Tutorial: Creating a Registry Contract in Solidity

Creating a Registry Contract in Solidity Suggest Edits

When using the smart contract pattern for verifications, you can use any blockchain you would like. For the sake of this guide, we will focus on Solidity , which is the programming language for the Ethereum Virtual Machine.

Setup

For this tutorial, we will use Hardhat. Hardhat provides tooling to set up your development environment, test contract code, deploy contracts, and more. Follow this guide to get started with Hardhat.

Once your Hardhat project is set up, it's time to create our registry contract. In the contracts folder.

Creating The Contract

In the contracts folder, create a new file called VerificationRegistry.sol . In that file, we want to specify the version of Solidity we're using. We're going to be using OpenZeppelin contracts soon, so we need to make sure our contract uses the same Solidity version as OpenZeppelin, which is currently 0.8.0 . A the top of your file add:

sol pragma solidity ^0.8.0; Now, before we move on, let's install OpenZeppelin's library of contracts . This will allow us to pull in dependent contracts that are safe and audited without us having to write them ourselves. To install, run the following:

npm install @openzeppelin/contracts

Once the dependencies are installed, we can import the appropriate libraries into our contract. Let's take a second to understand what those libraries are.

The first library we will import into our contract is the Ownable library. This one is designed to make it easier to lock function calls down to the owner of the contract.

The next library we will import is the ECDSA library. This library is used to help us decode signed and hashed data.

Finally, we will import the EIP712 library. This is the most important library in our contract since it is foundational to how we will verify credentials. For more on this, be sure to reference the Smart Contract Patterns section .

To import these libraries, add the following below the Solidity version in our contract:

sol pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/Ownable.sol"; import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol"; import "@openzeppelin/contracts/utils/cryptography/draft-EIP712.sol"; There's one additional library we want to import. This one is an interface library that allows us to more easily map the types necessary in our contract variables and functions. For brevity, we're not going to write the interface itself. We'll just copy it from the existing example from Circle. You can grab a copy of the interface library contract here .

Create a new contract file in your contracts folder and call it IVerificationRegistry.sol . Paste the contents of the example file linked above into that contract, then return to your VerificationRegistry.sol file.

Add one more import to the Verification Registry contract:

sol import "./IVerificationRegistry.sol"; Now, we're ready to write the contract!

State Variables

Our Verification Registry requires some contract state. If you remember from the Smart Contract Patterns part of the documentation, this means either a verifier or a subject will have to pay gas to update the state of the contract when a new verification is added to the registry.

Let's go ahead and start creating the contract and add in some variables. First, we need to define the contract itself. We do that like so:

sol contract VerificationRegistry is Ownable, EIP712("VerificationRegistry", "1.0"), IVerificationRegistry {

} There are a couple of interesting things going on already. First, we are using the EIP712 library to help us with signature validation and hashing. When we use that library, it requires two parameters: a name and a version.

Next, we are extending our contract to use the types defined in our interface contract.

Ok, now let's define some variables. We'll drop them all below and then we'll talk through each.

sol // Verifier addresses mapped to metadata (VerifierInfo) about the Verifiers. mapping(address => VerifierInfo) private _verifiers;

// Verifier signing keys mapped to verifier addresses mapping(address => address) private _signers;

// Total number of active registered verifiers uint256 _verifierCount;

// All verification records keyed by their uuids mapping(bytes32 => VerificationRecord) private _verifications;

// Verifications mapped to subject addresses (those who receive verifications) mapping(address => bytes32[]) private _verificationsForSubject;

// Verifications issued by a given trusted verifier (those who execute verifications) mapping(address => bytes32[]) private _verificationsForVerifier;

// Total verifications registered (mapping keys not being enumerable, countable, etc) uint256 private _verificationRecordCount; The comments above each variable help explain what their purpose is, but we'll walk through it here as well.

The _verifiers variable is a mapping of information tied to verifiers. If we look at the IVerificationRegistry.sol file, we can see the VerifierInfo type looks like this:

sol struct VerifierInfo { bytes32 name; string did; string url; address signer; } We won't copy over every type into this tutorial, but this shows how you can see the shape of the expected data for various variables.

The next variable we have is _signers . This is a mapping of the keys verifiers use to sign the verification records. It's how the Verification Registry contract can validate the verifier is who they say they are.

Next, we have a simple count to help track the number of verifiers: _verifierCount .

sol uint256 _verifierCount; We, of course, have to track the actual verifications as well. That is done in the mapping called _verifications .

sol mapping(bytes32 => VerificationRecord) private _verifications; In this example, we are mapping the verification identifier (a UUID) to the verification record. You may want to customize this mapping to your specification, but UUIDs are relatively collision resistant and tend to be good unique identifiers for things like this.

