# DAuth - A distributed authentication mechanism for blockchains

Madhavan Malolan

May 22, 2018

**Abstract**

Since the popularity of Ethereum and other general purpose blockchains, distributed apps (DApps) have started to gain traction. DApps are usually tied to smart contracts on the underlying chain and have a front end like a website to interact with the user. To give a personalized experience the website must have a way to identify the user. This paper presents a mechanism to identify users - by determining the address they own on the block chain.

This paper tries to mitigate two major problems on user authentication and identification - human readable identifiers like usernames and simple password based logins. All of this is done in a way such that the system may be built in a completely decentralized manner, by keeping the user experience consistent with the flows users are already familiar with - *Login with Facebook*, *Sign in with Google* and the like.

## 1 Introduction

DApps have become increasingly popular because of the potential they hold, but the developer tools available to make that a reality are far from sufficient. Authentication and identification are primary requirements for any personalized user experience. Let us consider the example of one of the most popular DApps at the time of this writing - CryptoKitties. CryptoKitties is a online collectibles game based on Ethereum's ERC-721 [2] non fungible token specification. A user may buy, sell or breed a set of *cryptokittes* she owns. This is a web based application. To access the kitties you own, you must install a Web3 provider like MetaMask on the web browser you are using. You cannot login into your account that has MetaMask of some other user (or address). This makes the *kitties* inaccessible from other devices.

However, this is a major step down from other web apps like, say, Gmail which is an email service. One can easily *login* into their account from any machine as long as they remember their Gmail username and password. Many other web apps use a centralized login mechanism owned by larger players, *e.g.* Sign in with Google, Login with Facebook, Sign in with Twitter. These systems use a protocol called OAuth [1] and require a central authority like Google, Facebook and Twitter to mediate the authentication.

In this paper we propose a new solution, *DAuth*, that provides the same user experience and at the same time doing away with the requirement of a central authority.

## 2 DAuth

### 2.1 Design goals

DAuth is designed to work on top of a block chain that allows for the required primitives to enforce the rules on a blockchain. However, this paper focuses on an implementation on Ethereum.

#### 2.1.1 Human readable

An account on the Ethereum blockchain is identified by an address by a long string of 42 characters that is hard to memorize and share [3]. An address looks like `0x2DB6Ff6eB673138E2dcc96EA7B9037552F788aF4`. DAuth aims to present a way for people to tie their address with a human readable username that is unique on the blockchain. This registration is completely trustless and doesn't require any authority to grant approval to register a username.

### 2.1.2 Ubiquitous access to login

Current mechanisms like Web3 providers on the web browser make it hard to access DApps from different devices and sandboxed environments like mobile apps [4]. DAuth aims to provide a way such that the users are able to *login* into the web apps by using a familiar username - password based authentication mechanism that is completely independent of the device or environment in which the user operates.

### 2.1.3 Decentralized

Current third party authentication mechanisms based on OAuth require a central authority to operate. Should one of these providers close down the OAuth API, the apps based on authentication from these services will be rendered useless.

DAuth tries to make way for a system that is completely independent of any central actors. The system must work even when there is only one user who might potentially be hosting her own DAuth service.

### 2.1.4 Free and instantaneous identification

Some authentication mechanisms use signed messages to identify a user. However this approach, though secure, has multiple problems. It incurs a transaction fees and takes time for the transaction to be confirmed in a block on the blockchain, taking about 12 seconds on Ethereum. DAuth aims to be instantaneous with a latency of a few milliseconds which is caused due to network transfers.

### 2.1.5 Tolerance to forgotten or compromised credentials

Though current mechanisms using Web3 and signed messages are fairly secure - it is not a feasible option for people to own hardware wallets or even install simple Web3 extensions on their browsers [5]. The alternative is to trust a third party with the private *and* public key of their account - as is the case with cloud wallets like Coinbase. This is a major security hole because if the keys are compromised, the user has no control over the account she *owned*. DAuth aims to decouple the keys used in signing transactions from the ones used for identification, so that the Ethers or coins in the wallet stay secure even if the authentication credentials are compromised. The risk associated with trusting a third party - if need be - is lesser. Also, if a user feels her credentials have been compromised, she may change the credentials using a transaction signed with the account's keys.

## 2.2 Components

### 2.2.1 Authentication Key Pair

Every address registered on the DAuth system must have an associated Public-Private key pair. This key pair is used in the *authentication game* that is described in section [?].

### 2.2.2 Smart Contract

The entire DAuth system is governed by a Smart Contract. This contract is responsible for maintaining a mapping between an address and the following.

