenV1.owner();        expect(owner).to.be.equal(deployer.address);      });    });  }); |

**Step 8. Verify Folder and File Structure**

So far, if you have been following the tutorial exactly, you should have a folder structure and files open like in the screenshot.

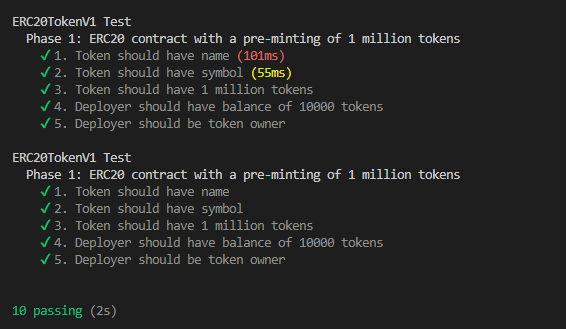


**Step 9. Perform tests**

Now test the contract with the following command

|  |
| --- |
| npx hardhat test |

Both tests pass, as you can see from the previous screenshot.



**Step 10. Prepare to Deploy the Contract to the Goerli Network**

This step will use the Goerli Test Network. If you need test Ether, receive it from

<https://goerlifaucet.com/>

In the directory “upgradable” create a “scripts” directory.

In the directory “scripts” create the file deployProxy.js

Paste the below code into the deployProxy.js file

|  |
| --- |
| // scripts/deployProxy.js const { ethers, upgrades } = require("hardhat");  async function main() {    const ERC20TokenV1 = await ethers.getContractFactory("ERC20TokenV1");    const erc20TokenV1 = await upgrades.deployProxy(ERC20TokenV1, [1000000]);    await erc20TokenV1.deployed();    const implementationAddress = await upgrades.erc1967.getImplementationAddress(      erc20TokenV1.address    );    console.log(      "ERC20TokenV1 contract deployed at address: " + erc20TokenV1.address    );    console.log(      "implementation contract deployed at address: " + implementationAddress    );  }  main().catch((error) => {    console.error(error);    process.exitCode = 1;  }); |

**Step 11. Deploy the Contract to goerli**

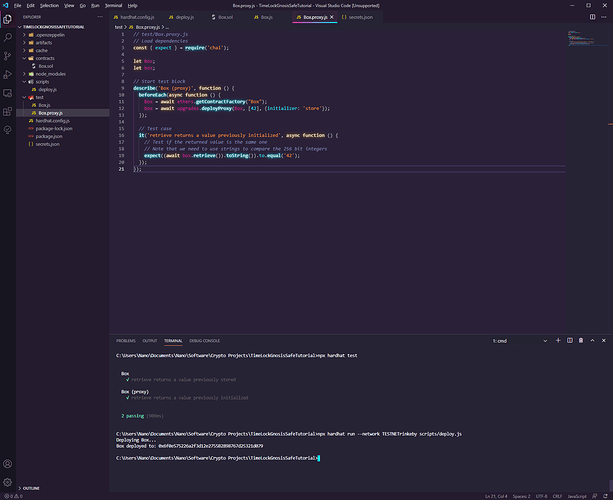
Run the following command to deploy the contract.

|  |
| --- |
| npx hardhat run --network goerli scripts/deployProxy.js |

A message should appear saying your Box contract was deployed to an Address

IMPORTANT - This address is the PROXY ADDRESS and you must note this down.

In this example the Proxy Address is **0x6eE935B6B58A428Fd550b9F38D63d9b80E013D35 as shown in the screenshot below.**



**Step 12. Verify the Contracts for your specific project.**

The proxy address shown is already verified thanks to Open Zeppelin.

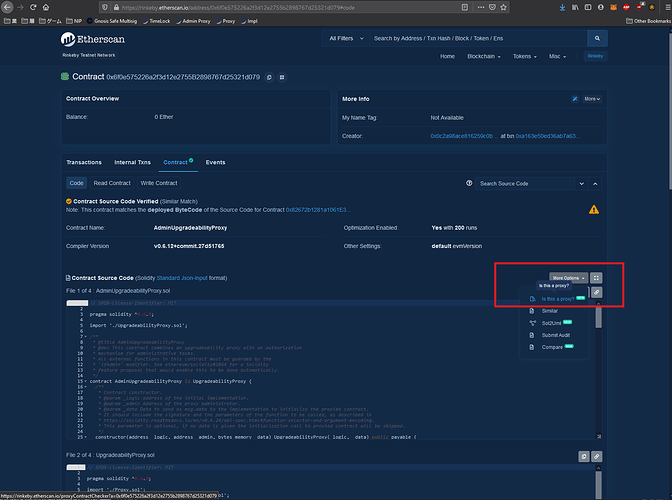
However, we need to still verify our IMPLEMENTATION Contract Code, which is Box.sol.

Navigate to Etherscan and go to your Contract.

The example Proxy Contract is:

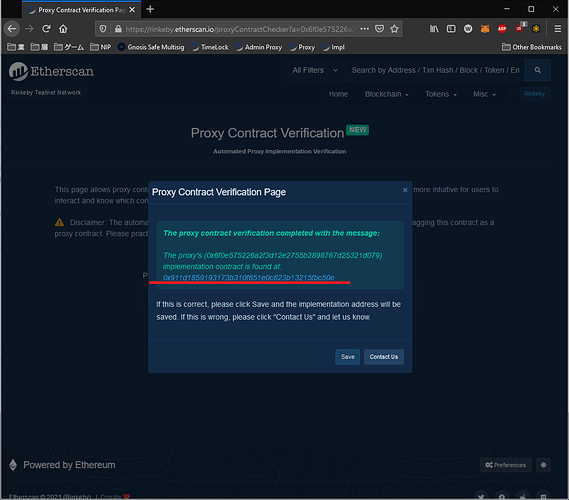
<https://goerli.etherscan.io/address/0x6eE935B6B58A428Fd550b9F38D63d9b80E013D35>

Click More Options, and click “Is this a Proxy?”



Click Verify

Now the IMPLEMENTATION ADDRESS is shown.



If you’re following the tutorial exactly, then your code will already be verified, but assuming it is not, you will need to verify it with the next steps.

Important - Copy the IMPLEMENTATION ADDRESS down.

In this example my implementation is

[https://goerli.etherscan.io/address/0x911d1859193173b310f651e0c823b13215fbc50e 16](https://rinkeby.etherscan.io/address/0x911d1859193173b310f651e0c823b13215fbc50e)

Back in your IDE or Compiler run the following verify command

|  |
| --- |
| npx hardhat verify --network goerli PROXYADDRESS |

For this tutorial’s Proxy Address the example command is

npx hardhat verify --network goerli

0x911d1859193173b310f651e0c823b13215fbc50e

The message will say "successfully verified contract Box on Etherscan."

You can then navigate to your Implementation Contract

[https://goerli.etherscan.io/address/0x911d1859193173b310f651e0c823b13215fbc50e#code 9](https://rinkeby.etherscan.io/address/0x911d1859193173b310f651e0c823b13215fbc50e#code)

And view its code.

**Step 13. Create the Gnosis Safe Multisig**

A multisig is a wallet that is shared between a group of other individuals.

To do any transaction, it requires multiple signatures from the individuals.

In this example, we will only have 1 individual, but in real use cases you will have your team on the gnosis safe.

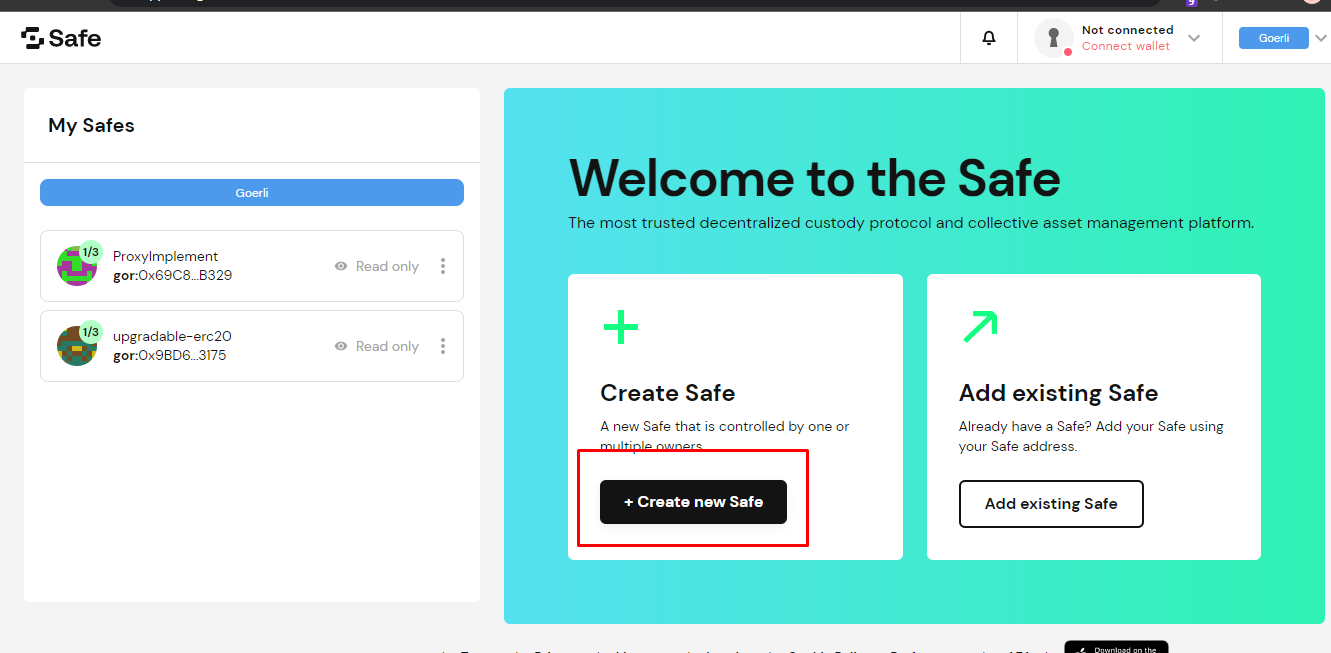
Be sure to have a MetaMask, or other compatible wallet, for use with Gnosis before progressing.

Set your network to goerli.

