Tutorial: Verification Registry in Solidity

Interacting With a Verification Registry Written in Solidity Suggest Edits

As discussed in the Smart Contract Patterns section of the documentation, using smart contracts to manage verification registries provides a trustless and decentralized mechanism for individuals and organizations to look up verification statuses, verification credentials, revoke credentials, and more. Any smart contract and blockchain can be used, but we will focus on Solidity in this example.

Solidity is the language for the Ethereum Virtual Machine. Solidity is the most widely used blockchain programming language.

First, we'll use the example verifier registry contract that Verite has created. You can find that contract here . While this contract should be treated as an example and you should create your own as needed by your specifications when deploying to production, it covers the necessary actions that a verification registry would need to take.

Setting Up Hardhat

Hardhat is a smart contract development tools platform. It provides command-line functionality, the ability to run a local Ethereum test node, and more. We'll create a new project using Hardhat.

The below will walk you through the set-up instructions, but their documentation will have the most up-to-date instructions, so click this link to follow their getting started guide to create a blank project.

Prerequisites:

- Node.js >= 12.0
- npm or yarn

Assuming the prerequisites are met, you can create a new Hardhat project by creating a new folder:

```
mkdir registry-contract && cd registry-contract
```

Now, you'll need to initialize the project and install Hardhat as a development dependency:

Shell npm init npm i -D hardhat Now, it's time to set up an empty Hardhat project. Run the following command:

```
npx hardhat
```

You'll be prompted to select either a sample project or "Create an empty hardhat.config.js". Select the empty hardhat.config.js option.

Now, you'll want to install some tools to help you work with your smart contract. Run the following:

```
npm install --save-dev @nomiclabs/hardhat-ethers ethers @nomiclabs/hardhat-waffle ethereum-waffle chai
```

Next, you'll need to update your hardhat.config.js file. Add the following line to the top:

sol require("@nomiclabs/hardhat-waffle") In order to work on the smart contract, you'll need to create a contracts folder in the root of the project directory. Once you've done that, you can create a new file called VerificationRegistry.sol . For simplicity, you can copy over the code from the example Verite registry contract .

You'll notice this contract references another contract (an interface contract), and it imports some libraries from OpenZeppelin. So, we need to do two things. We need to install OpenZeppelin, and we need to copy over the interface contract. We'll explain why the interface contract is important momentarily. First, let's install OpenZeppelin's library of contracts:

```
npm install @openzeppelin/contracts
```

Now, you can create a new file in the contracts folder called IVerificationRegistry.sol . Copy the code from here and paste it into that file.

Before compiling the contracts, check the Solidity version in the contracts. You'll need to make sure it matches the version in the hardhat.config.js file. If it doesn't, simply update the config file to match.

Now, you can compile the contract with Hardhat by running the following command:

```
npx hardhat compile
```

Testing And Interacting With The Contract

The best way to explore how to interact with a registry contract is to write tests for it. While tests are not a 100% direct comparison to using the contract in an app, they are very close and it's relatively easy to extract test code and use it in your app.

So, to test the contract, create a folder inside the contracts folder called test . Then inside that folder create a file called RegistryTest.js . Instead of copying code over, we're going to write the code so we can understand all the important functions within the Verification Registry contract.

In your test file, require the following two libraries:

sol const { ethers } = require("hardhat") const { expect } = require("chai") The ethers library is included with Hardhat to make things simpler. Ethers.js allows you to interact with smart contracts and wallets. The chai library is a testing library we will use in our tests.

Let's write a describe statement to wrap our set of tests:

sol describe("VerificationRegistry", function () {}) When writing tests, you would groups test cases logically within as many describe statements as you need. For the sake of this tutorial, we're going to use just one describe statement and we'll put all of our tests inside that statement.

Just remember that each test we write should be added inside the describe statement.

Our first test will focus on creating a random wallet address that will be used for signing. In the real world, this address would be pre-determined and would not be created randomly.

sol // create a wallet to generate a private key for signing verification results const mnemonic = "announce room limb pattern dry unit scale effort smooth jazz weasel alcohol" const signer = ethers.Wallet.fromMnemonic(mnemonic)

// get a random subject address that will be used for verified subject tests let subjectAddress it("Should find a random address to use as a subject to verify", async function () { const addresses = await ethers.getSigners() const r = Math.floor(Math.random() * addresses.length) subjectAddress = await addresses[r].getAddress() expect(subjectAddress).to.not.equal(null || undefined) }) Note that the address is stored in a global variable called subjectAddress .