Next, we have the mapping of verifications for a specific subject:

sol mapping(address => bytes32[]) private _verificationsForSubject; This mapping requires the subject's wallet address and then maps it to an array of UUIDs.

Similarly, we track the verifications for verifiers with:

sol mapping(address => bytes32[]) private _verificationsForVerifier; And finally, we have a simple count for total verifications:

sol uint256 private _verificationRecordCount; Of course, the contract is nothing if it's just a bunch of state variables. Let's dive into writing the functions that will define the contract.

Contract Functions

The first function we'll define is to add a verifier. Without a verifier, the registry doesn't work. This function should look like this:

sol function addVerifier(address verifierAddress, VerifierInfo memory verifierInfo) external override onlyOwner { require(_verifiers[verifierAddress].name == 0, "VerificationRegistry: Verifier Address Exists"); _verifiers[verifierAddress] = verifierInfo; _signers[verifierInfo.signer] = verifierAddress; _verifierCount++; emit VerifierAdded(verifierAddress, verifierInfo); } Many of the functions on this contract can only be called by the owner of the contract (the address that deployed it). This function is no exception. The onlyOwner modifier uses the OpenZeppelin Ownable library to add a check that ensures the function is being called by the contract owner.

The function takes a verifier's address and the verifier information structured as defined by the VerifierInfo type.

You'll notice the emit at the end of the function. We don't have any events defined on the VerificationRegistry.sol contract, but the example interface contract does. This is completely optional, but emitting events is a nice way to allow others to listen to those events and take some action.

The next function is a simple one that is available to anyone to call. It takes a wallet address and returns a boolean

indicating whether the address is a verifier or not.

sol function isVerifier(address account) external override view returns (bool) { return _verifiers[account].name != 0; } The next function is equally simple. It is also callable by anyone, not just the contract owner. It returns the total number of registered verifiers:

sol function getVerifierCount() external override view returns(uint) { return _verifierCount; } We will also want to get information about specific verifiers—not just their wallet address, but their full VerifierInfo record. To do that, we can create the following function:

sol function getVerifier(address verifierAddress) external override view returns (VerifierInfo memory) { require(_verifiers[verifierAddress].name != 0, "VerificationRegistry: Unknown Verifier Address"); return _verifiers[verifierAddress]; } This function is another read-only function, and it can be called by anyone. It takes the verifier's wallet address and returns the full verifier info record.

Verifier info can change, so we need a function to update that information. This function can only be called by the contract owner, but it allows for such updates:

sol function updateVerifier(address verifierAddress, VerifierInfo memory verifierInfo) external override onlyOwner { require(_verifiers[verifierAddress].name != 0, "VerificationRegistry: Unknown Verifier Address"); _verifiers[verifierAddress] = verifierInfo; _signers[verifierInfo.signer] = verifierAddress; emit VerifierUpdated(verifierAddress, verifierInfo); } This function is a complete replacement, so even if only part of the VerifierInfo record is being updated, a whole new record has to be passed into the function.

This function also emits an event called VerifierUpdated that anyone can listen for.

As you might expect, we also need to be able to remove verifiers. Again, only the contract owner can call this function.

sol function removeVerifier(address verifierAddress) external override onlyOwner { require(_verifiers[verifierAddress].name != 0, "VerificationRegistry: Verifier Address Does Not Exist"); delete _signers[_verifiers[verifierAddress].signer]; delete _verifiers[verifierAddress]; _verifierCount--; emit VerifierRemoved(verifierAddress); } This function takes the wallet address for the verifier and removes that verifier from the _verifiers mapping. It removes the signing address for the verifier and reduces the total _verifierCount .

We now move into some of the actual verification logic with our functions. The first function is a modifier that acts very much like the onlyOwner modifier mentioned before. This one is a modifier that checks if the function is called by a verifier.

sol modifier onlyVerifier() { require(*verifiers[msg.sender].name != 0, "VerificationRegistry: Caller is not a Verifier");* ; } Now, we want to allow people to call the smart contract to get the number of verification records created. We can do that with this function:

sol function getVerificationCount() external override view returns(uint256) { return _verificationRecordCount; } Next up, we get to a very important function. This function checks if a particular address has a verification record.

sol function isVerified(address subject) external override view returns (bool) { require(subject != address(0), "VerificationRegistry: Invalid address"); bytes32[] memory subjectRecords = _verificationsForSubject[subject]; for (uint i=0; i block.timestamp) { return true; } } return false; } This is where the Verification Registry contract really shines. The verification is added to the registry, but that only has to happen once. Future checks can simply rely on this function or similar functions customized to your needs.

