How To Use Verifiable Credentials And Verite To Build An Off-Chain NFT Allowlist

NFT allowlists are expensive. The gas involved in creating and updating allowlists on Ethereum can be prohibitive. Fortunately, verifiable Credentials and Verite makes it easy and secure to run allowlists off-chain. Suggest Edits

While Verite can help solve significantly more complex challenges than managing an early access list (i.e., an "allowlist") for NFT projects, it seemed like a fun experiment to see how well Verite could handle moving the costly process off-chain while still maintaining data integrity. Before we dive in, let's talk about what an NFT allowlist is, how it is normally managed, and what the problems are.

What is an Allowlist?

For many NFT projects, rewarding early supporters is crucial to the business model and community health. An allowlist helps give early supporters a distinct experience with unique terms and timing. By putting supporters' wallet address on a list that grants them early access to mint NFTs from the new collection, these supporters can avoid what's commonly referred to as "gaswars ". Gas wars happen when a popular NFT project drops and people (and bots) spend exorbitant amounts of gas on the Ethereum network to ensure their minting transactions go through before anyone else's to avoid missing the narrow window of availability in an oversubscribed market. Needless to say, this negatively impacts all participants because it can price people out of the collection and force them to buy the NFTs on secondary market at a higher premium. On the Ethereum mainnet, gas fees have even spiked to higher than the mint prices of the NFTs! That's a hefty surcharge.

The allowlist concept lets people on the list mint for a period of time (normally 24 hours) before the public mint. This helps keep bots out of the mint, guarantees adequate supply to everyone on that list (or at least, guarantees each member on the list access to supply while it is adequate!), and keeps gas prices relatively low. NFT projects generally frame allowlist spots as a "reward" for community participation or various kinds of non-monetary contributions.

How Are Allowlists Normally Managed?

Historically, Ethereum-based NFT allowlists have been managed on-chain. This means a mapping of wallets addresses must be added to on-chain storage, usually via a function on the NFT project's smart contract that populates that persistent mapping in storage accessible to other functions. The transaction to add these wallet addresses can be incredibly expensive, since it is literally buying and taking up additional blockspace with each additional address. It has historically ranged from a few hundred dollars to a few thousand dollars depending on the price of ETH at the time, even if the allowlist is bundled into a single transaction. This number goes much higher broken out into multiple gas-inflicting updates.

Because of the cost, projects are incentivized to set the allowlist once and never update it. Every update costs money. This can lead to errors in the list, inequity, justifiably grouchy late-comers to the community, and other problems. Additionally, allowlists can often become "static requirements": rigid patterns that get over-applied by a one-size-fits-all approach. Services likePremint have begun to change this, which introduces an economy of scale to save on gas and other features. But further improvements are possible! Projects should have the flexibility to implementdynamic requirements on whom gets added to an allowlist and how.

That's where Verifiable Credentials come in.

How To Use Verite and Verifiable Credentials

We're going to be working through an Ethereum ERC-721 NFT contract alongside a mechanism that allows us to issue verifiable credentials to participants that we want to allow on the allowlist. We'll use Hardhat to help us generate the skeleton for our smart contract code and to make it easier to test and deploy.

We'll also useSign In With Ethereum (SIWE) to handle our user sessions. We're using SIWE becauseit provides more protections and assurances than the normal "Connect Wallet" flow does.

On the front-end side of the house, we'll build a simple page that allows potential allowlist recipients to request their verifiable credential, and we'll build the actual minting functionality.

Let's get started. You'll need Node.js version 12 or above for this. You'll also need a good text editor and some knowledge of the command line.

From your command line, change into the directory where you keep all your fancy NFT projects. Then, let's clone the example app I built ahead of this tutorial to make our lives easier.

git clone https://github.com/circlefin/verite-minter-allowlist This is a repository that uses SIWE's base example app and extends it. So what you'll have is a folder for your frontend application, a folder for your backend express server, and a folder for your smart contract-related goodies.

