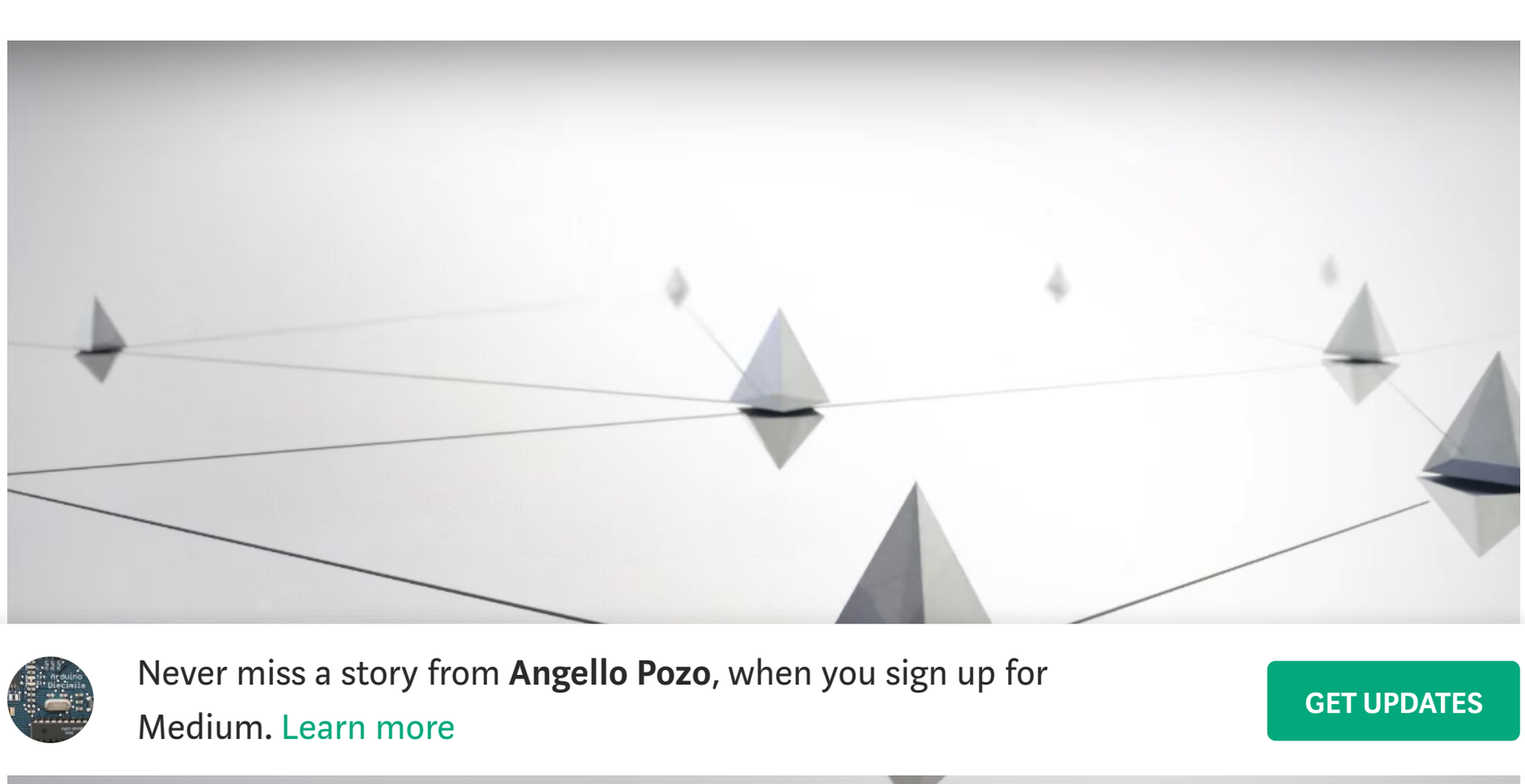
Angello Pozo Follow
Co-Founder of HelloSugoi. Hacking away on Ethereum (blockchain) DApps. Follow me on https://twitter.com/angellopozo
Jun 9, 2017 · 4 min read

Ethereum: Signing and Validating



A core primitive of Ethereum and other cryptocurrencies is the ability to sign data that can be verified by anyone. This powers the distributed nature of blockchain. In Bitcoin you sign a transaction saying you want to give Sally 4 bitcoin. Without this property, anyone could make fake transactions giving themselves all coins.