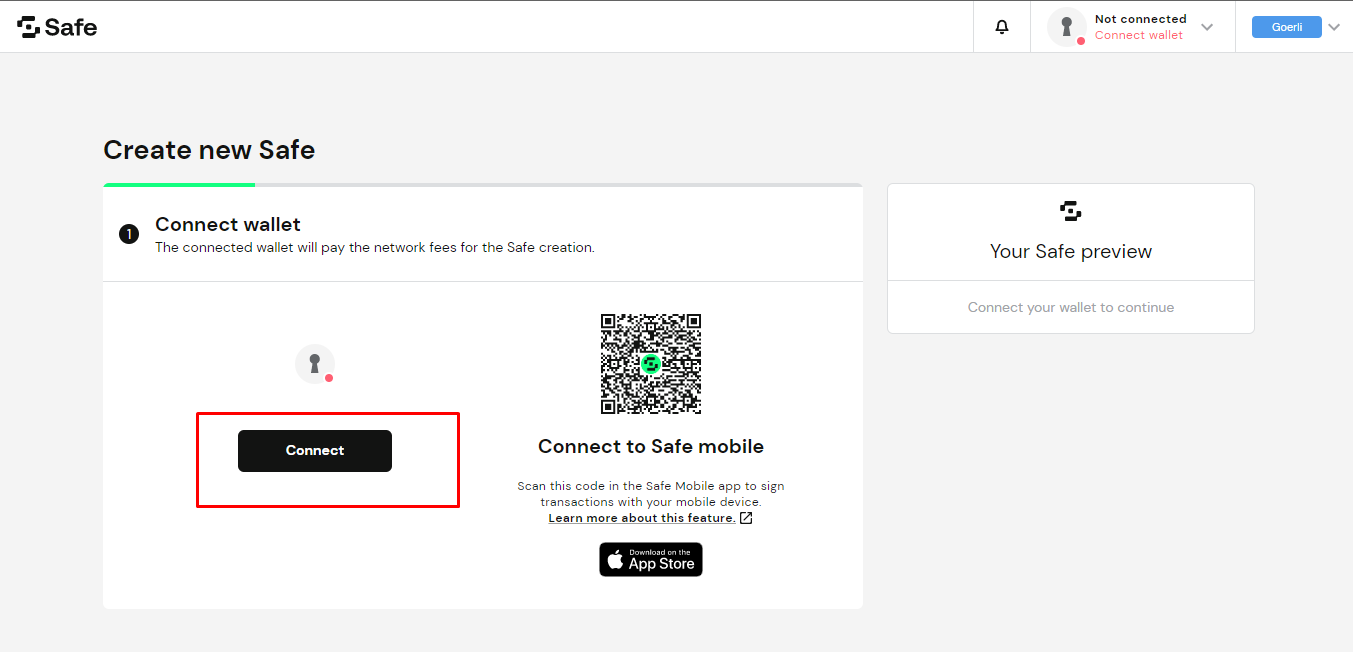
Go to the safe.global : <https://app.safe.global/welcome>

Follow the Steps:

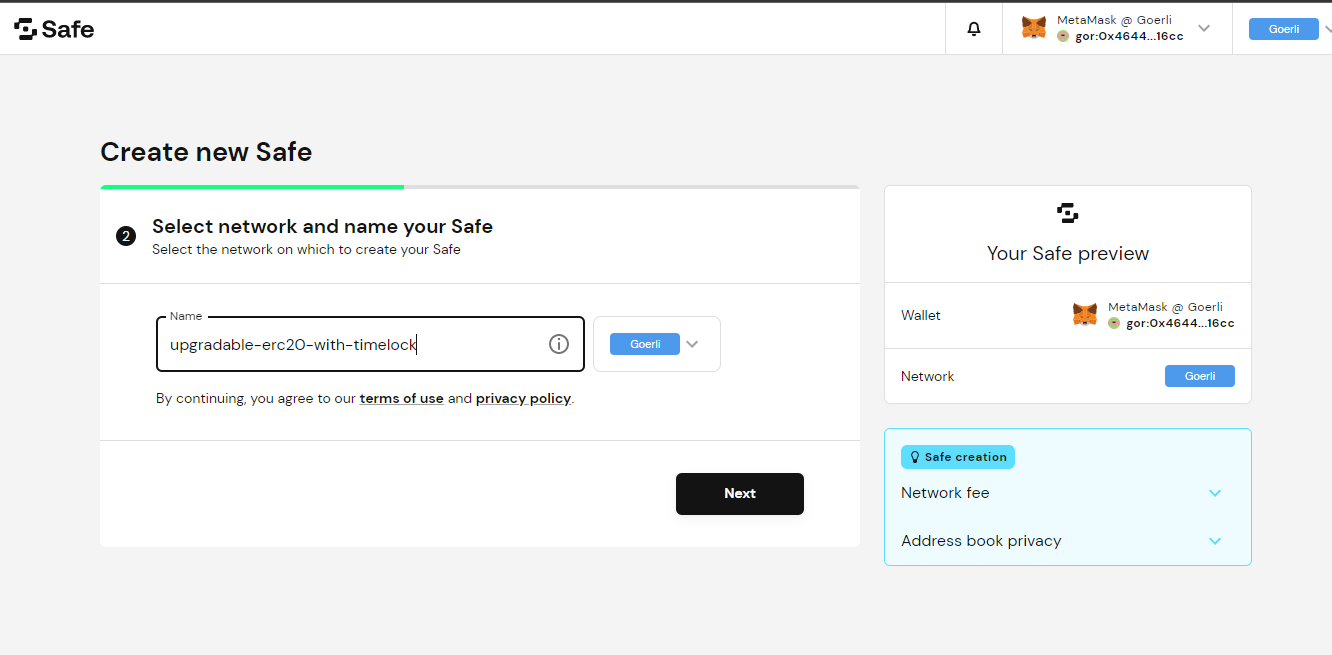
1. Create New Safe



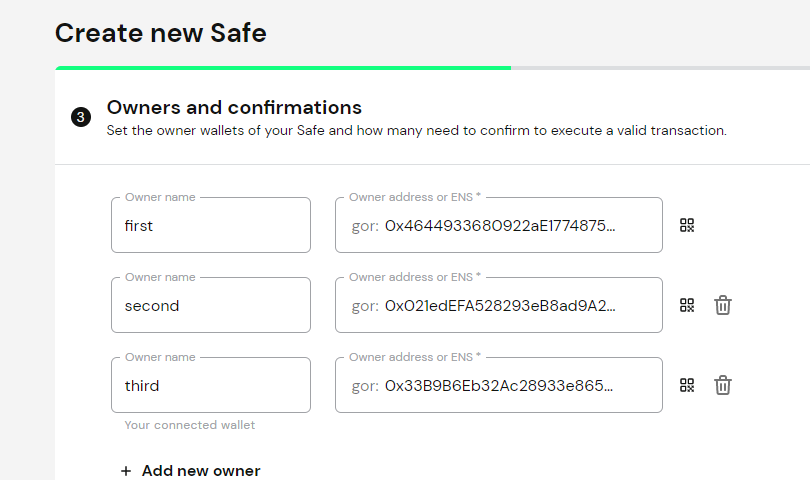
1. Connect Wallet



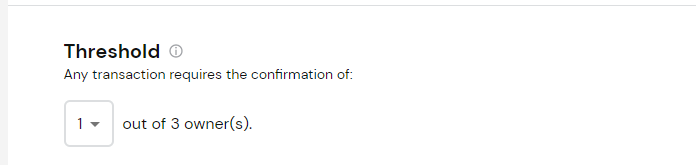
1. Name your Safe



1. Confirm Owners

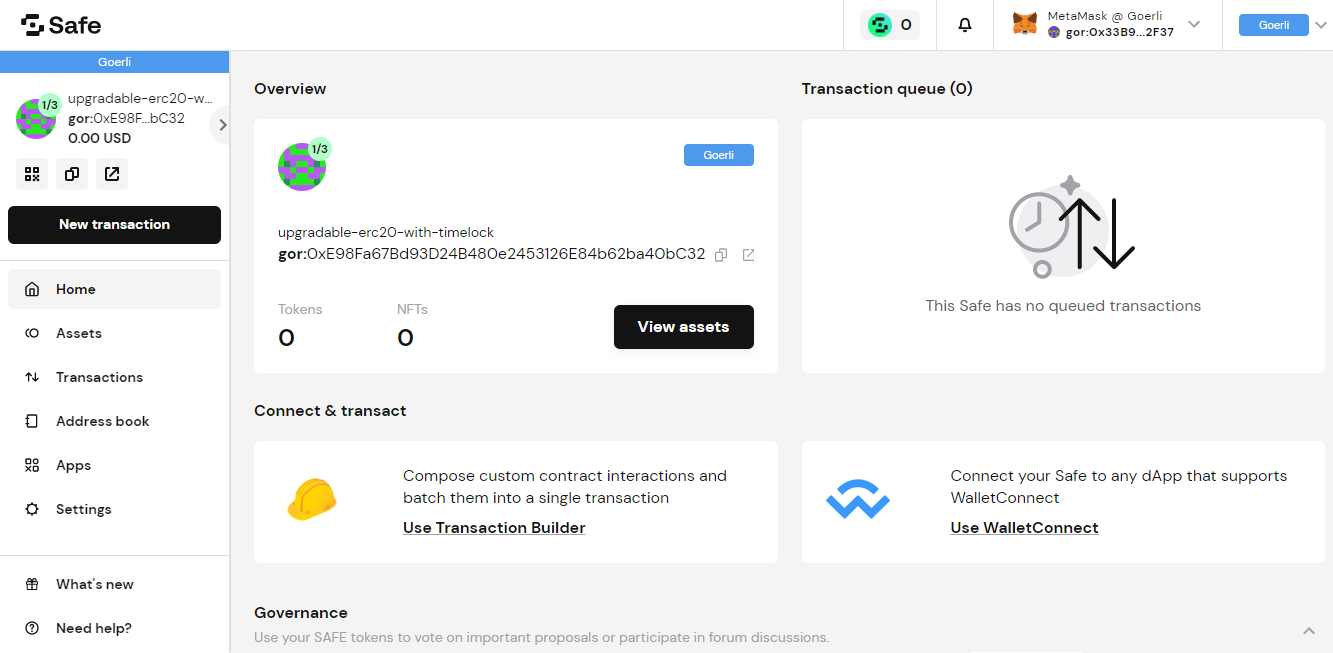


1. Set number of confirmations required.



1. Next & Next
2. Submit
3. Confirm
4. Get Started

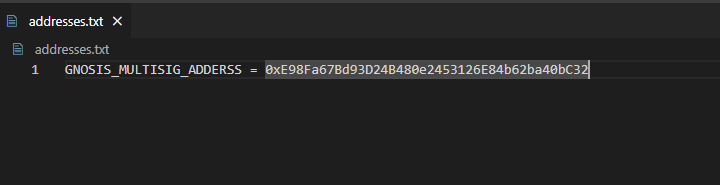
It should look similar to this screen after the setup of the safe is complete.



**Important - Copy down your Gnosis Safe Address**

**Create a new file address.txt and save this address**

In this example it is 0xE98Fa67Bd93D24B480e2453126E84b62ba40bC32



**Step 14. Create the TimeLock Contract**

Back in your IDE, use the command

|  |
| --- |
| npm install @openzeppelin/contracts |

This tutorial will use Open Zeppelin’s TimeLock Contract as a base.

which can be found in your node\_modules folder under

## [openzeppelin-contracts](https://github.com/OpenZeppelin/openzeppelin-contracts)/[contracts](https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts)/[governance](https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/governance)/TimelockController.sol

Copy and paste this file into the “contracts” folder.

Open the file and modify

pragma solidity ^0.8.0;

to

pragma solidity ^0.8.6;

import "../access/AccessControl.sol";

to

import "@openzeppelin/contracts/access/AccessControl.sol";

**Step 15. Create the TimeLock Deploy script**

In the “scripts” folder, create deploy-timelock.js

and paste the below code into the file.

|  |
| --- |
| // deployer for timelock  const argumentsArray = require('../arguments.js');  async function main() {    var delayTime = argumentsArray[0];  var proposersArray = argumentsArray[1];  var executorsArray = argumentsArray[2];  var admin=argumentsArray[3]  const [deployer] = await ethers.getSigners();    console.log(  "Deploying contracts with the account:",  deployer.address  );    console.log("Account balance:", (await deployer.getBalance()).toString());    const Token = await ethers.getContractFactory("TimelockController");   const token = await Token.deploy(delayTime, proposersArray, executorsArray,admin);   console.log("Timelock address:", token.address);   }    main()  .then(() => process.exit(0))  .catch(error => {  console.error(error);  process.exit(1);  }); |

Step 16. Create the **arguments** file for our TimeLock deployer.

Now create a file arguments.js in the upgradable folder.

