Distributed Sign-in: A decentralized, user-controlled authentication system

Key words: Web apps, authentication, social-engineering, cryptography

Noah Gallant
Columbia University
Implementation: https://github.com/NoahGallant/distributed-sign-in
8/20/2016

**This is not so much a polished paper but an in-depth README explaining the technology and motivation behind DS-i.*

Motivation/Problem

Part 1: A table of hashed passwords is no longer secure.

Web security is becoming an ever-growing topic as a majority of human civilization becomes connected via the Internet. Computer, cell-phone, and web applications require some sort of log-in system to verify identity before accessing user-specific data and functionality. This is nearly exclusively accomplished through connected one's account (via publicly listed email or username) to a password. Passwords are then stored in the web applications' databases as hashed values. A hashed value is one which can only be derived from its original value and not the other way around (a perfectly hashed value cannot be de-hashed). These hashed values are computed by a secret key stored on an apps' servers. It is important to note that databases and servers are often kept in separate web (and physical) locations, thus separating a key from the key-derived hash values in order to retain the integrity of the hash.

However, vulnerabilities in hashing algorithms and database systems mean there are frequently online postings of a hacked list of passwords, which can subsequently be translated to the original passwords by ever-evolving password-cracking software, or by a rainbow table (see: https://en.wikipedia.org/wiki/Rainbow_table). Thus, it no longer seems practical to keep a list of even the most securely hashed passwords in a single place.

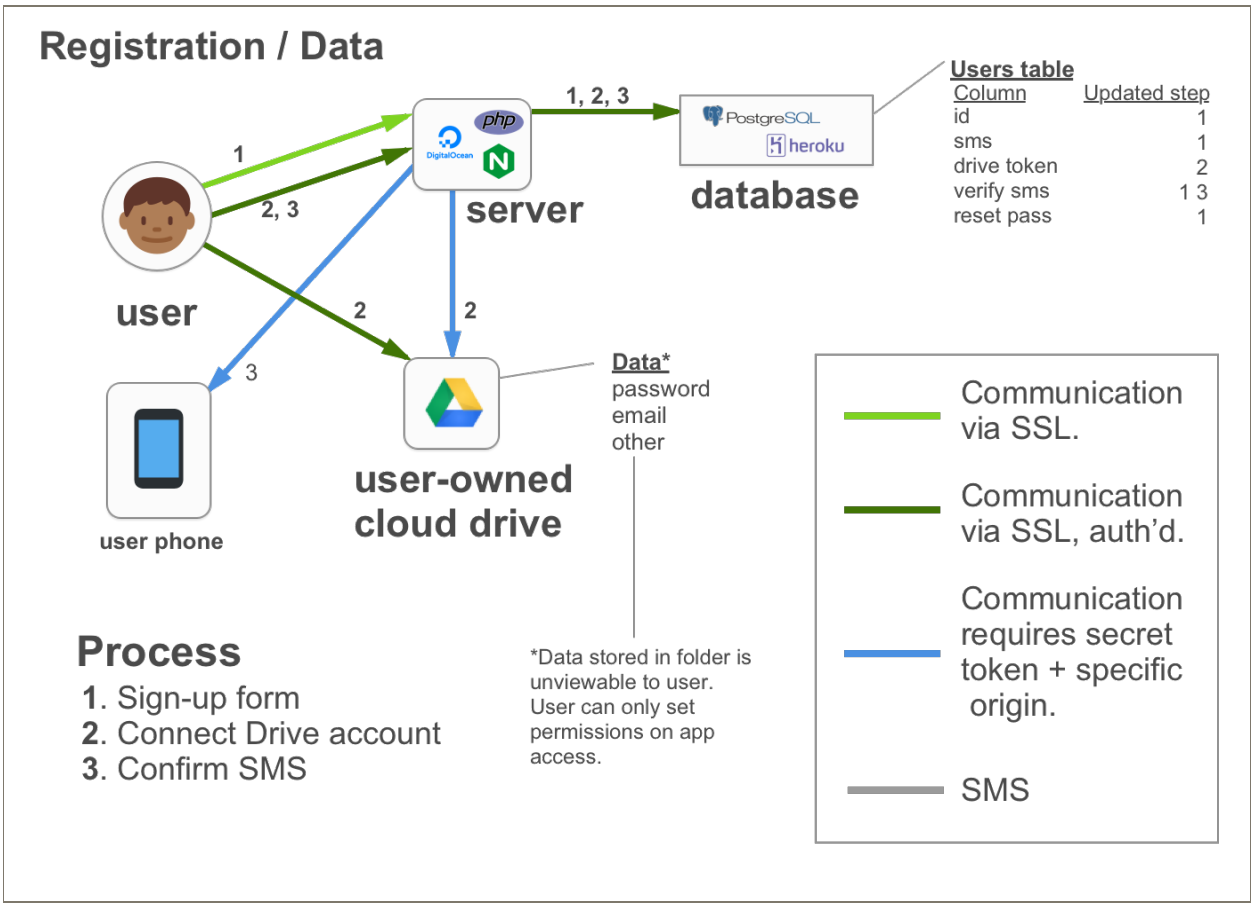
Part 2: Social engineering means we need to un-tangle our accounts, especially email.