. . .

TL;DR;

If you go to ecrecover-example on github for the full codebase. Simply follow the instructions in the README.md and see the results in the command line.

What is Signing?

Signing is the act of a user A "signing" data that anyone can validate came from user A. This is used in transactions to check if they are real.

A common question is "how can you validate transactions are real?" The short answer is public-key cryptography. It's an algorithm with 3 parts.

1. Key Creation
2. Encryption/Signing
3. Decryption/Validation

Encryption is generally used to hide data in other data. If you encrypt a string like 'hello world' you get something like `dqE3gJz/+5CQHfSJwMP2nQ`. Its purpose is to hide the message 'hello world'. Signing is used to create a different output string, but you also publicize the original message.

The key creation will output two strings, a public and private key. It links them through an algorithm that has the signing and validation properties. A signature will take in a public key, private key, and message. The output will be another string that is the signature.

1. Signature = F(public key, private key, message)
2. Validation = F(Signature, message)
3. Is Valid if: Validation = public key

Notice how validation does not require knowledge of the private key. This is what allows 3rd parties to validate information. If the output of the validation function is equal to the public key then the signature is real, otherwise its fake.

The signature is made up of 3 variables: v, r, s. Ethereum employs Elliptic curve cryptography and those variables are simply part of the underlying math.

Why Sign?

Signing is a nice way to know something is being done by the correct person/contract. This means we can trust that someone is actually doing what they say they are.

Instead of real world signatures, which can be faked, the digital ones can not. If you want to know user A did something, make them sign it before moving forward. Then if a dispute arrises, check the signature.

Development:

As a developer you want your users to sign a message. There are 3 parts to creating this feature in your respective DApp (Distributed Application).

1. Solidity validator function
2. Client code to sign a message
3. Client code to call Solidity validator

Solidity Validator:

Solidity provides a globally available method `ecrecover` that returns an address. If the return address is the same as the signer, then the signature is real.

```
pragma solidity ^0.4.8;
contract Verifier {
    function recoverAddr(bytes32 msgHash, uint8 v, bytes32 r, bytes32 s) returns (address)
        return ecrecover(msgHash, v, r, s);
    }

    function isSigned(address _addr, bytes32 msgHash, uint8 v, bytes32 r, bytes32 s) retur
        return ecrecover(msgHash, v, r, s) == _addr;
    }
}
```

Verifier.sol hosted with ❤ by GitHub view raw

Solidity Verifier.sol

The code above creates a `Verifier` contract with the `recoverAddr` and `isSigned` functions. The latter will return an address. Requiring you as a developer to validate, outside of Solidity, that the address is correct. The second method, `isSigned` does the check within Solidity. `isSigned` will return true or false if the `msgHash` is signed by `_addr`.

Creating Signature:

There are two ways to create a signature:

1. Using Web3's Javascript function `web3.eth.sign`
2. Calling the RPC API of an Ethereum node

If you are using Javascript then all you have to do is require `web3` and attach to an Ethereum node. In the code below, I am running a private Ethereum node bound to `localhost:8545`. NOTE: THIS WILL NOT WORK ON TESTRPC

```
const Web3 = require('web3')
const provider = new Web3.providers.HttpProvider('http://localhost:8545')
const web3 = new Web3(provider)

function toHex(str) {
 var hex = ''
 for(var i=0;i<str.length;i++) {
  hex += ''+str.charCodeAt(i).toString(16)
 }
 return hex
}

let addr = web3.eth.accounts[0]
let msg = 'I really did make this message'
let signature = web3.eth.sign(addr, '0x' + toHex(msg))
console.log(signature)
```

genSignature.js hosted with ❤ by GitHub view raw

Generate an Ethereum signature

There is no built in function to convert a string to hex. So I used the function `toHex` to do the conversion. The users address (`web3.eth.accounts[0]`) and message with the `0x` prefixed are passed into the `web3.eth.sign` function.