When a single verification is needed, this next function will support that. It takes the UUID for the verification as an argument and then returns the full verification record:

sol function getVerification(bytes32 uuid) external override view returns (VerificationRecord memory) { return _verifications[uuid]; } But what about when you need all the verifications for a particular subject? Let's build a function that will return an array of verifications for a particular wallet address.

sol function getVerificationsForSubject(address subject) external override view returns (VerificationRecord[] memory) { require(subject != address(0), "VerificationRegistry: Invalid address"); bytes32[] memory subjectRecords = _verificationsForSubject[subject]; VerificationRecord[] memory records = new VerificationRecord; for (uint i=0; i<subjectRecords.length; i++) { VerificationRecord memory record = _verifications[subjectRecords[i]]; records[i] = record; } return records; } This function takes the wallet address for the subject, filters on the existing verifications for just that address, and returns the records.

Similarly, we can get an array of verifications from a single verifier with this function:

sol function getVerificationsForVerifier(address verifier) external override view returns (VerificationRecord[] memory) { require(verifier != address(0), "VerificationRegistry: Invalid address"); bytes32[] memory verifierRecords = _verificationsForVerifier[verifier]; VerificationRecord[] memory records = new VerificationRecord; for (uint i=0; i<verifierRecords.length; i++) { VerificationRecord memory record = _verifications[verifierRecords[i]]; records[i] = record; } return records; } The function takes an argument of verifier and will return all of the verifications for that verifier.

As you would imagine, verifications may be issued in mistake or become invalid for some other reason. So we need a way to revoke those in the registry. We can use a function with the onlyVerifier modifier to do so:

sol function revokeVerification(bytes32 uuid) external override onlyVerifier { require(_verifications[uuid].verifier == msg.sender, "VerificationRegistry: Caller is not the original verifier"); _verifications[uuid].revoked = true; emit VerificationRevoked(uuid); } This function takes the UUID of the record as an argument, checks to make sure the record was created by the address making the call, and then updates the revoked status to true.

Notice, the record is not removed. It is updated. This is helpful when performing audits and tracking record history.

This function also emits an event so those interested in listening for revocations can be notified.

Of course, there are times where a verification will need to be completely removed. For that, we can create a function to remove them:

sol function removeVerification(bytes32 uuid) external override onlyVerifier { require(_verifications[uuid].verifier == msg.sender, "VerificationRegistry: Caller is not the verifier of the referenced record"); delete _verifications[uuid]; emit VerificationRemoved(uuid); } Like the previous function, this one takes the UUID of the record as an argument. It verifies the record was created by the address making the call, and then it removes the record from the registry entirely.

This function emits a separate event indicating removal of a record.

The next function we will cover is a big one. All of these go hand-in-hand, but without this function, nothing else is possible. This is the function to add verifications to the registry. Let's look at the function and then walk through what's going on.

sol function registerVerification(VerificationResult memory verificationResult, bytes memory signature) external override onlyVerifier returns (VerificationRecord memory) { VerificationRecord memory verificationRecord = _validateVerificationResult(verificationResult, signature); require(verificationRecord.verifier == msg.sender, "VerificationRegistry: Caller is not the verifier of the verification"); _persistVerificationRecord(verificationRecord); emit VerificationResultConfirmed(verificationRecord); return verificationRecord; } First, let's take a look at the arguments. The function takes a verificationResult argument. This is defined in the VerificationResult type. Second, the function takes a signature . Both of these arguments are hard to conceptualize unless you see them in action, so it is recommended that you take a look at theVerification Registry in Solidity tutorial.

The tutorial referenced above will walk you through how to create a signed message that can be sent through as an argument.

As you can see, this function uses the onlyVerifier modifier, which makes sense. A verifier has to be the one to add a verification record.

The function then takes the verification result and passes it through another function for validation called _validateVerificationResult . This is a pretty massive function and we'll cover it in a few minutes. But the basics of the function are that it uses the EIP712 standard and verifies both the signature and the format of the record before inserting the record into the registry.

Next, the function persists the record by calling the _persistVerificationRecord function. That's a pretty straightforward function, so we'll document it here. You can add this in your contract anywhere you'd like as it's a helper function more than anything else.

sol function _persistVerificationRecord(VerificationRecord memory verificationRecord) internal { // persist the record count and the record itself, and map the record to verifier and subject _verificationRecordCount++; _verifications[verificationRecord.uuid] = verificationRecord; _verificationsForSubject[verificationRecord.subject].push(verificationRecord.uuid); _verificationsForVerifier[verificationRecord.verifier].push(verificationRecord.uuid); } As you can see, this function has an internal modifier, meaning it can only be called by another function on the contract. It takes the verificationRecord and it updates the _verificationRecordCount , sets the record to the _verifications mapping, adds the record as associated to the subject, and adds the record as associated to the verifier.