Now, let's add a test for deploying the contract:

sol // deploy the contract, which makes this test provider the contract's owner let verificationRegistry let contractOwnerAddress it("Should deploy", async function () { const deployer = await ethers.getContractFactory("VerificationRegistry") verificationRegistry = await deployer.deploy() await verificationRegistry.deployed() contractOwnerAddress = verificationRegistry.deployTransaction.from expect(contractOwnerAddress).to.not.be(null || undefined) }) Again, we use global variables to hold data that will be used in tests that we have yet to write. We could actually run our tests now and start to see if they are passing. If you'd like to, you can run:

```
npx hardhat test
```

You should see that both tests pass.

getVerifier

Let's continue by writing the next test. In this test, we get to leverage one of the core functions of the registry contract. This function checks if a given address is a verifier.

sol it("Should not find a verifier for an untrusted address", async function () { await expect(verificationRegistry.getVerifier(contractOwnerAddress)).to.be .reverted }) Let's break this down. When we deploy the contract, it gets assigned to the variable verificationRegistry . This allows us to execute functions on the contract. So, in this test, we are executing the function called getVerifier . We expect this function call to fail and be reverted because the contract owner (the address that deployed the contract) has not been added as a verifier.

addVerifier

Now that we've seen the contract properly reject an address that is not listed as a verifier, let's see how we can add a new verifier. We're going to add the contract owner address as a verifier.

sol // create a test verifier const testVerifierInfo = { name: ethers.utils.formatBytes32String("Circle Internet Financial"), did: "did:web:circle.com", url: "https://www.circle.com/en/about-circle", signer: signer.address }

// make the contract's owner a verifier in the contract it("Should become a registered verifier", async function () { const setVerifierTx = await verificationRegistry.addVerifier(contractOwnerAddress, testVerifierInfo) // wait until the transaction is mined const tx = await setVerifierTx.wait() expect(tx.from).to.equal(contractOwnerAddress) }) Here, the addVerifier function on the contract takes two arguments: the address to add as a verifier and the verifier info. The verifier info object is defined by the contract requirements. It's possible to create your own contract that expects and returns different fields.

You'll see the signer in the above example is defined as signer.address . Remember, signer is a global variable we store earlier in testing.

isVerifier

This next test shows how to use the isVerifier function on the contract. Unlike the getVerifier function, this one should not throw and revert. Instead, it should return a boolean every time. It simply takes in an address and will tell you if that address is a valid verifier.

sol it("Should ensure owner address maps to a verifier", async function () { const isVerifier = await verificationRegistry.isVerifier(contractOwnerAddress) expect(isVerifier).to.be.true }) In the test, we again use the contract owner address. Since we added that address as a verifier in the previous step, the response should be true .

isVerified

This function is another one that returns a simple boolean. It checks if an address has already been verified.

sol it("Should see the subject has no registered valid verification record", async function () { const isVerified = await verificationRegistry.isVerified(contractOwnerAddress) expect(isVerified).to.be.false }) Our expect statement here says that the result should be false. This is because we have not verified any records yet.

getVerifierCount

The verification registry contract can be implemented as a global standard, by a consortium, by a single organization, or even by an individual. So, there may be many verifiers listed on the contract. This is where the getVerifierCount function can come in handy.

sol it("Should have one verifier", async function () { const verifierCount = await verificationRegistry.getVerifierCount() expect(verifierCount).to.be.equal(1) }) In our tests, we have still only added the contract owner address, so when we call getVerifierCount , we expect there to be one verifier.

Now that we have one verifier on the registry contract, we can call getVerifier again, and we should have success this time:

sol it("Should find a verifier for owner address", async function () { const retrievedVerifierInfo = await verificationRegistry.getVerifier(contractOwnerAddress) expect(retrievedVerifierInfo.name).to.equal(testVerifierInfo.name) expect(retrievedVerifierInfo.did).to.equal(testVerifierInfo.did) expect(retrievedVerifierInfo.url).to.equal(testVerifierInfo.url) })

updateVerifier

Verifier information can change, so it's important to be able to update verifiers. The updateVerifier function takes care of this.

sol it("Should update an existing verifier", async function () { testVerifierInfo.url = "https://circle.com" const setVerifierTx = await verificationRegistry.updateVerifier(contractOwnerAddress, testVerifierInfo) // wait until the transaction is mined await setVerifierTx.wait() const retrievedVerifierInfo = await verificationRegistry.getVerifier(contractOwnerAddress) expect(retrievedVerifierInfo.url).to.equal(testVerifierInfo.url) }) The function takes the verifier's address and the new verifier info object.

removeVerifier

When a verifier needs to be removed, the removeVerifier function can be called. As with all of the functions that update the state of this contract, this function can only be called by the contract owner.

sol it("Should remove a verifier", async function () { const removeVerifierTx = await verificationRegistry.removeVerifier(contractOwnerAddress) // wait until the transaction is mined await removeVerifierTx.wait() const verifierCount = await verificationRegistry.getVerifierCount() expect(verifierCount).to.be.equal(0) }) The removeVerifier function takes the verifier address as an argument and then removes that verifier.

registerVerification

This is where things might get a little confusing, so let's start by explaining how Solidity can validate signed data. The EIP712 standard defines how Solidity contracts can manage signed and hashed data. EIP712's summary aptly describes its purpose:

Signing data is a solved problem if all we care about are bytestrings. Unfortunately in the real world we care about complex meaningful messages. Hashing structured data is non-trivial and errors result in loss of the security properties of the system.