In that paste the following code

|  |
| --- |
| module.exports = [  150, // minimum delay timer in seconds  ['0x2106fD5d58B76Cd56480eCE1dC5f864a4914a021'],// Proposers Array  ['0x2106fD5d58B76Cd56480eCE1dC5f864a4914a021'],// Executors Array "<enter your metamask goerli testnet account address here"> ]; |

You will need to modify “0x2106fD5d58B76Cd56480eCE1dC5f864a4914a021” to be your Gnosis Safe Address.

These are the arguments used when deploying the TimeLock.

For this example I have set the minimum delay timer to 150, which is 2.5 minutes

The Proposers and the Executors for our TimeLock are both set to the Gnosis Safe’s address.

These parameters are in Array form, meaning you can set multiple Proposers and Executors,

but for this example, because we are using a MultiSig, we will use only the MultiSig to pass our transactions through.

**Step 17. Understand what the TimeLock does with the Gnosis Multisig Safe**

Open the TimelockController.sol file

It has roles, which are addresses that can do certain actions.

The proposers are allowed to schedule transactions.

The executors are allowed to execute transactions.

These are both set to the Gnosis Safe Address in the arguments.js file.

The two primary functions we need to understand are the schedule and the execute functions.

These two functions will be called by our Gnosis Multisig.

In order to have our TimeLock Execute Transactions, they first must be Scheduled with the schedule function.

After the delay time elapses, then you can call the execution function toExecute the transaction.

Both the schedule and the execute function have the same arguments, except that execute does not have the delay argument.

Below is a short explanation of each argument in the file-

**target** - Address of the contract you want to execute the transaction on.

**value** - a value that can be assigned with this, normally it will be 0.

**data** - this is the data of a transaction

**predecessor** - if this transaction relies on one before it, you will put it’s Operation ID here. Each Operation performed by the TimeLock has an ID attached to it. This can be used in conjunction with the predecessor argument. If you do not have a predecessor use 0x0000000000000000000000000000000000000000000000000000000000000000

**salt** - this is a password that you can use when scheduling transactions, and must be input during the execution function to actually execute the function. This could be useful if you were sharing a TimeLock with many different people or had many users, and wanted to safeguard your transactions from being executed by other individuals. In this example there is TimeLock for one team of users, so the salt is set to 0x0000000000000000000000000000000000000000000000000000000000000000.

**delay** - this is the delay you would like. It must be equal to or greater than the minimum delay. In this example it was set to 150 seconds. You have the option to make the delay longer.

Whenever you schedule a transaction, it generates an ID associated with the operation. Later on in the example, you will see how to obtain this ID from the schedule transaction and how to get information about it.

There is also a cancel function, which you can use to cancel an operation with it’s ID.

If you would like to change your minimum delay, you can use the updateDelay function.

Note that in order to use this, you must call it with the TimeLock itself.

**Step 18. Optional Testing**

You may wish to develop some tests to experiment with the TimeLock function before deploying.

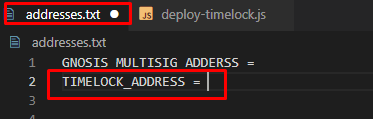
As this code is not modified and is from Open Zeppelin, it will not be tested because it is expected to work as is.

**Step 19. Deploy and Verify the TimeLock.**

Run

|  |
| --- |
| npx hardhat run --network goerli scripts/deploy-timelock.js |

Important - Note down the TimeLock’s address into address.txt



In the example it is 0xa0ea1e6Bea3BE3f9565E557AB6Aefe22D1e0e74e

Now verify using the following command

npx hardhat verify --network goerli --constructor-args arguments.js 0xa0ea1e6Bea3BE3f9565E557AB6Aefe22D1e0e74e

Notice that we have to include the arguments.js file

Otherwise the verification will not know what arguments we used when deploying.

Navigate to the Address in Etherscan to verify that it was verified.

**Step 20. Transfer Ownership of the Proxy Admin (Administrator of the Proxy, not the actual Proxy) to the TimeLock.**

In order to upgrade the contract, ERC20TokenV1.sol to ERC20TokenV2.sol, we use the Proxy Administrator to upgrade.

Upon creation, the owner of the Proxy Administrator is the deployer.

To find the Proxy Admin address, you can look in goerli.json in the .oppenzeppelin folder. At the bottom of the file is

|  |
| --- |
| "admin": {  "address": "0x697492C144FfE1Fd32dcc9CC47005A968a94694B",  "txHash": "0xbfccfa1d79a55fbdb8eb28fb7cc2143760f6263020ca121f3d00716f1a46534d" } |

Important - Note the Proxy Admin Address down.

Inside the scripts folder, create the file “**transferOwnership**.js” and copy the below code into it.

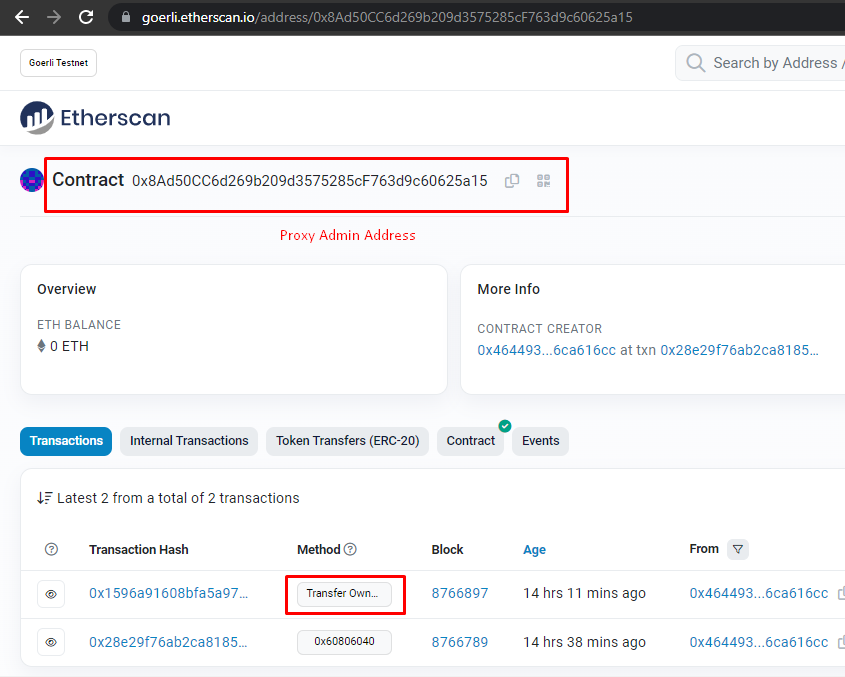
|  |
| --- |
| async function main() {    const newOwnerOfTheProxyAdmin = "0xe0Ea7A063acB21112528ce5d1b85D8261E5849bb";    // this will be the address of the TimeLock, as we need it to be the owner of the Proxy Admin.    console.log("Transferring ownership of ProxyAdmin...");    // The owner of the ProxyAdmin can upgrade our contracts    await upgrades.admin.transferProxyAdminOwnership(newOwnerOfTheProxyAdmin);    console.log(      "Transferred ownership of ProxyAdmin to:",      newOwnerOfTheProxyAdmin    );  }  main()    .then(() => process.exit(0))    .catch((error) => {      console.error(error);      process.exit(1);    }); |

You will need to modify 0xa0ea1e6Bea3BE3f9565E557AB6Aefe22D1e0e74e to be your TimeLock address

Now run

|  |
| --- |
| npx hardhat run --network goerli scripts/transfer\_ownership.js |

to transfer ownership to the TimeLock.



**Step 21. Prepare ERC20TokenV2.sol for the Upgrade**

Create ERC20TokenV2.sol in the contracts folder and paste the following code.

In ERC20TokenV2, have added new functionalities like **Mint(), Burn(), Buy(), allowanceToken().**

|  |
| --- |
| // contracts/ERC20TokenV2.sol // SPDX-License-Identifier: MIT  pragma solidity ^0.8.6;  import "@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol";  import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";  contract ERC20TokenV2 is Initializable, ERC20Upgradeable {      address public owner;      function initialize(uint \_totalSupply) public initializer {          \_\_ERC20\_init("ERC20 Token", "TKN");          \_mint(msg.sender, \_totalSupply);          owner = msg.sender;      }      modifier onlyOwner() {          require(msg.sender == owner, "Only owner can perform this operation");          \_;      }      /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PHASE 2 : Added Mint(), Burn(), Buy() and allowanceToken() \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/      function mint(address to, uint256 amount) public onlyOwner {          \_mint(to, amount);      }      function burn(address to, uint256 amount) public onlyOwner {          \_burn(to, amount);      }      receive() external payable {}      function buy() external payable returns(bool) {          require(msg.sender.balance >= msg.value && msg.value >= 0, "Insufficient Balance in the Contract");          (bool success, ) = address(this).call{value: msg.value }("");          require(success,"Unsucessful buy attempt");          uint \_amount = msg.value \* 1000;          \_transfer(owner, msg.sender, \_amount);          return true;      }      function allowanceToken(address \_account, uint256 \_amount) public returns (bool){          require (\_account != address(this) && \_amount != 0, "ERC20: ");          \_spendAllowance(msg.sender, \_account, \_amount);          return true;      }  } |

**Step 22. Create ERC20TokenV2Tests for Contract and Proxy**

Create ERC20TokenV2.js in the test folder and copy the below code into it.

|  |
| --- |
| // test/ ERC20TokenV2.js const { expect } = require("chai");  const { ethers, upgrades } = require("hardhat");  let tokenV2;  let deployer, user1;  const tokenName = "ERC20 Token";  const tokenSymbol = "TKN";  const amountToEther = (amount) => {    return ethers.utils.parseEther(amount.toString());  };  const amount = amountToEther(1000000);  describe("ERC20TokenV2 Test", () => {    before(async () => {      const ERC20TokenV2 = await ethers.getContractFactory("ERC20TokenV2");      tokenV2 = await upgrades.deployProxy(ERC20TokenV2, [amount]);      await tokenV2.deployed();      const account = await ethers.getSigners();      deployer = account[0];      user1 = account[1];    });    describe("Phase 1: ERC20 contract with a pre-minting of 1 million tokens", () => {      it("1. Token should have name", async () => {        const name = await tokenV2.name();        expect(name).to.be.equal(tokenName);      });      it("2. Token should have symbol", async () => {        const symbol = await tokenV2.symbol();        expect(symbol).to.be.equal(tokenSymbol);      });      it("3. Token should have 1 million tokens", async () => {        const totalSupply = await tokenV2.totalSupply();        expect(totalSupply.toString()).to.be.equal(amount.toString());      });      it("4. Deployer should have balance of 10000 tokens", async () => {        const balance = await tokenV2.balanceOf(deployer.address);        expect(balance.toString()).to.be.equal(amount.toString());      });      it("5. Deployer should be token owner", async () => {        const owner = await tokenV2.owner();        expect(owner).to.be.equal(deployer.address);      });    });    describe("Phase 2: Testing mint(), burn(), allowance() functionalities", () => {      it("8. Only owner should be able to mint the token", async () => {        const amount2 = amountToEther(5000);        const beforeTotalSupply = await tokenV2.totalSupply();        const beforeBalance = await tokenV2.balanceOf(deployer.address);        const tx = await tokenV2          .connect(deployer)          .mint(deployer.address, amount2);        await tx.wait();        const afterTotalSupply = await tokenV2.totalSupply();        expect(afterTotalSupply).to.be.greaterThan(beforeTotalSupply);        const afterBalance = await tokenV2.balanceOf(deployer.address);        expect(afterBalance).to.be.greaterThan(beforeBalance);      });      it("9. User1 should not be able to mint the token", async () => {        const amount2 = 5000;        await expect(tokenV2.connect(user1).mint(user1.address, amount2)).to.be          .reverted;      });      it("10. Only owner should be able to burn the token", async () => {        const amount2 = amountToEther(2000);        const beforeTotalSupply = await tokenV2.totalSupply();        const beforeBalance = await tokenV2.balanceOf(deployer.address);        const tx = await tokenV2          .connect(deployer)          .burn(deployer.address, amount2);        await tx.wait();        const afterTotalSupply = await tokenV2.totalSupply();        expect(afterTotalSupply).to.be.lessThan(beforeTotalSupply);        const afterBalance = await tokenV2.balanceOf(deployer.address);        expect(afterBalance).to.be.lessThan(beforeBalance);      });      it("11. User1 should not be able to burn the token", async () => {        const amount2 = amountToEther(2000);        await expect(tokenV2.connect(user1).burn(user1.address, amount2)).to.be          .reverted;      });      it("12. User1 should be able to buy the tokens", async () => {        const beforeBalance = await tokenV2.balanceOf(user1.address);        console.log(beforeBalance);        const valueOption = { value: amountToEther(2) };        await tokenV2.connect(user1).buy(valueOption);        const afterBalance = await tokenV2.balanceOf(deployer.address);        expect(afterBalance).to.be.greaterThan(beforeBalance);      });    });  }); |
|  |

