Authentication and Message Signing

Authentication

User Identity

Definition

The **Primary Identity** of a user is a blockchain address, typically an Ethereum address. The notation follows the "chain-specific addresses" format as implemented by Gnosis Safe.

```
eth:0x4714C7EfE5D0213615FC6CBB8717B524eC433e9a
```

Transferability

- The user's primary identity must not be transferable to another user.
- Most of the time, the primary identity will be an Ethereum Externally Owned Account (EOA). In this case, the primary identity is controlled by an Ethereum private key, ideally stored on a hardware wallet in optimal secure conditions.
    - An EOA's private key can be compromised but it cannot be transferred to a new owner. This is because it's impossible to prove to the new owner that the original owner didn't store a copy (i.e. fully erased their knowledge) of the key. There is no guaranteed way to shut out the original owner.
    - When the private key is compromised, users are supposed to burn their primary identity and start completely from scratch with a new key.
- Another possibility we may want to support, is that the primary identity is a smart contract, which would typically be a wallet contract such as Gnosis Safe or Argent wallet.
    - The concern here, is that these contracts can be transferred between owners, often by design.
    - Account abstraction is a hot topic currently in the Ethereum community and it may become established as the standard UX pattern in the future to indicate primary identities of users.
    - Keeping these two points in mind, we should try and avoid the usage smart contracts as long as possible, but remain compatible so that we keep the door open for this future possibility.

Pseudonymity and Uniqueness

- Users may choose to remain pseudonymous, or they may choose to share their real-world identity with others and even prove the ownership of their primary identity (by publicly signing a message.)
- Users can own multiple pseudonymous accounts. It's not possible to prevent this and we won't try to.

Aliases

- The user may have one or more aliases associated with their identity. These aliases should never be confused with the primary identity itself. Examples are:
    - An ENS domain name that the user set up to share their identity more easily with other.

    - A blockchain address on a sidechain, roll-up, or compatible L1 blockchain that shares the same private key with the primary identity.

- Contracts and applications that keep a reference to the user identity must always use the primary identity, i.e. the fully resolved blockchain address, in order to avoid issues when an alias changes and no longer refers to the right address.

Primary Authentication Method

Since the user's primary identity is a blockchain address, there is an obvious way for the user to authenticate themselves by signing a message.

- If the user interaction involves a transaction with the blockchain that houses their primary identity, the transaction will automatically be signed with their key and thus authenticate them.
- If the user interaction does not involve a blockchain transaction, the application that wishes to authenticate the user may ask them to sign a message with their key. This is a commonly used approach by dapps such https://dework.xyz/, https://wonderverse.xyz/, and countless others.

Burn Message

If a user fears that their Primary Identity is compromised, they can explicitly burn the Ethereum address as a signal that it must no longer be trusted and that any future activity is potentially done by an impostor.

Since this is only a signaling flag, it suffices to agree which method will be used for it and then to call this method in a on-chain transaction (properly signed by the primary identity's key):

```
ID_CONTRACT.BURN(<primary identity>)
```

This action would revoke any Delegated Identities (see further) and block any future actions for the address. It must not be possible to undo a burn, because this undo function would also be available to the impostor.

Delegated Identities

The user is not expected to use their Primary Identity very often to authenticate themselves. This will be required in certain cases, but only if the security requirements necessitate it.

Most of the time, the user will instead use **Delegated Identities** to authenticate themselves. Typically, a delegated identity is tied to an application instance or an application session that needs to act (sign messages) on the user's behalf.

Delegated Identities have weaker security guarantees when it comes to authentication, but there are a lot of restrictions on what they can do, and they offer a drastically improved user experience compared to using a Primary Identity. For each use case, a design decision about the appropriate authentication method will need to be made based on the requirements.

Creating and Revoking Delegate Identities

Using their Primary Identity, the user can create as many Delegate Identities as they want. These Delegated Identities can also be revoked again, either manually, or automatically when their expiration timestamp passes.

- When the user creates a **Delegate Identity**, the app that wants to act on the user's behalf, will first create a private/public key pair. This can for example be a Powerhouse Connect desktop app that the user just installed, or a Powerhouse Fusion application that the user is logging in to.
- The app will assemble an `approval message` and calculate the `approval hash` from that message, as expressed in the pseudo-code below:

```
(privKey, pubKey) = GENERATE_KEY_PAIR();

<approval message> = {
    signer: "powerhouse/connect",
    publicKey: <pubKey>
};

<approval hash> = HASH (<approval message>);
```

- The Delegate Identity is then **approved** by the user by signing a transaction with their Primary Identity (this typically happens on Renown), and by submitting this transaction to the blockchain:

```
ID_CONTRACT.APPROVE_DELEGATE_ID (<approval hash>, <expiry>)
```

- The Delegate Identity can be **revoked** before the expiration time in the same way:

```
ID_CONTRACT.REVOKE_DELEGATE_ID (<approval hash>)
```

- An external observer *can* see that the user approved *a Delegate Identity* with their Primary Identity and that the approval is still valid (= not revoked and not expired.) The external observer *cannot* see who it is that was approved.
    - If someone were to present the `approval message` to the external observer, however, then the observer would indeed be able to verify that this is indeed the approval that was given.