- Username
- Authentication Public Key
- DAuth Server URL

### 2.2.3 DAuth Server

The *DAuth server* holds the user's private key and exposes an endpoint for the Authentication Game and a webpage for entering user credentials. Details for both of these functionalities are discussed in the next few sections. The user may use a third party DAuth server or host one on her own.

### 2.2.4 Verifier URL

The verifier is the entity that wants to identify the user, like the web app. This verifier must expose a REST endpoint for the Authentication Game.

## 2.3 Authentication Game

### 2.3.1 User registration

The user must send a message to the DAuth smart contract. This must be a signed message consisting of the desired username, the authentication public key and the DAuth URL. This transaction may incur transaction fees. This is a one time registration. The DAuth server stores the hash of the password and the authentication private key that is to be used for identification.

### 2.3.2 Login screen - 1

To request a user to login, the app must present the user with a input box to enter *only* the username. The user is expected to enter the password only on the DAuth Server she has registered with. Once the user has provided the username, the app must query the block chain for the DAuth server corresponding to the username provided and redirect the user to the webpage exposed by the DAuth Server, called *Login screen - 2*. The app also passes the verifier URL to Login screen - 2.

### 2.3.3 Login screen - 2

The user must enter the password on this webpage. The user must be sure that she is on the DAuth server she had registered with - by looking at the URL or any other unique identifiers on the webpage. The address of this webpage is a `GET` request made with parameters `action=login` made to the DAuth Server URL stored on the smart contract.

The password is never to be transmitted over the wire.

Once the user enters the password, two strings are generated

- Code - A random string. This identifies the current authentication session. The DAuth server must validate an authentication request identified by the code, exactly once. Once authenticated, all future sessions using this code will be rejected.
- HashCode - The hash of the *hash of password* and the above code.

These strings are then passed to the verifier URL.

### 2.3.4 Verifier URL

The user is redirected to the verifier URL which gets the username, code and hashcode from Login Screen - 2. It must fetch the DAuth Server and the authentication public key from the block chain corresponding to the username. It must then generate two strings.

- Secret - A random string known only to the verifier.
- Cipher - The Secret encrypted using the user's authentication public key.

Then, the verifier must send a request with the following parameters to the DAuth Server's verification endpoint.

- Code
- Hashcode
- Cipher
- Username

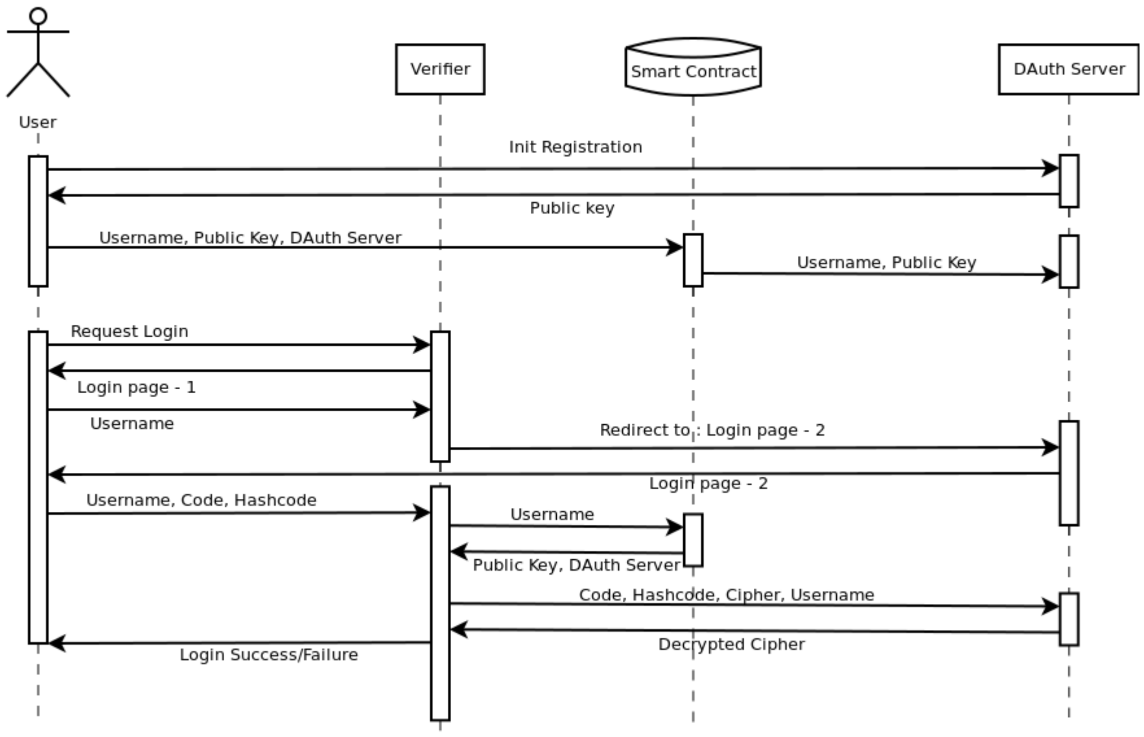
The response to the request is an attempted decryption of the Cipher provided. If the response is the same as the Secret used to generate the Cipher, the authentication is successful else not - the verifier URL may respond accordingly.