Create ERC20TokenV2.proxy.js in the test folder and copy the below code into it.

|  |
| --- |
| // test/ ERC20TokenV2.proxy.js const { expect } = require("chai");  const { ethers, upgrades } = require("hardhat");  let tokenV2;  let deployer, user1;  const tokenName = "ERC20 Token";  const tokenSymbol = "TKN";  const amountToEther = (amount) => {    return ethers.utils.parseEther(amount.toString());  };  const amount = amountToEther(1000000);  describe("ERC20TokenV2 Proxy Test", () => {    before(async () => {      const TokenV1 = await ethers.getContractFactory("ERC20TokenV1");      const TokenV2 = await ethers.getContractFactory("ERC20TokenV2");      const tokenV1 = await upgrades.deployProxy(TokenV1, [amount]);      tokenV2 = await upgrades.upgradeProxy(tokenV1.address, TokenV2);      const account = await ethers.getSigners();      deployer = account[0];      user1 = account[1];    });    describe("Phase 1: ERC20 contract with a pre-minting of 1 million tokens", () => {      it("1. Token should have name", async () => {        const name = await tokenV2.name();        expect(name).to.be.equal(tokenName);      });      it("2. Token should have symbol", async () => {        const symbol = await tokenV2.symbol();        expect(symbol).to.be.equal(tokenSymbol);      });      it("3. Token should have 1 million tokens", async () => {        const totalSupply = await tokenV2.totalSupply();        expect(totalSupply.toString()).to.be.equal(amount.toString());      });      it("4. Deployer should have balance of 10000 tokens", async () => {        const balance = await tokenV2.balanceOf(deployer.address);        expect(balance.toString()).to.be.equal(amount.toString());      });      it("5. Deployer should be token owner", async () => {        const owner = await tokenV2.owner();        expect(owner).to.be.equal(deployer.address);      });    });    describe("Phase 2: Testing mint(), burn(), allowance() functionalities", () => {      it("8. Only owner should be able to mint the token", async () => {        const amount2 = amountToEther(5000);        const beforeTotalSupply = await tokenV2.totalSupply();        const beforeBalance = await tokenV2.balanceOf(deployer.address);        const tx = await tokenV2          .connect(deployer)          .mint(deployer.address, amount2);        await tx.wait();        const afterTotalSupply = await tokenV2.totalSupply();        expect(afterTotalSupply).to.be.greaterThan(beforeTotalSupply);        const afterBalance = await tokenV2.balanceOf(deployer.address);        expect(afterBalance).to.be.greaterThan(beforeBalance);      });      it("9. User1 should not be able to mint the token", async () => {        const amount2 = 5000;        await expect(tokenV2.connect(user1).mint(user1.address, amount2)).to.be          .reverted;      });      it("10. Only owner should be able to burn the token", async () => {        const amount2 = amountToEther(2000);        const beforeTotalSupply = await tokenV2.totalSupply();        const beforeBalance = await tokenV2.balanceOf(deployer.address);        const tx = await tokenV2          .connect(deployer)          .burn(deployer.address, amount2);        await tx.wait();        const afterTotalSupply = await tokenV2.totalSupply();        expect(afterTotalSupply).to.be.lessThan(beforeTotalSupply);        const afterBalance = await tokenV2.balanceOf(deployer.address);        expect(afterBalance).to.be.lessThan(beforeBalance);      });      it("11. User1 should not be able to burn the token", async () => {        const amount2 = amountToEther(2000);        await expect(tokenV2.connect(user1).burn(user1.address, amount2)).to.be          .reverted;      });      it("12. User1 should be able to buy the tokens", async () => {        const beforeBalance = await tokenV2.balanceOf(user1.address);        const valueOption = { value: amountToEther(2) };        await tokenV2.connect(user1).buy(valueOption);        const afterBalance = await tokenV2.balanceOf(deployer.address);        expect(afterBalance).to.be.greaterThan(beforeBalance);      });    });  }); |

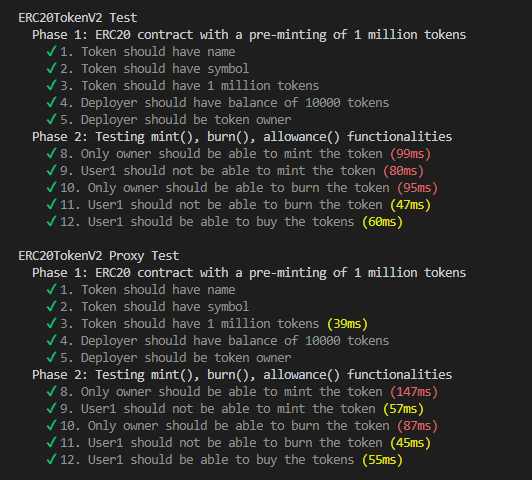
**Step 23. Run the tests and verify that all pass**

Run

|  |
| --- |
| npx hardhat test |

You should see a message that indicates all of them pass, including the upgrade of the ERC20TokenV1 to ERC20TokenV2.

Consult the screenshot below for how it should look.



**Step 24. Script to Prepare the new upgrade to V2**

Inside the scripts folder, create the file “prepareUpgradeForV2.js” and copy the below code into it.

|  |
| --- |
| // scripts/prepareUpgradeForV2.js const { ethers, upgrades } = require("hardhat");  async function main() {    const proxyAddress = "0x6eE935B6B58A428Fd550b9F38D63d9b80E013D35";    // This is the proxy address NOT the admin of the proxy.    const ERC20TokenV2 = await ethers.getContractFactory("ERC20TokenV2");    console.log("Preparing upgrade...");    const erc20TokenV2 = await upgrades.prepareUpgrade(      proxyAddress,      ERC20TokenV2    );    console.log("ERC20TokenV2 at:", erc20TokenV2);  }  main()    .then(() => process.exit(0))    .catch((error) => {      console.error(error);      process.exit(1);    }); |

You will need to change 0x6f0e575226a2f3d12e2755B2898767d25321d079 to your proxy address.

Note that it is the Proxy Address, not the Administrator of the Proxy (Proxy Admin).

Run

|  |
| --- |
| npx hardhat run --network goerli scripts/ prepareUpgradeForV2.js |

Copy down the new Implementation Address, in this example it will be 0xbe0b67cf6301E9B061b34FFae02D45ed671A7475

Now run below script to verify the new Implementation Contract.

|  |
| --- |
| npx hardhat verify --network goerli 0xbe0b67cf6301E9B061b34FFae02D45ed671A7475 |

**Step 25. Optional… Another way to Get the Proxy Admin Address**

As stated in an earlier step…

The easiest way to find the Proxy Admin address is to look in look in goerli.json in the .oppenzeppelin folder. At the bottom of the file is

|  |
| --- |
| "admin": {  "address": "0x697492C144FfE1Fd32dcc9CC47005A968a94694B",  "txHash": "0xbfccfa1d79a55fbdb8eb28fb7cc2143760f6263020ca121f3d00716f1a46534d" } |

Note the Address down.

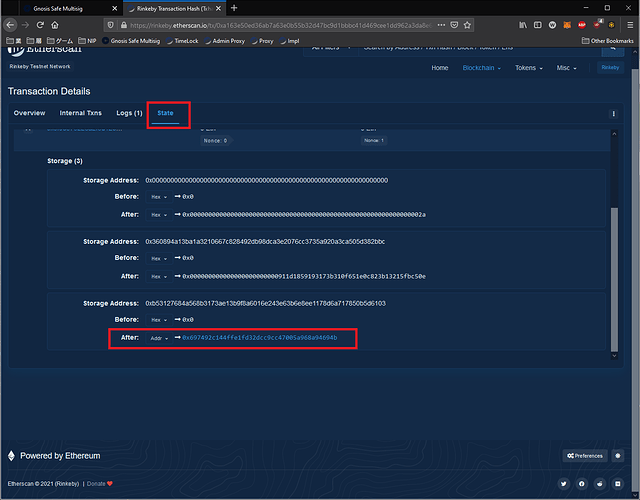
You can also find the Proxy Admin address by going to the Proxy Address on goerli etherscan, and finding the Contract Creation Transaction.

After going to that transaction, click the State tab.

In the 3rd State, click the drop-down arrow to see 3 Storage addresses.

The last storage address is the Proxy Administrator. You will need to change this number from hex to Address.

Consult the screenshot below for guidance.



At one point in time, you could find the Administrator of the Proxy’s address via the Proxy’s read functions. But in the newer version this is not available.

**Step 26. Get the data for the Upgrade Transaction**

Go to the Proxy Admin’s page in Etherscan.

Click Write Contract

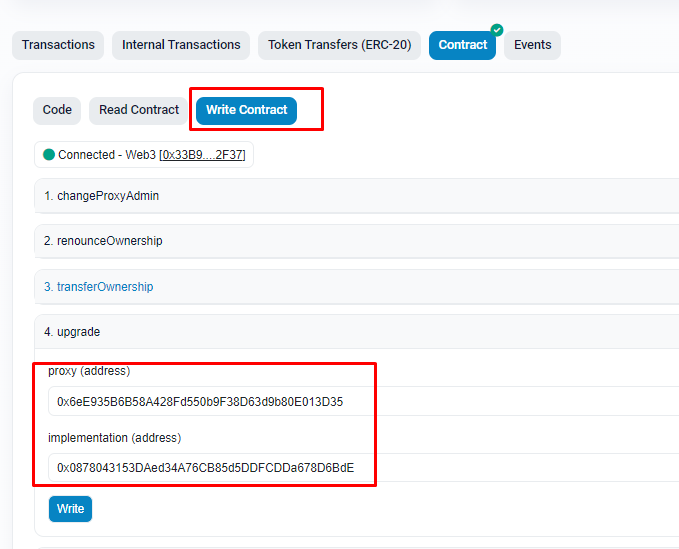
Connect to Web3

Select function 4, upgrade

proxy (address) = The Proxy Address, for this example it is 0x6f0e575226a2f3d12e2755B2898767d25321d079

implementation (address) = The New Implementation Address, for this example it is 0xbe0b67cf6301E9B061b34FFae02D45ed671A7475

Refer to the screenshot below for guidance.