Modern hacking has now evolved to a point where many high-level, high-profile hacks involve steps completely removed from a terminal on a computer. In order to target a specific person hackers may infiltrate someone's account by first targeting their email. They may try to gain access through calling a company and impersonating the individual, or simply figuring out the answers to their security questions. Once the hacker has gained access to someone's email account, they may then reset the password as most password-resets are accomplished via email.

Many modern applications also offer a way to log in through Facebook, Github or Google. This creates an additional susceptibility in the sign-in system that an application no longer controls the security of a log-in, authentication is externalized. This begs the question of responsibility and security of users' data, especially if the connected account is compromised.

Solution

Registration System / Data Model



This diagram is an attempt to model the registration system implemented within DS-i. The actual system maintained by the application is a simple web app stack, in this case LEPP. User data (beside their mobile phone number [SMS]) is all stored in their Google Drive within an App Data folder. The App Data folder can only be accessed by white-listed IP's, *however*, it can be deleted and its permissions modified *only* by the user. This means two very important things:

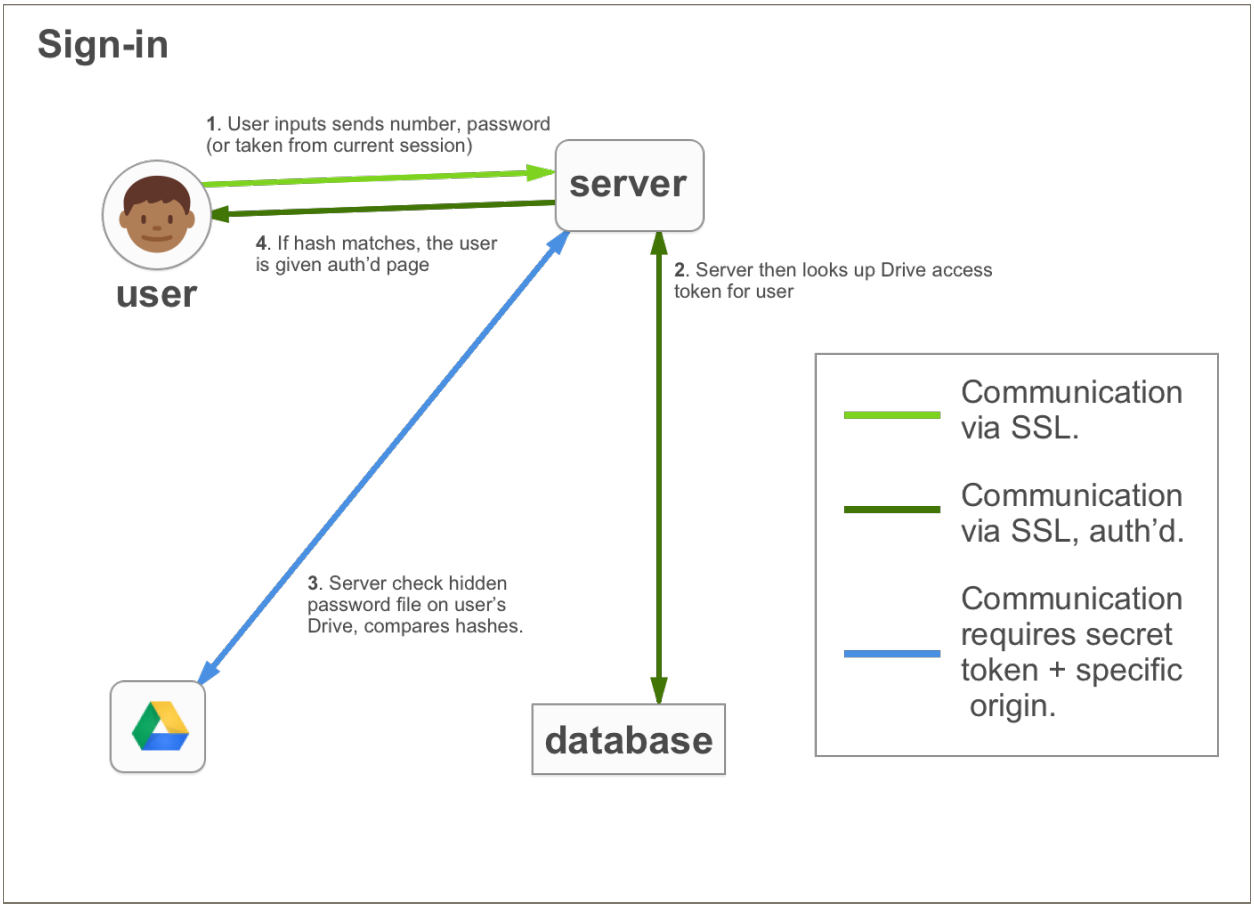
1. The user controls *all access* to their data.
2. The user's Google account cannot be used to compromise the account on the DS-i system.