As such, the adage “don’t roll your own crypto” applies. Instead, a peer-reviewed well-tested standard method needs to be used. This EIP aims to be that standard. The example registry contract implements EIP712, but before we can use the standard, we have to structure our data properly so that it can be sent to the contract.

sol // get a deadline beyond which a test verification will expire // note this uses an external scanner service that is rate-throttled // add your own API keys to avoid the rate throttling // see https://docs.ethers.io/api-keys/ let expiration = 9999999999 it("Should create a deadline in seconds based on last block timestamp", async function () { const provider = ethers.getDefaultProvider() const lastBlockNumber = await provider.getBlockNumber() const lastBlock = await provider.getBlock(lastBlockNumber) expiration = lastBlock.timestamp + 300 })

// format an EIP712 typed data structure for the test verification result let domain, types, verificationResult = {} it("Should format a structured verification result", async function () { domain = { name: "VerificationRegistry", version: "1.0", chainId: 1337, verifyingContract: await verificationRegistry.resolvedAddress } types = { VerificationResult: [{ name: "schema", type: "string" }, { name: "subject", type: "address" }, { name: "expiration", type: "uint256" }] } verificationResult = { schema: "circle.com/credentials/kyc", subject: subjectAddress, expiration: expiration } }) There's a lot going on here, so let's break it down. As the comments suggest, we first have to define an expiration timestamp at some point beyond when the verification record will expire.

Next, we are defining three global variables: domain , types , and verificationResult . We then assign values to those variables.

The domain variable is assigned to an object containing information about the registry. The types variable is defining the VerificationResult type. The verificationResult is defining the shape of the verification result itself using test data.

It's important to note that the verificationResult comes from the actual verifier completing their verification process. SeeVerifying Credentials andVerification for more.

Next, we need to create a digest and sign it:

sol let signature it("Should sign and verify typed data", async function () { signature = await signer._signTypedData(domain, types, verificationResult) const recoveredAddress = ethers.utils.verifyTypedData(domain, types, verificationResult, signature) expect(recoveredAddress).to.equal(testVerifierInfo.signer) }) In the above test code, we took our verification result object, signed it, then verified it.

Now, we get to the registerVerification function from the registry contract. This function takes the signed data and registers it with the contract, storing it on-chain:

sol it("Should register the subject as verified and create a Verification Record", async function () { const verificationTx = await verificationRegistry.registerVerification(verificationResult, signature) await verificationTx.wait() const verificationCount = await verificationRegistry.getVerificationCount() expect(verificationCount).to.be.equal(1) }) The registerVerification function takes the verificationResult and the signature data as arguments. By combining these two elements, verifications can be validated at any point in the future. After running this, if we were to then call the isVerified function from earlier using the credential owner's address as an argument, the result would be true:

sol it("Should verify the subject has a registered and valid verification record", async function () { const isVerified = await verificationRegistry.isVerified(subjectAddress) expect(isVerified).to.be.true })

getVerificationsForSubject

Now that you have been able to add verifications to the registry contract, it's possible to get all the verifications for a given wallet address.

sol let recordUUID = 0 it("Get all verifications for a subject address", async function () { const records = await verificationRegistry.getVerificationsForSubject(subjectAddress) recordUUID = records[0].uuid expect(records.length).to.equal(1) }) We define recordUUID for testing purposes only here, which you'll see in the next section. But outside of that, this function is very simple. It returns an array of verifications for a subject.

getVerificationsForVerifier

Similarly, the getVerificationsForVerifier function will return an array of verifications completed by a give verifier address.

sol it("Get all verifications for a verifier address", async function () { const records = await verificationRegistry.getVerificationsForVerifier(contractOwnerAddress) expect(records[0].uuid).to.equal(recordUUID) expect(records.length).to.equal(1) })

getVerification

Finally, we have the getVerification function. This function will return a single verification based on that verification's identifier.

sol it("Get a verification using its uuid", async function () { const record = await verificationRegistry.getVerification(recordUUID) expect(ethers.utils.getAddress(record.subject)).not.to.throw }) You can see we use the recordUUID variable from earlier in our tests to fetch the specific verification. In a real environment, you would have that identifier stored somewhere like in a database.

Conclusion

The above is a roadmap, not a prescription. If you are building a verification registry on Ethereum or an EVM-compatible chain, this guide should help you understand the types of functions that may be necessary for full implementation.

Because this is a roadmap, you may find yourself implementing fewer functions or more functions. You may add additional functionality to the contract. You may decide to implement the registry in a global capacity or restricted to a single verifier. There are plenty of options, but hopefully, this guide will act as a nice kickstart to your eventual implementation. Updated5 months ago