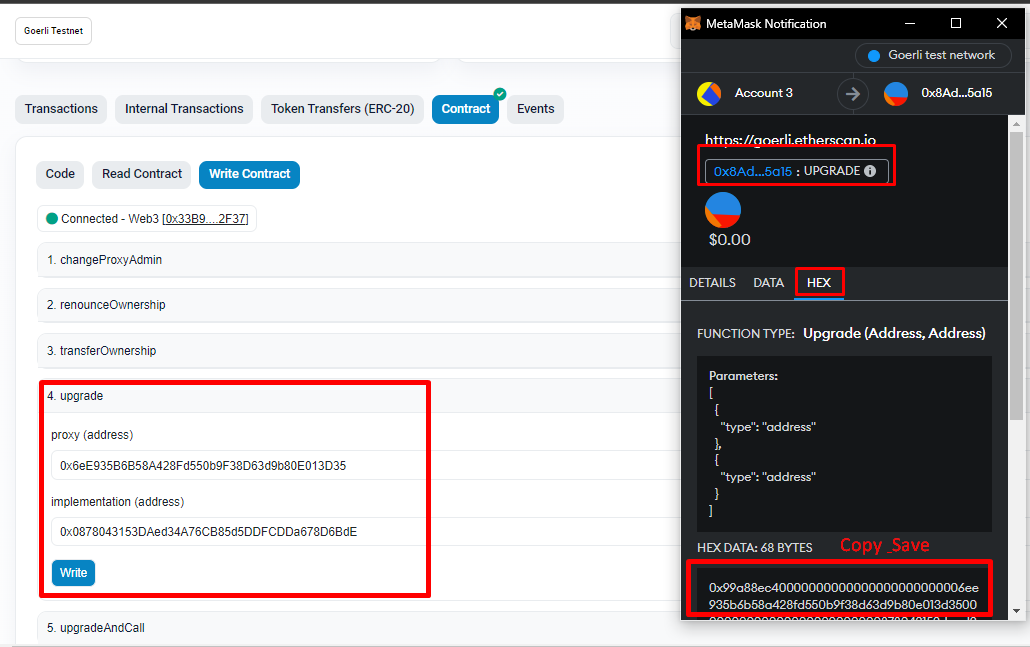


***Click write, but DO NOT confirm.***

In the MetaMask tab, click Data.

Find the Hex Data field, and copy down the Hex Data.

Refer to the screenshot below for how it should look.



For this example it is…

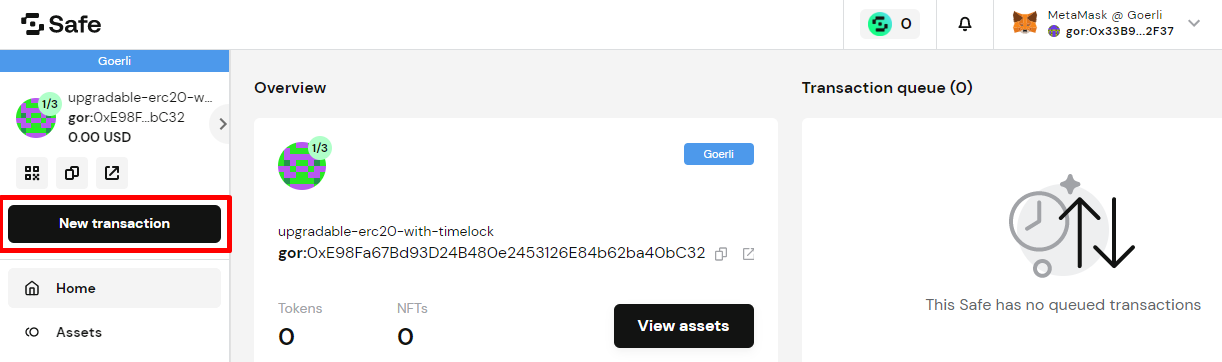
0x99a88ec40000000000000000000000006ee935b6b58a428fd550b9f38d63d9b80e013d350000000000000000000000000878043153daed34a76cb85d5ddfcdda678d6bde

Cancel the transaction

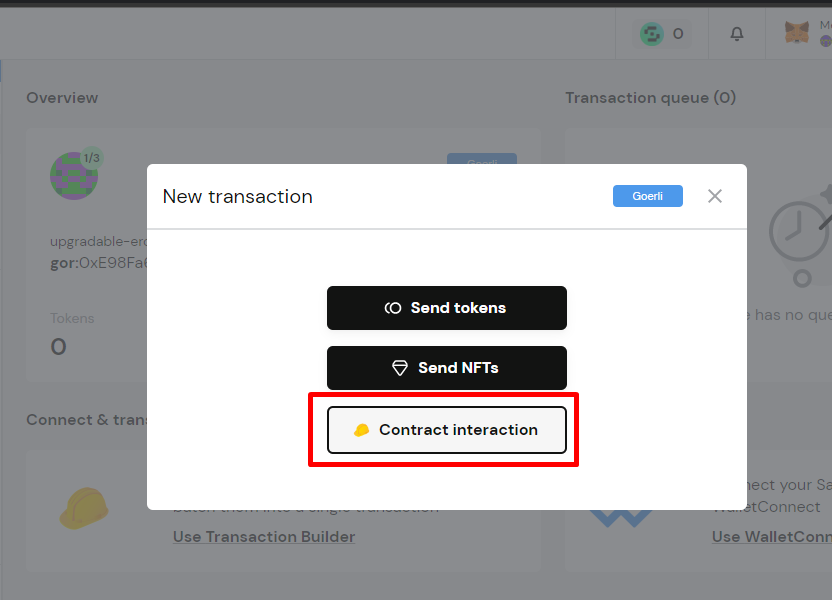
**Step 27. Schedule the Transaction to Upgrade the Contract with the Gnosis Safe using the TimeLock.**

Go to your safe in the Gnosis app

1)Click New Transaction



2)Contract Interaction



3)In the “Contract Address” field, paste your TimeLock address

For this example it is 0xe0Ea7A063acB21112528ce5d1b85D8261E5849bb

4) Copy the timelock abi from hardhat artifacts and paste it here.

From the Method dropdown box below, select “schedule”.

Fill out the fields as shown below

target (address) = Your Proxy Admin Address (0x697492C144FfE1Fd32dcc9CC47005A968a94694B)

value (uint256) = 0

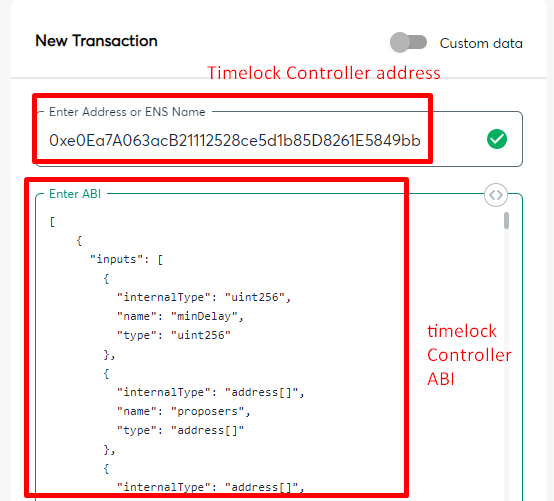
data (bytes) = The data for the Upgrade Transaction (0x99a88ec40000000000000000000000006f0e575226a2f3d12e2755b2898767d25321d079000000000000000000000000be0b67cf6301e9b061b34ffae02d45ed671a7475)

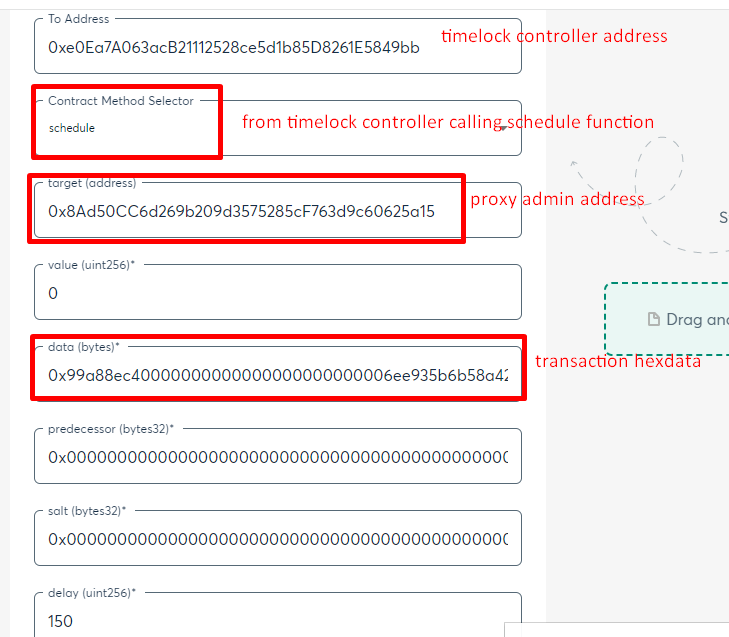
predecessor (bytes32) = 0x0000000000000000000000000000000000000000000000000000000000000000

salt (bytes32) = 0x0000000000000000000000000000000000000000000000000000000000000000

delay (uint256) = 150

Refer to the Screenshot below for how this should look.





For explanations on each field, please refer back to the TimeLock section as these are TimeLock functions.

Click Review

If you have done something incorrectly, it will warn you, back up and fix the error by reviewing the steps. It is likely the data.

Click Submit

Click Confirm in MetaMask.

**Step 28. Verify that the transaction was scheduled in the TimeLock**

While waiting 2.5 Minutes for the transaction to go through the TimeLock, go to the TimeLock’s address in etherscan.

Click the Events tab.

The most recent transaction is the “CallScheduled” one.

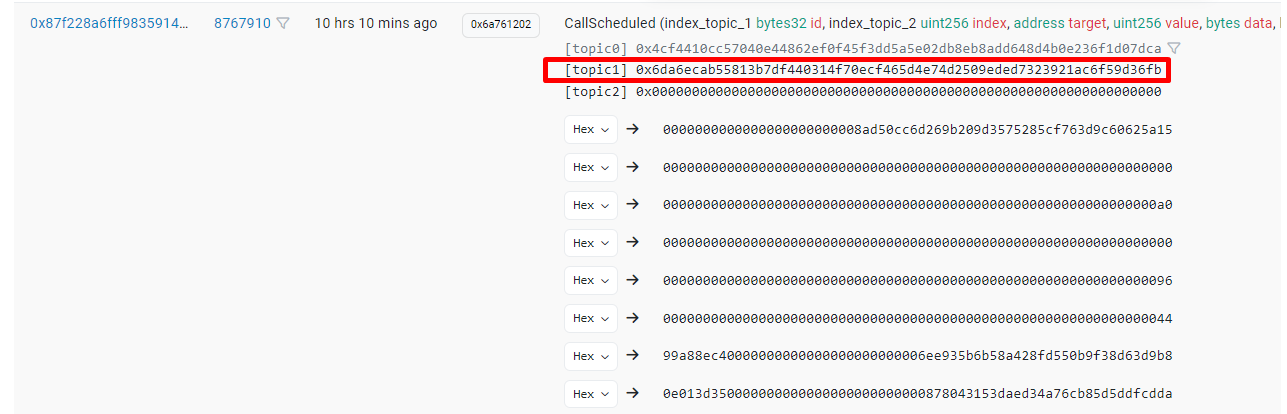
In that transaction log you can investigate the data passed into the CallScheduled function.

In [topic1] is the ID of the operation.

You can use this Operation ID in order to cancel a scheduled transaction.

You can use this Operation ID in the Predecessor field of Schedule/Execute functions.

