Saturday, 9 December 2023

Ethereum Learning

Signature[1]

Basics

- Signature(a number) = an input message + a private key + a random secret
- Signature algorithms
  - RSA
  - AES
  - Ethereum and Bitcoin use Elliptic Curve Digital Signature Algorithm, or ECDSA
  - RSA and AES can also be used to encryption, but ECDSA is only a signature algorithm
- Trapdoor functions

Signing and Verifying Using ECDSA

- ECDSA signatures consist of two numbers (integers): `r` and `s`. Ethereum also uses an additional `v` (recovery identifier) variable. The signature can be notated as `{r, s, v}`.
- The process of signing
  - Calculate a hash (`e`) from the message to sign
  - Generate a secure random value for `k`.
  - Calculate point (x_1, y_1) on the elliptic curve by multiplying k with the G constant of the elliptic curve.
  - Calculate $r = x_1 \bmod n$. If r equals zero, go back to step 2.
  - Calculate $s = k^{-1}(e + rd_a) \bmod n$. If s equals zero, go back to step 2.

1. Message signing and verifying

- For **message signing** in Ethereum , the hash is usually calculated with `Keccak256("\x19Ethereum Signed Message:\n32" + Keccak256(message))`. This ensures that the signature cannot be used for purposes outside of Ethereum. The fixed `0x19` byte prefix was chosen, so that the signed message cannot be an RLP encoded signed transaction, since RLP encoded transactions never start with `0x19`.
- The `{r, s, v}` signature can be combined into one 65-byte-long sequence: 32 bytes for `r`, 32 bytes for `s`, and one byte for `v`. If we encode that as a hexadecimal string, we end up with a 130-character-long string
- A full message signature may look like

Signature


```
{
 "address": "0x76e01859d6cf4a8637350bdb81e3cef71e29b7c2",
 "msg": "Hello world!",
 "sig":
"0x21fbf0696d5e0aa2ef41a2b4ffb623bcaf070461d61cf7251c74161f82fec3a4370
854bc0a34b3ab487c1bc021cd318c734c51ae29374f2beb0e6f2dd49b4bf41c",
 "version": "2"
}
```


- In order to verify a message, we need the original message, the address of the private key it was signed with, and the signature `{r, s, v}` itself.
- The process of **message** verifying
  - Calculate the hash (`e`) for the message to recover
  - Calculate point `R = (x1, y1)` on the elliptic curve, where x_1 is `r` for `v = 27`, or `r + n` for `v = 28`
  - Calculate $u_1 = -zr^{-1} \bmod n$ and $u_2 = sr^{-1} \bmod n$
  - Calculate point $Q_a = (x_a, y_a) = u_1 \times G + u_2 \times R$. `Qa` is the point of the *public* key for the *private* key that the address was signed with. We can derive an address from this and check if that matches with the provided address. If it does the signature is valid.

- `v` is the last byte of the signature, and is either 27 (`0x1b`) or 28 (`0x1c`). Two addresses can be derived from r and s(two points on the curve). So we use v to indicate which point to use. In most implementations, the `v` is just 0 or 1 internally, but 27 was added as arbitrary number for signing Bitcoin messages and Ethereum adapted that as well.
- Since EIP-155, we also use the chain ID to calculate the `v` value. This prevents replay attacks across different chains. Currently, this is only used for signing transaction however, and is not used for signing messages.

2. Standardisation of signed messages

- "\x19Ethereum Signed Message:\n32" + Keccak256(message)" is called personal_sign format. Cause only from address is specified, but no to address, one can use the signature to perform reply attack. For example, if user `A` signs a message and sends it to contract `X`, user `B` can copy that signed message and send it to contract `Y`
- EIP-191 and EIP-712 are some of the proposals that aim to solve this problem.

2.1 EIP-191: Signed Data Standard

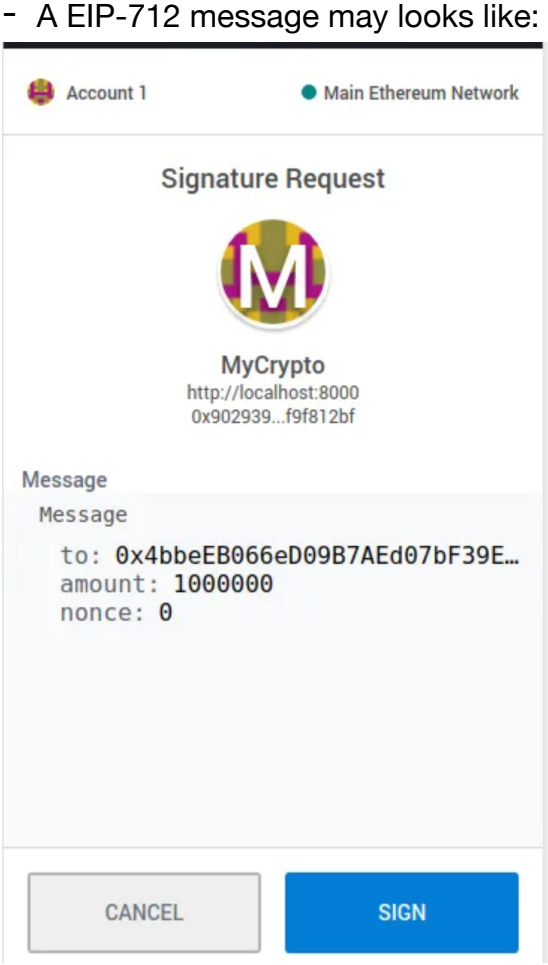
- EIP-191 is a very simple proposal: It defines a version number and version specific data. The format looks like this:

```
0x19 <1 byte version> <version specific data> <data to sign>
```