Let's start by looking at the backend server. Open the backend/src/index.js file. There's a lot going on in here, but half of it is

related to SIWE, which is very well documented. So, we're going to gloss over those routes and just trust that they work (they do).

Request Allowlist Endpoint

Scroll down in the file until you see the route for requestAllowlist . Now, before we go any further, let me walk through a quick explanation of how this entire flow will work.

1. Project runs a web app and a server
2. Web app handles both requesting/issuing verifiable credentials associated with the allowlist and minting
3. During the minting process, a verifiable credential must be sent back to the project's server.
4. Backend handles checking to see if a wallet should receive a credential, then generating the credential.
5. Backend handles verifying that a credential sent as part of the minting process is valid.
6. If credential is valid, backend signs an EIP712 message with a private key owned by the project.
7. Signature is returned to the frontend and includes it as part of the mint function on the smart contract.

We'll dive into details on the smart contract in a moment, but that's the basic flow for the front and backends. For those who love a good diagram, we've got you covered:

Now, if we look at the route called requestAllowlist , we'll see:

JavaScript if (!req.session.siwe) { res.status(401).json({ message: "You have to first sign_in" }) return } const address = req.session.siwe.address if (!validateAllowlistAccess(address)) { res.status(401).json({ message: "You are not eligible for the allowlist" }) return }

const { subject } = await getOrCreateDidKey(address)

const issuerDidKey = await getIssuerKey() const application = await createApplication(issuerDidKey, subject) const presentation = await getPresentation(issuerDidKey, application)

res.setHeader("Content-Type", "application/json") res.json(presentation) We are using the SIWE library to make sure the user is signed in and has a valid session. This also gives us the user's wallet address. Remember, we're trusting that all the SIWE code above this route works (it does).

Next, we are checking to see if the project has determined that wallet to be eligible for the allowlist. This is a very low-tech process. Most projects ask participants to do something in order to get on the list and then they manage a spreadsheet of addresses. In this code example, we have a function called validateAllowlistAccess() that checks a hardcoded array of addresses from a config file:

JavaScript const config = JSON.parse(fs.readFileSync("../config.json"))

const validateAllowlistAccess = (address) => { return config.addressesForAllowlist.includes(address) } Next, we need to create a DID (decentralized identifier) key for the associated wallet (or we need to look up an existing DID key). In a perfect world, we'd be using a built-in credential wallet integration with the user's Ethereum wallet, but since we don't have that, we're going to manage a delegated key system. The system works like this:

1. Project checks to see if there is a DID key for the wallet in question in the database (note: the database here is just disk storage, but can be anything you'd like).
2. If there is a DID key, project uses that key for Verite functions.
3. If there is no DID key, project generates one and adds the mapping to the database.

That's happening here:

JavaScript const { subject } = await getOrCreateDidKey(address) And the getOrCreateDidKey() function looks like this:

JavaScript const getOrCreateDidKey = async (address) => { const db = JSON.parse(fs.readFileSync("db.json")) let keyInfo = db.find((entry) => entry.address === address) if (!keyInfo) { const subject = randomDidKey(randomBytes) subject.privateKey = toHexString(subject.privateKey) subject.publicKey = toHexString(subject.publicKey) keyInfo = { address, subject } db.push(keyInfo) fs.writeFileSync("db.json", JSON.stringify(db)) }

return keyInfo } As you can see, our database is making use of the always fashionable file system. We look up the key or we generate a new one using Verite's randomDidKey function. We then convert the public and private key portion of the payload to hex strings for easier storage.

Ok, moving on. Next, we grab the issuer key. This is a DID key that is associated with the project.

JavaScript const issuerDidKey = await getIssuerKey() Much like the function to get the user's DID key, the getIssuerKey function just does a look up in the DB and returns the key. Remember to always protect your keys, kids. Even though these keys are exclusively for signing and issuing credentials, you should protect them as if they could spend your ETH.