See the screenshot below displaying this transaction.



**Step 29. Execute the Transaction to Upgrade the Contract with the Gnosis Safe using the TimeLock.**

Make sure your delay timer has passed, in this example it was 2.5 minutes.

Go to your safe in the Gnosis app

1)Click New Transaction

2)Contract Interaction

3)In the “Contract Address” field, paste your TimeLock address

4)For this example it is 0xa0ea1e6Bea3BE3f9565E557AB6Aefe22D1e0e74e

5)Copy the timelock abi from hardhat artifacts and paste it here.

6)From the Method dropdown box below, select “execute”

7)Fill out the fields as shown below

Value (ether) = 0

target (address) = Proxy Admin Address

(0x697492C144FfE1Fd32dcc9CC47005A968a94694B)

value (uint256) = 0

data (bytes) = The data for the Upgrade Transaction

(0x99a88ec40000000000000000000000006f0e575226a2f3d12e2755b2898767d25321d079000000000000000000000000be0b67cf6301e9b061b34ffae02d45ed671a7475)

predecessor (bytes32) = 0x0000000000000000000000000000000000000000000000000000000000000000

salt (bytes32) = 0x0000000000000000000000000000000000000000000000000000000000000000

Click Review

If you have done something incorrectly, it will warn you, back up and fix the error by reviewing the steps. It is likely the data.

Click Submit

Click Confirm in MetaMask.

**Step 30. Verify that the New Implementation contract is in place.**

Navigate to your Proxy Address in Etherscan. (0x6f0e575226a2f3d12e2755B2898767d25321d079 for this example).

The Contract Address 0x6f0e575226a2f3d12e2755B2898767d25321d079 page

allows users to view the source code, transactions, balances, and analytics for the contract address. Users can also interact and make transactions to the contract directly on…

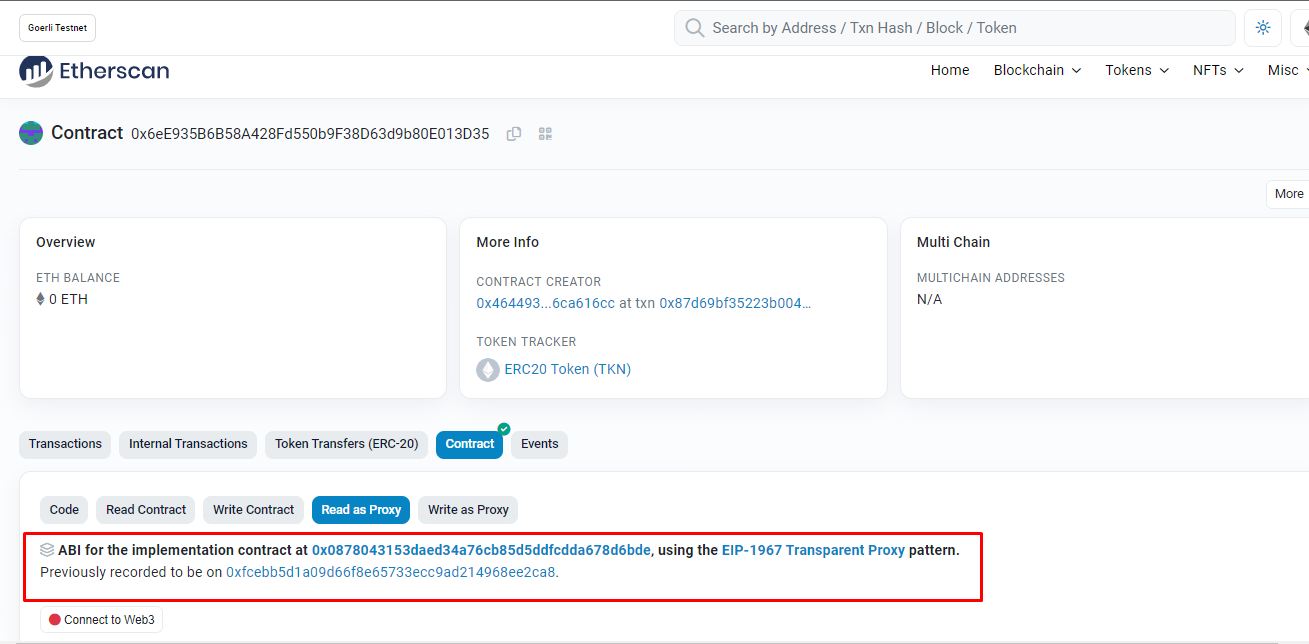
Click Read as Proxy.

A message should say " ABI for the implementation contract at 0xbe0b67cf6301e9b061b34ffae02d45ed671a7475, using the EIP-1967 Transparent Proxy pattern.

Previously recorded to be on 0x911d1859193173b310f651e0c823b13215fbc50e."

Verify that the Implementation Address is the one that was created when you ran the prepareupgrade.js script.

Refer to the below screenshot for guidance.



**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* STARTING V3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*]**

**Step 21. Prepare ERC20TokenV3.sol for the Upgrade**

Create ERC20TokenV2.sol in the contracts folder and paste the following code.

In ERC20TokenV2, have added new functionalities like **withdraw() with** **nonReentrant and getContractBalance().**

|  |
| --- |
| // contracts/ERC20TokenV3.sol // SPDX-License-Identifier: MIT  pragma solidity ^0.8.6;  import "@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol";  import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";  contract ERC20TokenV3 is Initializable, ERC20Upgradeable {      address public owner;      function initialize(uint \_totalSupply) public initializer {          \_\_ERC20\_init("ERC20 Token", "TKN");          \_mint(msg.sender, \_totalSupply);          owner = msg.sender;      }      modifier onlyOwner() {          require(msg.sender == owner, "Only owner can perform this operation");          \_;      }       modifier nonReentrant() {          bool locked;          require(locked == false, "Cannot Reenter Method");          locked = true;          \_;      }      /\*\*\*\*\*\*\* PHASE 2: Added Mint(), Burn(), Buy() \*\*\*\*\*\*\*\*\*\*\*\*/      function mint(address to, uint256 amount) public onlyOwner {          \_mint(to, amount);      }      function burn(address to, uint256 amount) public onlyOwner {          \_burn(to, amount);      }      receive() external payable {}      function buy() external payable returns(bool) {          require(msg.sender.balance >= msg.value && msg.value >= 0, "Insufficient Balance in the Contract");          (bool success, ) = address(this).call{value: msg.value }("");          require(success,"Unsucessful buy attempt");          uint \_amount = msg.value \* 1000;          \_transfer(owner, msg.sender, \_amount);          return true;      }      function allowanceToken(address \_account, uint256 \_amount) public returns (bool){          require (\_account != address(this) && \_amount != 0, "ERC20: ");          \_spendAllowance(msg.sender, \_account, \_amount);          return true;      }      /\*\*\*\*\*\*\* PHASE 3: Added Withdraw() and nonReentrant \*\*\*\*\*\*\*\*\*\*/      function withdraw(uint \_amount) external payable  onlyOwner nonReentrant returns(bool) {          require (\_amount <= address(this).balance, "Insufficient Balance");          (bool success, ) = owner.call{value: \_amount }("");          require(success,"Withdrawl of funds unsucessful");          return(true);      }      function getContractBalance() public view returns(uint256) {          return address(this).balance;      }  } |

**Step 22. Create ERC20TokenV3Tests for Contract and Proxy**

Create ERC20TokenV3.js in the test folder and copy the below code into it.

|  |
| --- |
| const { expect } = require("chai");  const { ethers, upgrades } = require("hardhat");  let tokenV3;  let deployer, user1;  const tokenName = "ERC20 Token";  const tokenSymbol = "TKN";  const amountToEther = (amount) => {    return ethers.utils.parseEther(amount.toString());  };  const amount = amountToEther(1000000);  describe("ERC20tokenV3 Test", () => {    before(async () => {      const ERC20TokenV3 = await ethers.getContractFactory("ERC20TokenV3");      tokenV3 = await upgrades.deployProxy(ERC20TokenV3, [amount]);      const account = await ethers.getSigners();      deployer = account[0];      user1 = account[1];    });    describe("Phase 1: ERC20 contract with a pre-minting of 1 million tokens", () => {      it("1. Token should have name", async () => {        const name = await tokenV3.name();        expect(name).to.be.equal(tokenName);      });      it("2. Token should have symbol", async () => {        const symbol = await tokenV3.symbol();        expect(symbol).to.be.equal(tokenSymbol);      });      it("3. Token should have 1 million tokens", async () => {        const totalSupply = await tokenV3.totalSupply();        expect(totalSupply.toString()).to.be.equal(amount.toString());      });      it("4. Deployer should have balance of 10000 tokens", async () => {        const balance = await tokenV3.balanceOf(deployer.address);        expect(balance.toString()).to.be.equal(amount.toString());      });      it("5. Deployer should be token owner", async () => {        const owner = await tokenV3.owner();        expect(owner).to.be.equal(deployer.address);      });    });    describe("Phase 2: Testing mint(), burn(), allowance() functionalities", () => {      it("8. Only owner should be able to mint the token", async () => {        const amount2 = amountToEther(5000);        const beforeTotalSupply = await tokenV3.totalSupply();        const beforeBalance = await tokenV3.balanceOf(deployer.address);        const tx = await tokenV3          .connect(deployer)          .mint(deployer.address, amount2);        await tx.wait();        const afterTotalSupply = await tokenV3.totalSupply();        expect(afterTotalSupply).to.be.greaterThan(beforeTotalSupply);        const afterBalance = await tokenV3.balanceOf(deployer.address);        expect(afterBalance).to.be.greaterThan(beforeBalance);      });      it("9. User1 should not be able to mint the token", async () => {        const amount2 = 5000;        await expect(tokenV3.connect(user1).mint(user1.address, amount2)).to.be          .reverted;      });      it("10. Only owner should be able to burn the token", async () => {        const amount2 = amountToEther(2000);        const beforeTotalSupply = await tokenV3.totalSupply();        const beforeBalance = await tokenV3.balanceOf(deployer.address);        const tx = await tokenV3          .connect(deployer)          .burn(deployer.address, amount2);        await tx.wait();        const afterTotalSupply = await tokenV3.totalSupply();        expect(afterTotalSupply).to.be.lessThan(beforeTotalSupply);        const afterBalance = await tokenV3.balanceOf(deployer.address);        expect(afterBalance).to.be.lessThan(beforeBalance);      });      it("11. User1 should not be able to burn the token", async () => {        const amount2 = amountToEther(2000);        await expect(tokenV3.connect(user1).burn(user1.address, amount2)).to.be          .reverted;      });      it("12. User1 should be able to buy the tokens", async () => {        const beforeBalance = await tokenV3.balanceOf(user1.address);        const valueOption = { value: amountToEther(2) };        await tokenV3.connect(user1).buy(valueOption);        const afterBalance = await tokenV3.balanceOf(deployer.address);        expect(afterBalance).to.be.greaterThan(beforeBalance);      });      describe("Phase 3: Testing Withdraw Functionality", () => {        it("Only owner should be able to withdraw the amount", async () => {          const beforeBalance = await tokenV3.getContractBalance();          const amount = amountToEther(1);          const tx = await tokenV3.connect(deployer).withdraw(amount);          await tx.wait();          const afterBalance = await tokenV3.getContractBalance();          expect(afterBalance).to.be.lessThan(beforeBalance);        });      });    });  }); |
|  |

Create ERC20TokenV3.proxy.js in the test folder and copy the below code into it.