Finally, our original registerVerification function emits a message indicating that a new verification record was created.

One important note on this function is that it must be called by the verifier. We mentioned this earlier, but you'll see in a second that a subject can also add their own record to the registry.

This next function allows a subject to handle adding their signed verification record to the registry:

sol function _registerVerificationBySubject(VerificationResult memory verificationResult, bytes memory signature) internal returns (VerificationRecord memory) { require(verificationResult.subject == msg.sender, "VerificationRegistry: Caller is not the verified subject"); VerificationRecord memory verificationRecord = _validateVerificationResult(verificationResult, signature); _persistVerificationRecord(verificationRecord); emit VerificationResultConfirmed(verificationRecord); return verificationRecord; } This function operates similarly to the previous one, but it expects the caller to be the subject of the verification record. Outside of that, the actions are the same.

The final note on both of the above two functions is that you may want to implement additional logic based on your specific use cases. Always remember, this contract serves as a guide, but you can extend and modify it however you'd like.

Moving on, let's create a function that allows a subject to remove a verification record about themself. We previously created a function that allowed a verifier to remove a record, so this will be similar but from the subject's perspective.

sol function _removeVerificationBySubject(bytes32 uuid) internal { require(_verifications[uuid].subject == msg.sender, "VerificationRegistry: Caller is not the subject of the referenced record"); delete _verifications[uuid]; emit VerificationRemoved(uuid); } The function takes the record's UUID as an argument, it checks if the caller is the record's subject, and then it deletes the record from the registry, emitting the VerificationRemoved event at the end.

Ok, we covered the _validateVerificationResult function briefly earlier, but we didn't write out the function. Let's do that now. But remember, it's a doozy and it makes use of a lot of the internal workings we imported from the EIP712 library.

sol function _validateVerificationResult(VerificationResult memory verificationResult, bytes memory signature) internal view returns(VerificationRecord memory) {

```
bytes32 digest = _hashTypedDataV4(keccak256(abi.encode(
    keccak256("VerificationResult(string schema,address subject,uint256 expiration)"),
    keccak256(bytes(verificationResult.schema)),
    verificationResult.subject,
    verificationResult.expiration
)));

// recover the public address corresponding to the signature and regenerated hash
address signerAddress = ECDSA.recover(digest, signature);

// retrieve a verifier address for the recovered address
address verifierAddress = _signers[signerAddress];

// ensure the verifier is registered and its signer is the recovered address
require(
    _verifiers[verifierAddress].signer == signerAddress,
    "VerificationRegistry: Signed digest cannot be verified"
);

// ensure that the result has not expired
require(
    verificationResult.expiration > block.timestamp,
    "VerificationRegistry: Verification confirmation expired"
);

// create a VerificationRecord
VerificationRecord memory verificationRecord = VerificationRecord({
    uuid: 0,
    verifier: verifierAddress,
    subject: verificationResult.subject,
    entryTime: block.timestamp,
    expirationTime: verificationResult.expiration,
    revoked: false
});

// generate a UUID for the record
bytes32 uuid = _createVerificationRecordUUID(verificationRecord);
verificationRecord.uuid = uuid;

return verificationRecord;
```

} The function has some comments to help you understand what's happening, but at the end of the day, think of this function as simply verifying a signature is actually valid before adding a record to the registry. The function updates the verification record passed to it with four new pieces of information:

1. Verifier Address
2. Entry Time
3. Revoked
4. UUID

The UUID is created by calling a helper function called _createVerificationRecordUUID . You may implement a different method for identifying records, but if you choose to use UUIDs generated on the contract, this function should help:

sol function _createVerificationRecordUUID(VerificationRecord memory verificationRecord) private view returns (bytes32) { return keccak256(abi.encodePacked(verificationRecord.verifier, verificationRecord.subject, verificationRecord.entryTime, verificationRecord.expirationTime, _verificationRecordCount)); } And that's it. That's the last function. Of course, there are many more you can add when building your own contract, but this is more than enough to get you started. In fact, this is based on the demo contract Verite uses.

Wrapping Up

A verification registry contract is designed to create an easily searchable, easily verifiable central repository for verifications. This makes things reusable and faster.

This implementation is in Solidity and designed for EVM-compatible blockchains, but a verification registry contract can be implemented on any blockchain. Updated5 months ago * Table of Contents * * * Setup * * * Creating The Contract * * State Variables * * Contract Functions * * Wrapping Up