JavaScript const getIssuerKey = async () => { let issuer = JSON.parse(fs.readFileSync("issuer.json")) if (!issuer.controller) {

issuer = randomDidKey(randomBytes) issuer.privateKey = toHexString(issuer.privateKey) issuer.publicKey = toHexString(issuer.publicKey) if (!issuerDidKey.signingKey) { const randomWallet = ethers.Wallet.createRandom() const privateKey = randomWallet._signingKey().privateKey issuerDidKey.signingKey = privateKey } fs.writeFileSync("issuer.json", JSON.stringify(issuer)) }

return issuer } As you can see, in addition to creating a DID key or fetching a DID key with this function, we are creating a signing key using an ETH wallet. This will be the same key we use to deploy the smart contract and sign a message later. Stand by for disclaimers!

Next, we call a function called createApplication .

JavaScript const createApplication = async (issuerDidKey, subject) => { subject.privateKey = fromHexString(subject.privateKey) subject.publicKey = fromHexString(subject.publicKey) const manifest = buildKycAmlManifest({ id: issuerDidKey.controller }) const application = await composeCredentialApplication(subject, manifest) return application } This function includes some helpers to convert the DID key private and public keys back from hex strings to buffers. The function then uses the buildKycAmlManifest function from the Verite library to build a manifest that will be used in the credential application. It should be noted that I'm using the KycAmlManifest but you could create your own manifest that more closely mirrors adding someone to an allowlist. The KycAmlManifest fit closely enough for me, though.

Finally, the manifest is used and passed into the Verite library function composeCredentialApplication and the application is returned.

When the application is built, we now call a function called getPresentation :

JavaScript const getPresentation = async (issuerDidKey, application) => { issuerDidKey.privateKey = fromHexString(issuerDidKey.privateKey) issuerDidKey.publicKey = fromHexString(issuerDidKey.publicKey)

const decodedApplication = await validateCredentialApplication(application)

const attestation = { type: "KYCAMLAttestation", process: "https://verite.id/definitions/processes/kycaml/0.0.1/usa", approvalDate: new Date().toISOString() }

const credentialType = "KYCAMLCredential"

const issuer = buildIssuer(issuerDidKey.subject, issuerDidKey.privateKey) const presentation = await composeFulfillmentFromAttestation(issuer, decodedApplication, attestation, credentialType)

return presentation } We're using the project's issuer DID key here. We validate and decode the application using Verite's validateCredentialApplication function. Then, we have to attest to the credential presentation.

Using the issuer private key and public key, we call the Verite library buildIssuer function. With the result, we can then create the verifiable presentation that will ultimately be passed back to the user by calling Verite's composeFulfillmentFromAttestation function.

It is that presentation that is sent back to the user. We'll take a look at the frontend shortly, but just know that the presentation comes in the form of a JWT.

Verify Mint Access Endpoint

Next, we'll take a look at the verifyMintAccess route. This route includes significantly more functionality. Let's dive in!

JavaScript try { const { jwt } = req.body

if (!req.session || !req.session.siwe) { return res.status(403).send("Unauthorized, please sign in") } const address = req.session.siwe.address

const decoded = await verifyVerifiablePresentation(jwt)

const vc = decoded.verifiableCredential[0]

const decodedVc = await verifyVerifiableCredential(vc.proof.jwt)

const issuerDidKey = await getIssuerKey()

const { subject } = await getOrCreateDidKey(address)

const offer = buildKycVerificationOffer(uuidv4(), issuerDidKey.subject, "https://test.host/verify") const submission = await composePresentationSubmission(subject, offer.body.presentation_definition, decodedVc)