|  |
| --- |
| // test/ ERC20TokenV3.proxy.js const { expect } = require("chai");  const { ethers, upgrades } = require("hardhat");  let tokenV3;  let deployer, user1;  const tokenName = "ERC20 Token";  const tokenSymbol = "TKN";  const amountToEther = (amount) => {    return ethers.utils.parseEther(amount.toString());  };  const amount = amountToEther(1000000);  describe("ERC20tokenV3 Proxy Test", () => {    before(async () => {      const ERC20TokenV1 = await ethers.getContractFactory("ERC20TokenV1");      const ERC20TokenV2 = await ethers.getContractFactory("ERC20TokenV2");      const ERC20TokenV3 = await ethers.getContractFactory("ERC20TokenV3");      const tokenV1 = await upgrades.deployProxy(ERC20TokenV1, [amount]);      const tokenV2 = await upgrades.upgradeProxy(tokenV1.address, ERC20TokenV2);      tokenV3 = await upgrades.upgradeProxy(tokenV2.address, ERC20TokenV3);      const account = await ethers.getSigners();      deployer = account[0];      user1 = account[1];    });    describe("Phase 1: ERC20 contract with a pre-minting of 1 million tokens", () => {      it("1. Token should have name", async () => {        const name = await tokenV3.name();        expect(name).to.be.equal(tokenName);      });      it("2. Token should have symbol", async () => {        const symbol = await tokenV3.symbol();        expect(symbol).to.be.equal(tokenSymbol);      });      it("3. Token should have 1 million tokens", async () => {        const totalSupply = await tokenV3.totalSupply();        expect(totalSupply.toString()).to.be.equal(amount.toString());      });      it("4. Deployer should have balance of 10000 tokens", async () => {        const balance = await tokenV3.balanceOf(deployer.address);        expect(balance.toString()).to.be.equal(amount.toString());      });      it("5. Deployer should be token owner", async () => {        const owner = await tokenV3.owner();        expect(owner).to.be.equal(deployer.address);      });    });    describe("Phase 2: Testing mint(), burn(), allowance() functionalities", () => {      it("8. Only owner should be able to mint the token", async () => {        const amount2 = amountToEther(5000);        const beforeTotalSupply = await tokenV3.totalSupply();        const beforeBalance = await tokenV3.balanceOf(deployer.address);        const tx = await tokenV3          .connect(deployer)          .mint(deployer.address, amount2);        await tx.wait();        const afterTotalSupply = await tokenV3.totalSupply();        expect(afterTotalSupply).to.be.greaterThan(beforeTotalSupply);        const afterBalance = await tokenV3.balanceOf(deployer.address);        expect(afterBalance).to.be.greaterThan(beforeBalance);      });      it("9. User1 should not be able to mint the token", async () => {        const amount2 = 5000;        await expect(tokenV3.connect(user1).mint(user1.address, amount2)).to.be          .reverted;      });      it("10. Only owner should be able to burn the token", async () => {        const amount2 = amountToEther(2000);        const beforeTotalSupply = await tokenV3.totalSupply();        const beforeBalance = await tokenV3.balanceOf(deployer.address);        const tx = await tokenV3          .connect(deployer)          .burn(deployer.address, amount2);        await tx.wait();        const afterTotalSupply = await tokenV3.totalSupply();        expect(afterTotalSupply).to.be.lessThan(beforeTotalSupply);        const afterBalance = await tokenV3.balanceOf(deployer.address);        expect(afterBalance).to.be.lessThan(beforeBalance);      });      it("11. User1 should not be able to burn the token", async () => {        const amount2 = amountToEther(2000);        await expect(tokenV3.connect(user1).burn(user1.address, amount2)).to.be          .reverted;      });      it("12. User1 should be able to buy the tokens", async () => {        const beforeBalance = await tokenV3.balanceOf(user1.address);        const valueOption = { value: amountToEther(2) };        await tokenV3.connect(user1).buy(valueOption);        const afterBalance = await tokenV3.balanceOf(deployer.address);        expect(afterBalance).to.be.greaterThan(beforeBalance);      });      describe("Phase 3: Testing Withdraw Functionality", () => {        it("Only owner should be able to withdraw the amount", async () => {          const beforeBalance = await tokenV3.getContractBalance();          const amount = amountToEther(1);          const tx = await tokenV3.connect(deployer).withdraw(amount);          await tx.wait();          const afterBalance = await tokenV3.getContractBalance();          expect(afterBalance).to.be.lessThan(beforeBalance);        });      });    });  }); |

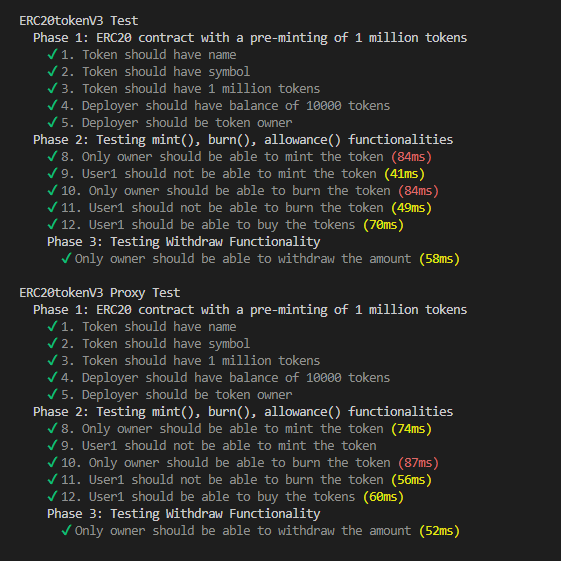
**Step 23. Run the tests and verify that all pass**

Run

|  |
| --- |
| npx hardhat test |

You should see a message that indicates all of them pass, including the upgrade of the ERC20TokenV2 to ERC20TokenV3.

Consult the screenshot below for how it should look.



**\*\*\*Step 24. Script to Prepare the new upgrade to V2**

Inside the scripts folder, create the file “prepareUpgradeForV2.js” and copy the below code into it.

|  |
| --- |
| // scripts/prepareUpgradeForV2.js const { ethers, upgrades } = require("hardhat");  async function main() {    const proxyAddress = "0x6eE935B6B58A428Fd550b9F38D63d9b80E013D35";    // This is the proxy address NOT the admin of the proxy.    const ERC20TokenV2 = await ethers.getContractFactory("ERC20TokenV2");    console.log("Preparing upgrade...");    const erc20TokenV2 = await upgrades.prepareUpgrade(      proxyAddress,      ERC20TokenV2    );    console.log("ERC20TokenV2 at:", erc20TokenV2);  }  main()    .then(() => process.exit(0))    .catch((error) => {      console.error(error);      process.exit(1);    }); |

You will need to change 0x6f0e575226a2f3d12e2755B2898767d25321d079 to your proxy address.

Note that it is the Proxy Address, not the Administrator of the Proxy (Proxy Admin).

Run

|  |
| --- |
| npx hardhat run --network goerli scripts/ prepareUpgradeForV2.js |

Copy down the new Implementation Address, in this example it will be 0xbe0b67cf6301E9B061b34FFae02D45ed671A7475

Now run below script to verify the new Implementation Contract.

|  |
| --- |
| npx hardhat verify --network goerli 0xbe0b67cf6301E9B061b34FFae02D45ed671A7475 |

**Step 25. Optional… Another way to Get the Proxy Admin Address**

As stated in an earlier step…

The easiest way to find the Proxy Admin address is to look in look in goerli.json in the .oppenzeppelin folder. At the bottom of the file is

|  |
| --- |
| "admin": {  "address": "0x697492C144FfE1Fd32dcc9CC47005A968a94694B",  "txHash": "0xbfccfa1d79a55fbdb8eb28fb7cc2143760f6263020ca121f3d00716f1a46534d" } |

Note the Address down.

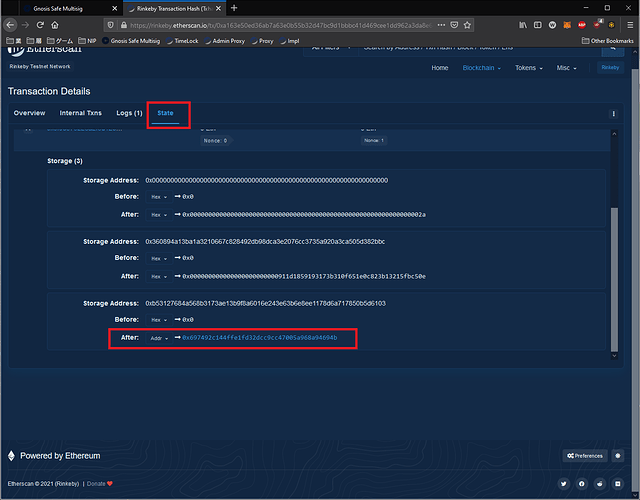
You can also find the Proxy Admin address by going to the Proxy Address on goerli etherscan, and finding the Contract Creation Transaction.

After going to that transaction, click the State tab.

In the 3rd State, click the drop-down arrow to see 3 Storage addresses.

The last storage address is the Proxy Administrator. You will need to change this number from hex to Address.

Consult the screenshot below for guidance.



At one point in time, you could find the Administrator of the Proxy’s address via the Proxy’s read functions. But in the newer version this is not available.

**Step 26. Get the data for the Upgrade Transaction**

Go to the Proxy Admin’s page in Etherscan.

Click Write Contract

Connect to Web3

Select function 4, upgrade

proxy (address) = The Proxy Address, for this example it is 0x6f0e575226a2f3d12e2755B2898767d25321d079

implementation (address) = The New Implementation Address, for this example it is 0xbe0b67cf6301E9B061b34FFae02D45ed671A7475

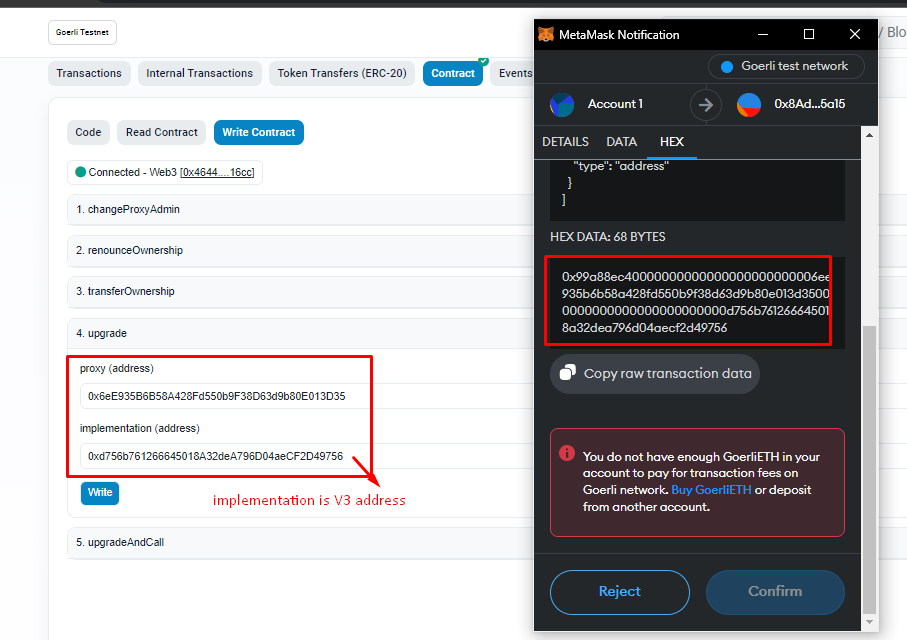
Refer to the screenshot below for guidance.

***Click write, but DO NOT confirm.***

In the MetaMask tab, click Data.

