title: "EIP-1271: Signing and Verifying Smart Contract Signatures" description: An overview of smart contract signature generation and verification with EIP-1271. We also walk through the EIP-1271 implementation used in Safe (previously Gnosis Safe) to provide a concrete example for smart contract developers to build on. author: Nathan H. Leung lang: en tags: ["eip-1271", "smart contracts", "verifying", "signing"] skill: intermediate published: 2023-01-12

The EIP-1271 standard allows smart contracts to verify signatures.

In this tutorial, we give an overview of digital signatures, EIP-1271's background, and the specific implementation of EIP-1271 used by Safe (previously Gnosis Safe). All together, this can serve as a starting point for implementing EIP-1271 in your own contracts.

## What is a signature?

In this context, a signature (more precisely, a "digital signature") is a message plus some sort of proof that the message came from a specific person/sender/address.

For instance, a digital signature might look like this:

1. Message: "I want to log in to this website with my Ethereum wallet."
2. Signer: My address is `0x000…`
3. Proof: Here is some proof that I, `0x000…`, actually created this entire message (this is usually something cryptographic).

It's important to note that a digital signature includes both a "message" and a "signature".

Why? For instance, if you gave me a contract to sign, and then I cut off the signature page and gave you back only my signatures without the rest of the contract, the contract would not be valid.

In the same way, a digital signature doesn't mean anything without an associated message!

## Why does EIP-1271 exist?

In order to create a digital signature for use on Ethereum-based blockchains, you generally need a secret private key which no one else knows. This is what makes your signature, yours (no one else can create the same signature without knowledge of the secret key).

Your Ethereum account (i.e. your externally-owned account/EOA) has a private key associated with it, and this is the private key that's typically used when a website or dapp asks you for a signature (e.g. for "Log in with Ethereum").

An app can verify a signature you create using a third-party library like ethers.jswithout knowing your private key and be confident that *you* were the one that created the signature.

> In fact, because EOA digital signatures use public-key cryptography, they can be generated and verified**off-chain**! This is how gasless DAO voting works — instead of submitting votes on-chain, digital signatures can be created and verified off-chain using cryptographic libraries.

While EOA accounts have a private key, smart contract accounts do not have any sort of private or secret key (so "Log in with Ethereum", etc. cannot natively work with smart contract accounts).

The problem EIP-1271 aims to solve: how can we tell that a smart contract signature is valid if the smart contract has no "secret" it can incorporate into the signature?

## How does EIP-1271 work?

Smart contracts don't have private keys that can be used to sign messages. So how can we tell if a signature is authentic?

Well, one idea is that we can just *ask* the smart contract if a signature is authentic!

What EIP-1271 does is it standardizes this idea "asking" a smart contract if a given signature is valid.

A contract that implements EIP-1271 must have a function called `isValidSignature` which takes in a message and a signature. The contract can then run some validation logic (the spec does not enforce anything specific here) and then return a value indicating whether the signature is valid or not.

If `isValidSignature` returns a valid result, that's pretty much the contract saying "yes, I approve this signature + message!"

### Interface

Here's the exact interface in the EIP-1271 spec (we'll talk about the `_hash` parameter below, but for now, think of it as the message that is being verified):

```jsx pragma solidity ^0.5.0;

contract ERC1271 {

// bytes4(keccak256("isValidSignature(bytes32,bytes)") bytes4 constant internal MAGICVALUE = 0x1626ba7e;

/* ** @dev Should return whether the signature provided is valid for the provided hash * @param _hash Hash of the data to be signed * @param _signature Signature byte array associated with _hash * * MUST return the bytes4 magic value 0x1626ba7e when function passes. * MUST NOT modify state (using STATICCALL for solc < 0.5, view modifier for solc > 0.5) * MUST allow external calls* / function isValidSignature( bytes32 _hash, bytes memory _signature) public view returns (bytes4 magicValue); } ```

## Example EIP-1271 Implementation: Safe

Contracts can implement `isValidSignature` in many ways — the spec only doesn't say much about the exact implementation.

One notable contract which implements EIP-1271 is Safe (previously Gnosis Safe).

In Safe's code, `isValidSignature` is implemented so that signatures can be created and verified in two ways:

1. On-chain messages
2. Creation: a safe owner creates a new safe transaction to "sign" a message, passing the message as data into the transaction. Once enough owners sign the transaction to reach the multisig threshold, the transaction is broadcast and run. In the transaction, there is a safe function called which adds the message to a list of "approved" messages.
3. Verification: call `isValidSignature` on the Safe contract, and pass in the message to verify as the message parameter and an empty value for the signature parameter (i.e. `0x`). The Safe will see that the signature parameter is empty and instead of cryptographically verifying the signature, it will know to just go ahead and check whether the message is on the list of "approved" messages.
4. Off-chain messages:
5. Creation: a safe owner creates a message off-chain, then gets other safe owners to sign the message each individually until there are enough signatures to overcome the multisig approval threshold.
6. Verification: call `isValidSignature`. In the message parameter, pass in the message to be verified. In the signature parameter, pass in each safe owner's individual signatures all concatenated together, back-to-back. The Safe will check that there are enough signatures to meet the threshold **and** that each signature is valid. If so, it will return a value indicating successful signature verification.

## What exactly is the `_hash` parameter? Why not pass the whole message?

You might have noticed that the `isValidSignature` function in the EIP-1271 interface doesn't take in the message itself, but instead a `_hash` parameter. What this means is that instead of passing the full arbitrary-length message to `isValidSignature`, we instead pass a 32-byte hash of the message (generally keccak256).

Each byte of calldata — i.e. function parameter data passed to a smart contract function — costs 16 gas (4 gas if zero byte), so this can save a lot of gas if a message is long.
```

### Previous EIP-1271 Specifications

There are EIP-1271 specifications in the wild that have an `isValidSignature` function with a first parameter of type `bytes` (arbitrary-length, instead of a fixed-length `bytes32`) and parameter name `message`. This is an older version of the EIP-1271 standard.

## How should EIP-1271 be implemented in my own contracts?

The spec is very open-ended here. The Safe implementation has some good ideas:

- You can consider EOA signatures from the "owner" of the contract to be valid.
- You could store a list of approved messages and only consider those to be valid.

In the end, it is up to you as the contract developer!

## Conclusion

EIP-1271 is a versatile standard that allows smart contracts to verify signatures. It opens the door for smart contracts to act more like EOAs — for instance providing a way for "Log in with Ethereum" to work with smart contracts — and it can be implemented in many ways (Safe having a nontrivial, interesting implementation to consider).