// The verifier will take the submission and verify its authenticity. There is no response // from this function, but if it throws, then the credential is invalid. try { await validateVerificationSubmission(submission, offer.body.presentation_definition) }

catch (error) { console.log(error) return res.status(401).json({ message: "Could not verify credential" }) }

let privateKey = ""

if (!issuerDidKey.signingKey) { throw new Error("No signing key found") } else { privateKey = issuerDidKey.signingKey }

let wallet = new ethers.Wallet(privateKey)

const domain = { name: "AllowList", version: "1.0", chainId: config.chainId, verifyingContract: config.contractAddress } const types = { AllowList: [{ name: "allow", type: "address" }] } const allowList = { allow: address }

const signature = await wallet._signTypedData(domain, types, allowList)

return res.send(signature) } catch (error) { console.log(error) res.status(500).send(error.message) } Once again, the first thing we check is that the user has a valid SIWE session. This route takes a body that includes the verifiable presentation we had sent to the user previously. So, the next step is to call the Verite function verifyVerifiablePresentation to then be able to extract the verifiable credential and call the verifyVerifiableCredential function.

As with our requestAllowlist route, we now need to get the issuer DID key and look up the user's delegated DID key. From there, we can use the issuer key to call the Verite library function buildKycVerificationOffer . We use the results of that call and the user's DID key to call the Verite library function composePresentationSubmission .

Now, we get on to the good stuff. We're going to make sure a valid credential was sent to us. We call the Verite library function validateVerificationSubmission . This function will throw if the credential is invalid. Otherwise, it does nothing. We're rooting for nothing!

Next, the code might get a little confusing, so I want to spend some time walking through this implementation and highlighting how you'd probably do this differently in production. Once the credential is verified, we need to sign a message with a private key owned by the project. For simplicity, I chose to use the same private key that would deploy the smart contract. This is not secure. Don't do this. Hopefully, this is enough to illustrate how to execute the next few steps, though.

We have the issuer DID key written to our database already (file system). We also included a signing key. We need that signing key to sign the message that will be sent back to the user. We use that key to build an Ethereum wallet that can be used for signing.

JavaScript let privateKey = ""

if (!issuerDidKey.signingKey) { throw new Error("No signing key found") } else { privateKey = issuerDidKey.signingKey }

let wallet = new ethers.Wallet(privateKey) Finally, we build out the EIP-712 message and sign it. The resulting signature hash is what we send back to the browser so the user can use it in the smart contract's minting function.

That was a lot, but guess what? The frontend and the smart contract should be a lot quicker and easier to follow.

Frontend

If we back out of the backend folder in our project, we can then switch into the frontend folder. Take a look at frontend/src/index.js . The requestAllowlist function is the one the user will call to hit the project's server's endpoint to see if the user is even allowed to get an allowlist credential. If so, the credential is returned and stored in localstorage:

JavaScript async function requestAllowlistAccess() { try { const res = await fetch(${BACKEND_ADDR}/requestAllowlist, { credentials: "include" }) const message = await res.json()

```
if (res.status === 401) {
  alert(message.message)
  return
}
localStorage.setItem("nft-vc", message)
alert("Credential received and stored in browser")
```

} catch (error) { console.log(error) alert(error.message) } } Again, this would look a lot nicer if there was a built-in credential wallet integration with Ethereum wallets, but for simplicity, the credential is being stored in localstorage. Safe, safe localstorage.

(narrator: localstorage is not safe).

When it's time to mint during the presale, the user clicks on the mint button and the mintPresale function is called:

JavaScript async function mintPresale() { const jwt = localStorage.getItem("nft-vc") if (!jwt) { alert("No early access credential found") return }

const res = await fetch(${BACKEND_ADDR}/verifyMintAccess, { method: "POST", headers: { "Content-Type": "application/json" }, body: JSON.stringify({ jwt }), credentials: "include" })

if (res.status === 401 || res.status === 403) { alert("You're not authorized to mint or not signed in with the right wallet") return }

const sig = await res.text()

const contract = new ethers.Contract(address, json(), signer) const allowList = { allow: address } let overrides = { value: ethers.utils.parseEther((0.06).toString()) } const mint = await contract.mintAllowList(1, allowList, sig, address, overrides) console.log(mint) alert("Minted successfully") } This function grabs the credential from localstorage and sends it along to the project's backend server. Assuming the signature from the project is returned, the user is now able to mint. That signature is sent to the smart contract as well as how many tokens should be minted and the amount of ETH necessary to mint. Note, the allowlist object that we send as well. This helps the smart contract verify the signature. Simple!