Find the Hex Data field, and copy down the Hex Data.

Refer to the screenshot below for how it should look.



For this example it is…

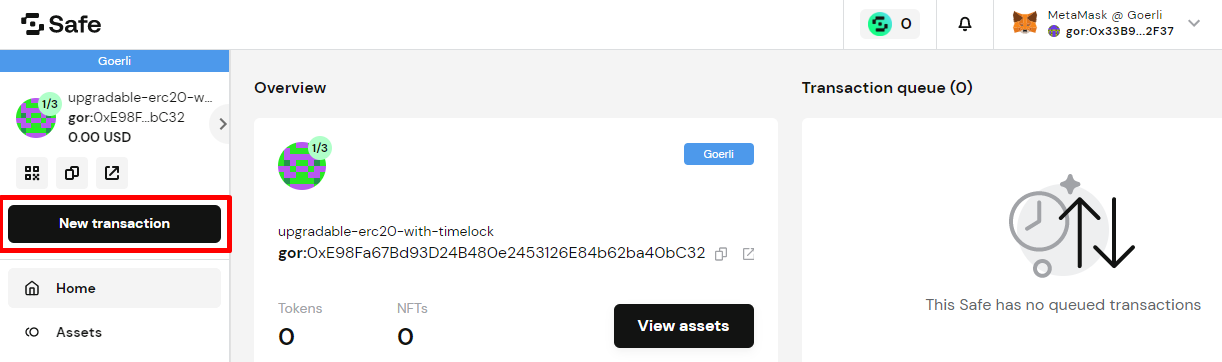
0x99a88ec40000000000000000000000006ee935b6b58a428fd550b9f38d63d9b80e013d350000000000000000000000000878043153daed34a76cb85d5ddfcdda678d6bde

Cancel the transaction

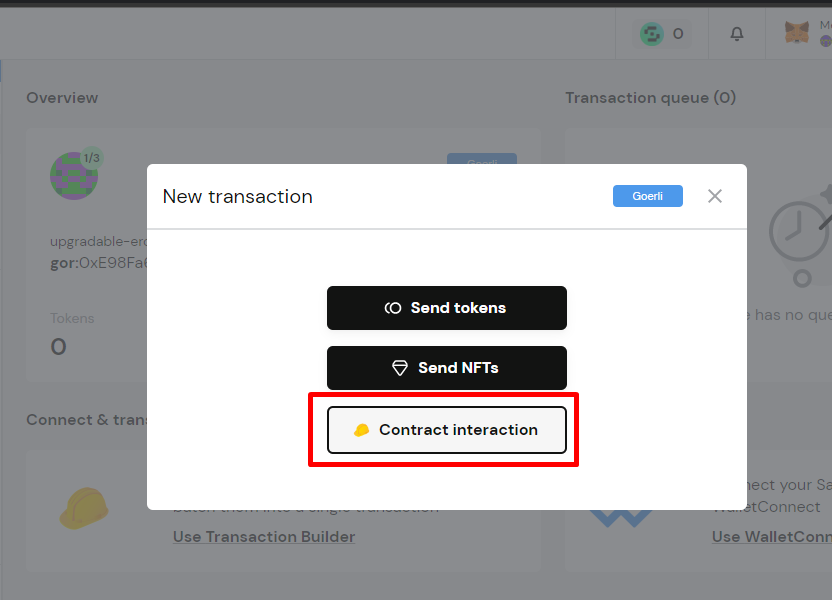
**Step 27. Schedule the Transaction to Upgrade the Contract with the Gnosis Safe using the TimeLock.**

Go to your safe in the Gnosis app

1)Click New Transaction



2)Contract Interaction



3)In the “Contract Address” field, paste your TimeLock address

For this example it is 0xe0Ea7A063acB21112528ce5d1b85D8261E5849bb

4) Copy the timelock abi from hardhat artifacts and paste it here.

From the Method dropdown box below, select “schedule”.

Fill out the fields as shown below

target (address) = Your Proxy Admin Address (0x697492C144FfE1Fd32dcc9CC47005A968a94694B)

value (uint256) = 0

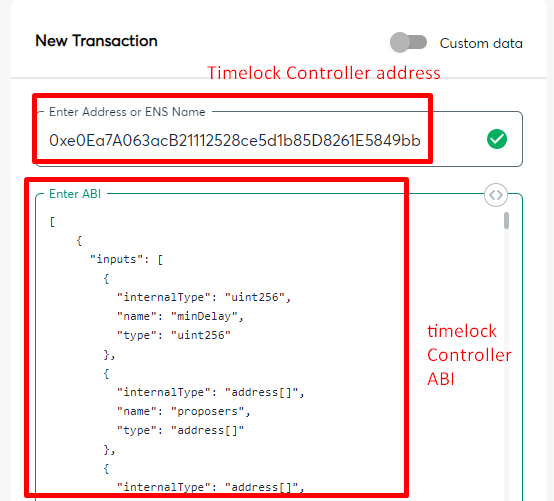
data (bytes) = The data for the Upgrade Transaction (0x99a88ec40000000000000000000000006f0e575226a2f3d12e2755b2898767d25321d079000000000000000000000000be0b67cf6301e9b061b34ffae02d45ed671a7475)

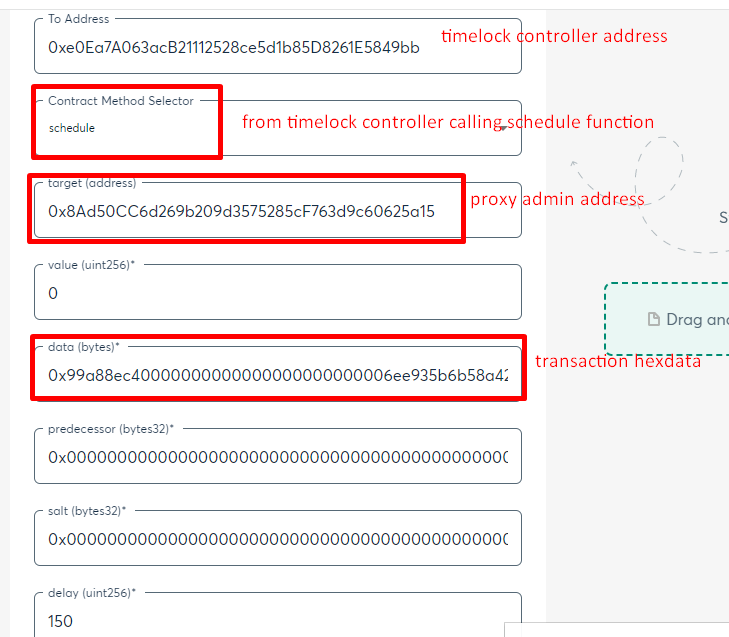
predecessor (bytes32) = 0x0000000000000000000000000000000000000000000000000000000000000000

salt (bytes32) = 0x0000000000000000000000000000000000000000000000000000000000000000

delay (uint256) = 150

Refer to the Screenshot below for how this should look.





For explanations on each field, please refer back to the TimeLock section as these are TimeLock functions.

Click Review

If you have done something incorrectly, it will warn you, back up and fix the error by reviewing the steps. It is likely the data.

Click Submit

Click Confirm in MetaMask.

**Step 28. Verify that the transaction was scheduled in the TimeLock**

While waiting 2.5 Minutes for the transaction to go through the TimeLock, go to the TimeLock’s address in etherscan.

Click the Events tab.

The most recent transaction is the “CallScheduled” one.

In that transaction log you can investigate the data passed into the CallScheduled function.

In [topic1] is the ID of the operation.

You can use this Operation ID in order to cancel a scheduled transaction.

You can use this Operation ID in the Predecessor field of Schedule/Execute functions.

See the screenshot below displaying this transaction.



**Step 29. Execute the Transaction to Upgrade the Contract with the Gnosis Safe using the TimeLock.**

Make sure your delay timer has passed, in this example it was 2.5 minutes.

Go to your safe in the Gnosis app

1)Click New Transaction

2)Contract Interaction

3)In the “Contract Address” field, paste your TimeLock address

4)For this example it is 0xa0ea1e6Bea3BE3f9565E557AB6Aefe22D1e0e74e

5)Copy the timelock abi from hardhat artifacts and paste it here.

6)From the Method dropdown box below, select “execute”

7)Fill out the fields as shown below

Value (ether) = 0

target (address) = Proxy Admin Address

(0x697492C144FfE1Fd32dcc9CC47005A968a94694B)

value (uint256) = 0

data (bytes) = The data for the Upgrade Transaction

(0x99a88ec40000000000000000000000006f0e575226a2f3d12e2755b2898767d25321d079000000000000000000000000be0b67cf6301e9b061b34ffae02d45ed671a7475)

predecessor (bytes32) = 0x0000000000000000000000000000000000000000000000000000000000000000

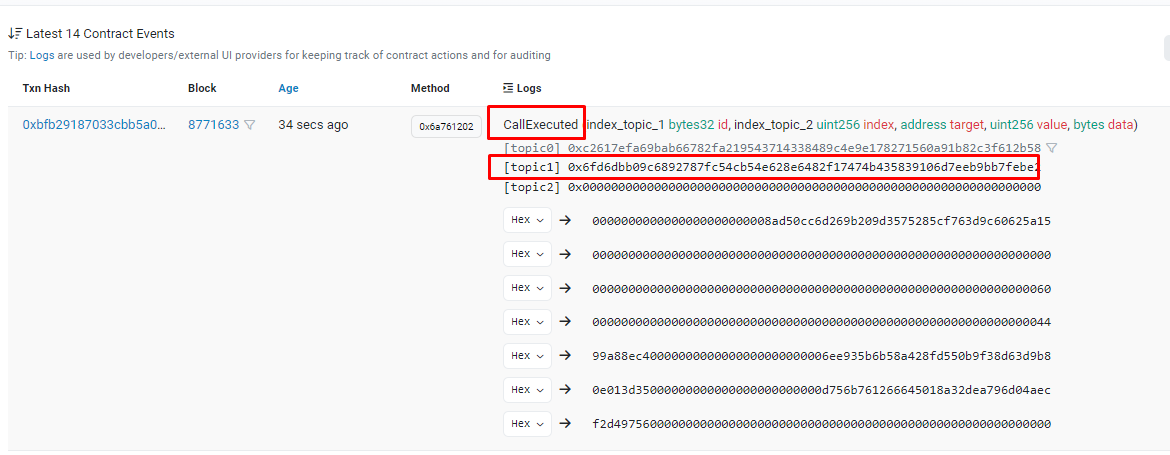
salt (bytes32) = 0x0000000000000000000000000000000000000000000000000000000000000000

Click Review

If you have done something incorrectly, it will warn you, back up and fix the error by reviewing the steps. It is likely the data.

Click Submit

Click Confirm in MetaMask.



**Step 30. Verify that the New Implementation contract is in place.**

Navigate to your Proxy Address in Etherscan. (0x6f0e575226a2f3d12e2755B2898767d25321d079 for this example).

The Contract Address 0x6f0e575226a2f3d12e2755B2898767d25321d079 page

allows users to view the source code, transactions, balances, and analytics for the contract address. Users can also interact and make transactions to the contract directly on…

Click Read as Proxy.

A message should say " ABI for the implementation contract at 0xbe0b67cf6301e9b061b34ffae02d45ed671a7475, using the EIP-1967 Transparent Proxy pattern.

Previously recorded to be on 0x911d1859193173b310f651e0c823b13215fbc50e."

Verify that the Implementation Address is the one that was created when you ran the prepareupgrade.js script.

Refer to the below screenshot for guidance.