Another way to create a signature is to call the Ethereum RPC API. With `curl` you should be able to make a request to an Ethereum node.

```
data '{

d59d4723bc913230a155d0d", "0x0564b25c8fcd6766f672d43252c8ee2597ad6c7a35315cf13e3b4d00bafc2e9f"],
```

eth_sign_RPC.sh hosted with ❤ by GitHub view raw

Call Ethereum RPC api eth_sign

The first parameter in `params` is the users address, and the second is the hex value of the message. Note for the RPC api to work your account must be unlocked. You will get something like the following back:

```
0x9955af11969a2d2a7f860cb00e6a00cfa7c581f5df2dbe8ea16700b33f4b4b9b69
f945012f7ea7d3febf11eb1b78e1adc2d1c14c2cf48b25000938cc1860c83e01
```

The long signature encodes the previously mentioned v, r, s variables. To extract these values, you need to parse the signature into substrings.

```
signature = signature.substr(2); //remove 0x
const r = '0x' + signature.slice(0, 64)
const s = '0x' + signature.slice(64, 128)
const v = '0x' + signature.slice(128, 130)
const v_decimal = web3.toDecimal(v)
```

extract_signature_params_snippet.js hosted with ❤ by GitHub view raw

Parsing out r, s, and v from the signature

NOTE: v must be a decimal number, hence the second `v_decimal` that turns hex v into decimal v.

DANGER: The resulting `v_decimal` must be either 27 or 28!

Checking if Correct:

With the validator and signing completed, all that is left is to actually check if the signature is real. There is one teeny tiny caveat. Remember when creating the signature we used the string `0x + toHex(msg)`. Well that is not the same hash that you pass into the validator!

Again, the hash to creating the signature is not the same for the validator. The reason is to protect the user from signing arbitrary payloads.

The solution is to add a custom Ethereum message, and length.

```
//FOR SIGNATURE Hex:
I really did make this message

//FOR VALIDATOR sha3:
\x19Ethereum Signed Message:\n30I really did make this message
```

This distinction is VERY necessary! Do not waste your time following any other steps.

The last step is to somehow call your Solidity code. I am using Truffle 3 to deploy the previous smart contract. Please note the location of the Ethereum node must be declared (`localhost:8545`) for the contract. Otherwise it will not work as expected.

```
//...
const SignVerifyArtifact = require('./contracts/SignAndVerifyExample')
const SignVerify = contract(SignVerifyArtifact)
SignVerify.setProvider(provider)
//...
SignVerify
  .deployed()
  .then(instance => {
    let fixed_msg = `\x19Ethereum Signed Message:\n${msg.length}${msg}`
    let fixed_msg_sha = web3.sha3(fixed_msg)
    return instance.verify.call(
      fixed_msg_sha,
      v_decimal,
      r,
      s
    )
  })
  .then(data => {
    console.log('-----data------')
    console.log(`input addr ==> ${addr}`)
    console.log(`output addr => ${data}`)
  })
  .catch(e => {
    console.error(e)
  })
```

verify_signature.js hosted with ❤ by GitHub view raw

Validate signature is valid

Truffle creates a deployed function that returns the contract instance. I create my sha3 message and pass the required variables into `instance.verify.call`. If the addresses returned by the last two lines are the same then the owner really did sign the message. Otherwise its a forgery that can be ignored.

Conclusion:

Signing data can be important for any kinds of DApp. Some obvious applications are rights management, copyright, patent ownership. Users could sign those files and anyone can validate that they did in fact make those things. What use case can you think of?

Blockchain Ethereum Solidity Smart Contracts Truffle

Angello Pozo Follow
Co-Founder of HelloSugoi. Hacking away on Ethereum (blockchain) DApps. Follow me on https://twitter.com/angellopozo

Responses

Write a response...

Conversation with Angello Pozo.

Taha Dhailey
Feb 13

Hi,
can you write a full code altogether and explain the files distribution

its very difficult to understand like this

1 response

Angello Pozo
Feb 17

Hey Taha, The entire code base is in the following repo: https://github.com/sogoiii/ecrecover-example

2

Conversation with Angello Pozo.

Taha Dhailey
Feb 13

i am getting this error while compiling genSignature
var length = string.length;

TypeError: Cannot read property 'length' of undefined

3 responses

Angello Pozo
Feb 17

Hello, I am uncertain what code or line you are looking at. Can you clarity please.

Show all responses