But how does that work with the smart contract exactly?

Smart Contract

If you open up the contract folder, you'll see a sub-folder called contracts . In there, you'll see the smart contract we're using in this example, called Base_ERC721.sol .

This is a pretty standard NFT minting contract. It's not a full implementation. There would be project-specific functions and requirements to make it complete, but it highlights the allowlist minting functionality.

The first thing to note is we're using the EIP-712 standard via a contract imported from OpenZeppelin. You can see that with this line:

import "@openzeppelin/contracts/utils/cryptography/draft-EIP712.sol"; Next, we are extending the ERC-721 contract and specifying use of EIP-712 here:

contract BASEERC721 is ERC721Enumerable, Ownable, EIP712("AllowList", "1.0") { ... A little further down in the contract, we create a struct that defines the allowlist data model. It's simple because we are only looking at the wallet address that should be on the allowlist:

struct AllowList { address allow; } We're going to focus in now on the mintAllowList function and the _verifySignature function. Our mintAllowList function starts off similar to a normal NFT minting function except it includes the required signature argument and dataToVerify argument. We do a couple of normal checks before we get to a check that verifies the signature itself. This is where the magic happens.

The _verifySignature function is called. It takes in the data model and the signature.

function _verifySignature(AllowList memory dataToVerify, bytes memory signature) internal view returns (bool) { bytes32 digest = _hashTypedDataV4(keccak256(abi.encode(keccak256("AllowList(address allow)"), dataToVerify.allow)));

require(keccak256(bytes(signature)) != keccak256(bytes(PREVIOUS_SIGNATURE)), "Invalid nonce"); require(msg.sender == dataToVerify.allow, "Not on allow list");

address signerAddress = ECDSA.recover(digest, signature);

require(CONTRACT_OWNER == signerAddress, "Invalid signature");

return true; } Using the EIP-712 contract imported through the OpenZeppelin library, we're able to create a digest representing the data that was originally signed. We can then recover the signing address and compare it to the expected address. In our simplified example, we expect the signer to be the same address as the contract deployer, but you can, of course, extend this much further.

To help avoid replay attacks, we also compare the current signature to a variable called PREVIOUS_SIGNATURE . If the signature is the same, we reject the entire call because a signature can only be used once.

Back to our mintAllowList function, if the signature is verified, we allow the minting to happen. When that's complete, we update the PREVIOUS_SIGNATURE variable. This is, as with many things in this demo, a simplified replay attack prevention model. This can and probably should be extended to support your own use cases.

Caveats and Conclusion

In a perfect world, we would not be issuing credentials to a delegated subject DID. In our example, we could have just as easily have issued to the user's wallet address, but we wanted to highlight the DID functionality as best as possible.

It is possible today for the user to manage their own DID and keys, but the tricky part comes, as mentioned earlier in this post, when interacting with crypto wallets. Signing a transaction is not the same as signing a JWT. The keys used are different, the signatures are different, and the flow is different. Until these things become unified and more seamless, this demo helps illustrate how Verite can be used today to enforce allowlist restrictions for an NFT minting project.

Hopefully, this sparks some creativity. Hopefully, it inspires some people to go and build even more creative solutions that leverage verifiable credentials and Verite. Updated5 months ago * Table of Contents * * What is an Allowlist? * * How Are Allowlists Normally Managed? * * How To Use Verite and Verifiable Credentials * * Request Allowlist Endpoint * * Verify Mint Access Endpoint * * Frontend * * Smart Contract * * Caveats and Conclusion