Secondary Authentication Method

Once the approval is signed with the user's Primary Identity and included in an on-chain block, the Delegate Identity can now authenticate themselves as the user:

- The Delegate Identity signs the approval message with their generated private key, creating an `authentication message` that will be shared with anyone trying to authenticate them.

```
<approval message> = {
    signer: "powerhouse/connect",
    publicKey: <pubKey>
}

<signature> = SIGN(<approval message>, <privKey>)

<authentication message> = {
    id: <primary identity>,
    signer: "powerhouse/connect",
    publicKey: <pubKey>,
    signature: <signature>
}
```

- The application that is trying to authenticate the user receives the authentication message and performs the following actions:

```
if (!SIGNATURE_CORRECT(<authentication message>)) {
    // Delegate identity failed to properly sign their authentication message
}

<approval message> = {
    signer: <authentication message>.signer,
    publicKey: <authentication message>.publicKey
}

<approval hash> = HASH (<approval message>);
```

```
if (ONCHAIN_APPROVAL_IS_VALID(<primary identity>, <approval hash>)) {
    // Delegate identity authenticated as user
} else {
    // Delegate identity NOT authenticated as user
}
```

- The application holding the Delegate Identity can now sign messages on the user's behalf. If the application's private key is compromised, the user should revoke the approval with their Primary Identity.

Note that the pseudo-code is only meant to illustrate the principles of the authentication process. The actual implementation should improve on the algorithm design by using industry standards as much as possible, and use things like challenges to prevent replay attacks and other exploits.

Message Signing and Validation

Problem Description

Using the primary and secondary authentication methods described in the preceding section, it's quite simple for users to authenticate themselves and sign messages in the present, real-time environment. They simply do this by signing the message with the private key of a primary identity that has not been burned, or a delegate identity that has not been revoked or expired. Recipients can easily verify that the message signature is valid at present by looking at the on-chain record.

However, if, at some time in the future, an external observer wants to validate that a given message was properly signed by an identity, especially that its signature was *valid at the time of signing*, they cannot do it if the identity is no longer valid.

For example:

- User Alice signs a message M with a primary or delegated identity private key. The signature is valid at the time of signing, but Alice does not share the message with anyone yet.
- Some time passes and the private key becomes invalid because (1) Alice burnt the primary identity or (2) the secondary identity key that was used has been revoked, or (3) the secondary identity key that was used, automatically expired.
- When Alice now shares the message M with Bob (for the first time), she can no longer prove that the message signature was valid at the time of signing. The only thing that Bob can verify, is that the message is signed with a key that was valid at some time in the past but is no longer valid at present.

The problem is simple: in order for Bob to verify that the message was valid at the time of signing, Bob needs a way to determine *when* the message was signed. If that can be determined with certainty, it again becomes possible to verify on-chain whether the identity was already burned, revoked, or expired at the time of signing.

Adding Timestamps to Messages

Timestamp Attestations

In order to accommodate future verification of the validity of message signatures, we will add timestamp attestations to the message.

The principle is simple: the sender or someone else presents the message to a trusted party that will attest that they saw the message at the present time. At any time in the future, it can then be verified that the sender identity was indeed valid at that time when the trusted party saw the message.

Message Example

```
{
    sender: {
        id: <primary identity>,
        signer: "powerhouse/connect",
        publicKey: <pubKey>
    },

    payload: <message content>,

    signature: "...",    // Signature of the payload by sender, verifiable as long
                         // as the signer public key is not burned/revoked/expired.

    attestations: [      // Attested timestamps by various authorities
        {
            timestamp: 1678456339,
            id: <signer 1>,
            signature: "..." // Signature of the payload + timestamp by a trusted signer
        },
        {
            timestamp: 1678459920,
            id: <signer 2>,
            signature: "..." // Signature of the payload + timestamp by a trusted signer
        }
    ]
}
```

Attested timestamps can be added at various times after a message was signed to increase the trust level, but it is expected that in practice the number of attestations will remain low (<5). Typically, only one or two attestations will be present.

Attestation Confidence Level

We cover 3 examples of attestations with varying confidence levels and costs that could be considered for different use cases:

- **Recipient Attestation**: as the cheapest form of attestation, and the lowest confidence level, a message can simply be timestamped by its immediate recipient:
    - Alice signs a new message and sends it to Bob.
    - Bob verifies that the message is valid and adds his signed timestamp to the attestations list.
    - Sometime later when Alice's identity is no longer valid, the message is shared with Charlie.
    - Charlie can no longer directly verify that the message was valid at the time when Alice signed it, but Charlie can see that Bob validated the message at the time.
    - Charlie can choose to trust Bob's honesty, or not.
- **Timestamp Service Attestation**: to improve the confidence level, consider a neutral timestamp service that is trusted by Alice and Bob. This can even be Twitter or another social media service.
    - Alice signs a new message and sends it to the timestamp service.
        - In case of Twitter, she just posts a tweet with the hash of the signed message.
    - The timestamp service accepts the message and it signs a timestamp attestation.
        - In case of Twitter, a link to the tweet with the signed message hash can be included as "attestation signature."
    - Sometime later when Alice's identity is no longer valid, the message is shared with Bob.
    - Bob can no longer directly verify that the message was valid at the time when Alice signed it, but Bob can see that the timestamp service validated the message at the time.

    - In the case of Twitter, Bob can see that a Tweet with the hash was posted at a time when the identity was still valid.
  - Bob can assume that the timestamp service (or Twitter) wasn't compromised in order to falsify the timestamp attestation, or not.
- **L2/L1 Attestation**: to further improve on the confidence level, consider the more expensive options of posting message hashes on an L2 or L1 blockchain instead of on Twitter.
  - Alice signs a new message, posts the hash to the blockchain, and pays to include it in the next block.
  - Sometime later when Alice's identity is no longer valid, the message is shared with Bob.
  - Bob can no longer directly verify that the message was valid at the time when Alice signed it, but Bob can see that the message hash was registered on-chain at a time when it was still valid.
  - Bob can assume that the blockchain consensus mechanism was not hacked to falsify the registered hash or its time of inclusion, or not.