- The version-specific data depends (as the name suggests) on the version we use. Currently, EIP-191 has three versions:
  - `0x00`: Data with "intended validator." In the case of a contract, this can be the address of the contract. If we specify an intended validator (e.g., a contract address), the contract can re-calculate the hash with its own address. Submitting the signed message to a different instance of a contract won't work, since it won't be able to verify the signature.
  - `0x01`: Structured data, as defined in EIP-712
  - `0x45`: Regular signed messages, like the current behaviour of `personal_sign`

2.2 EIP-712: Ethereum typed structured data hashing and signing

- EIP-712 is a proposal for “typed” signed data. This makes signing data more verifiable, by presenting it in a human-readable way.
- A EIP-712 message may looks like:




```
{
  types: {
    EIP712Domain: [
      { name: 'name', type: 'string' },
      { name: 'version', type: 'string' },
      { name: 'chainId', type: 'uint256' },
      { name: 'verifyingContract', type: 'address' },
      { name: 'salt', type: 'bytes32' }
    ],
    Transaction: [
      { name: 'to', type: 'address' },
      { name: 'amount', type: 'uint256' },
      { name: 'nonce', type: 'uint256' }
    ]
  },
  domain: {
    name: 'MyCrypto',
    version: '1.0.0',
    chainId: 1,
    verifyingContract: '0x098D8b363933D742476DDd594c4A5a5F1a62326a',
    salt: '0x76e22a8ee58573472b9d7b73c41ee29160bc2759195434c1bc1201ae4769afd7'
  },
  primaryType: 'Transaction',
  message: {
    to: '0x4bbeEB066eD09B7AEd07bF39EEe0460DFa261520',
    amount: 1000000,
    nonce: 0
  }
}
```

- For this method, we have to specify all the properties (e.g., `to`, `amount`, and `nonce`) and their respective types (e.g., `address`, `uint256`, and `uint256`), as well as some basic information about the application, called the *domain*.
- The domain contains information like the name of the application, the version, chain ID, the contract you’re interacting with, and a salt. The contract should verify this information, to make sure that a signature for one application cannot be used for another. This solves the problem of a potential replay attack described earlier.

3. Transaction signing and verifying

- Raw transaction may looks like

Raw Transaction

```
{
  "value": "0xde0b6b3a7640000",
  "data": "0x",
  "to": "0x4bbeeb066ed09b7aed07bf39eee0460dfa261520",
  "nonce": "0xa",
  "gasPrice": "0x2540be400",
  "gasLimit": "0x5208",
  "chainId": 0
}
```

- Signed transactions are RLP encoded, and consist of all transaction parameters (nonce, gas price, gas limit, to, value, data) and the signature (v, r, s). A signed transaction looks like this:

```
0xf86c0a8502540be400825208944bbeeb066ed09b7aed07bf39eee0460dfa2615208
80de0b6b3a7640000801ca0f3ae52c1ef3300f44df0bcfd1341c232ed6134672b16e3
5699ae3f5fe2493379a023d23d2955a239dd6f61c4e8b2678d174356ff424eac53da5
3e17706c43ef871
```

- The process of **transaction** signing
  - Encode the transaction parameters: RLP(nonce, gasPrice, gasLimit, to, value, data, chainId, 0, 0).
  - Get the Keccak256 hash of the RLP-encoded, unsigned transaction.
  - Sign the hash with a private key using the ECDSA algorithm, according to the steps described above.
  - Encode the signed transaction: RLP(nonce, gasPrice, gasLimit, to, value, data, v, r, s).
- Note that the chain ID is encoded in the `v` parameter of the signature, so we don't include the chain ID itself in the final signed transaction. We also don't specify any "From" address, as this can be recovered from the signature itself.

4. Verifying signatures with smart contracts

- What makes message signatures more interesting is that we can use smart contracts to verify the ECDSA signatures. Solidity has a built-in function called `ecrecover` (which is actually a precompiled contract at address 0x1) that will *recover* the address of the private key that a message was signed with.

- What makes something like this useful is that a user has a trustless way to give a smart contract certain commands without sending a transaction. The user could, for example, sign a message saying, “Please send 1 Ether from my address to this address.” A smart contract can then verify who signed that message, and execute that command, using a standard like EIP-712, and/or EIP-1077.

Conferences

1. https://medium.com/mycrypto/the-magic-of-digital-signatures-on-ethereum-98fe184dc9c7