### 2.3.5 DAuth Server verification endpoint

The DAuth Server verification endpoint is a `POST` request made to the DAuth Server URL with the query parameter `action=verify`. The DAuth Server verification endpoint verifies if the code and hashcode it receives are valid. For it to be valid, the code must never have been used before in a successful authentication and the hashcode should match the hash of the code and the password hash it has stored locally.

If the code and hashcode are valid, it decrypts the the Cipher using the private key stored locally. If the decryption is successful, the endpoint responds to the verifier URL with the decrypted value.

## 2.4 Flow of control



# 3 Implementation

## 3.1 Smart Contract

The smart contract is deployed at the address *dauth.etherbase.eth*.

```
pragma solidity ^0.4.20;
contract Dauth {
    address owner;
    mapping ( address => string ) usernames;
    mapping ( string => address ) reverse_usernames;
    mapping ( string => string ) dauth_urls;
    mapping ( string => string ) public_keys;

    event NewUser(address from, string username, bool success);
    event NewVerification(address from, string username, string verificationType,
        bool success);

    function Dauth() public {
        owner = msg.sender;
    }

    function killDauth() public {
  if(msg.sender == owner){
      selfdestruct(owner);
  }
    }

    function set(string username, string dauth_url, string public_key) public returns (bool){
  if ( reverse_usernames[username] == 0 ){
      usernames[msg.sender] = username;
      reverse_usernames[username] = msg.sender;
      dauth_urls[username] = dauth_url;
      public_keys[username] = public_key;
      NewUser(msg.sender, username, true);
      return true;
  }
  NewUser(msg.sender, username, false);
  return false;
    }

    function getUsername(address user_address) public view returns (string){
  return usernames[user_address];
    }

    function getAddress(string username) public view returns (address)  {
  return reverse_usernames[username];
    }

    function getDauthUrl(string username) public view returns (string)  {
  return dauth_urls[username];
    }

    function getDauthPublicKey(string username) public view returns (string)  {
        return public_keys[username];
    }
}
```

## 3.2 Login Page - 1

A web app developer may choose to host a web page that accepts the username, fetches the DAuth Server from the smart contract and redirect to the appropriate Login Page - 2. However, for the sake of convenience DAuth provides a ready to use Login Page - 1, which takes the verifier URL as a query parameter and does the appropriate redirections. This is hosted at `https://dauth.co/utils/login`

To accept a login that is to be verified at `https://example.com/dauth-verify`, the verifier must redirect the user to `https://dauth.co/utils/login?redirect=https://example.com/dauth-verify`

The developer may choose to use a utility to create a "Login with DAuth" branded button at `https://dauth.co/plugin`.

## 3.3 Verifier URL

DAuth has developed a simple to use NPM package (`dauth-verifier`) that can be used out of the box by developers.

```
npm install dauth-verifier
```

```
const dauth = require('dauth-verifier');

router.get("/", function(req, res, next){
    dauth.verify(req.query.username, req.query.code, req.query.hashcode).then(function(data){
        console.log("Login successful");
        // ... Logic for successful login ...
    }).catch(function(error){
        console.log("Login Failed");
        // ... Logid for failed login here ...
    });
});
```

## 3.4 DAuth Server

You may run your own DAuth Server by forking the DAuth Server present at the DAuth Github repository : `https://github.com/madhavanmalolan/dauth`

## 4 Future Work

### 4.1 Support multiple block chains

At this point of time we support only Ethereum. We however intend to expand the same authentication model to multiple block chains that allow for smart contract programming. Stellar, Neo and EOS are in immediate sight.

It would be beneficial if the usernames registered be unique across blockchains. More research needs to happen to enable that in a decentralized way.

### 4.2 Data Verification

Authentication mechanisms like *Sign in with Google* provide the web developer with information like name, age, sex and email address. DAuth, to become a complete replacement to these OAuth based authentication providers, should be able to provide such information to developers at the discretion of the user.

## References

[1] `https://oauth.net/2/`

[2] `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md`

[3] `https://bitcointalk.org/index.php?topic=1942431.0`

[4] `https://www.reddit.com/r/ethereum/comments/73v7ih`

[5] `https://ethereum.stackexchange.com/questions/24871`