These two key aspects of the DS-i system fit nicely within the motivation for this project. Furthermore, the mobile number of the user is used as the "username," this is done in order to emphasize

to the user the new importance of their phone in this login system as is noticeable in the reset methodology.

To learn more about the technologies used for encryption, random keys and data storage skip to the "**Technology**" section.

Sign-in System

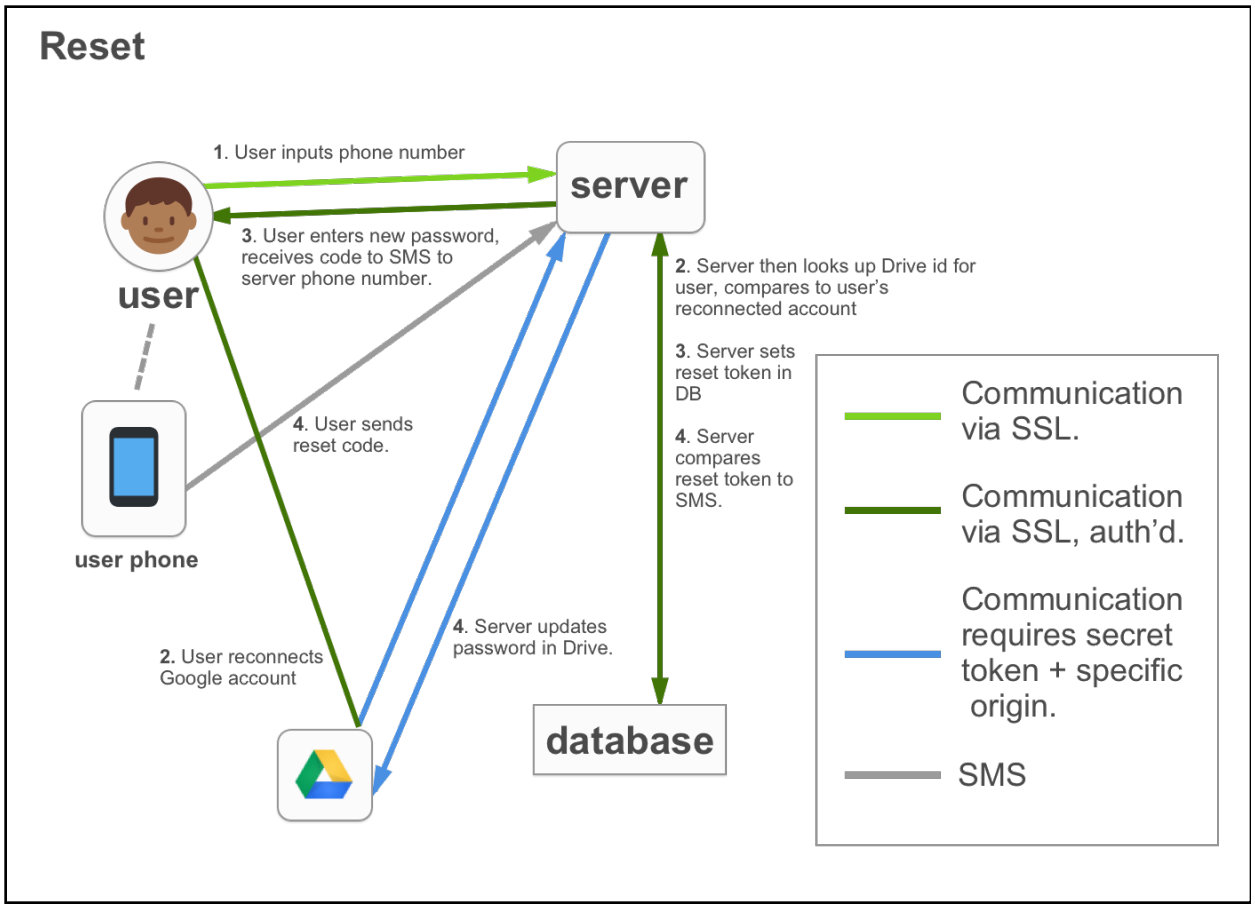


The sign-in methodology mimics the ideas emphasized in authorization. Once a user requests a secure page on a DS-i system, the server will check their credentials (either form input or encrypted session vars). The server retrieves the specific user Drive access token from the database, and then compares the hashed passwords. If the password is verified the server is given the user data, and may then give the user the appropriate access or information.

Reset System

While the registration/sign-in system is built to subvert database and encryption focused hacking, the reset system is specifically constructed to subvert social-engineering attacks. The diagram depicts the password recovery/reset flow which essentially requires the user to send an sms from their phone to the app server in order to set their new password*. Given developments in femto-cell technology and phone hacking in general, requiring a user to send an sms requires a significant level of hacking, including deleting the text from the phone in the case that the real user might see a password reset and remove

account access before any malicious activity occurs. This reset prompt also requires the original Google account to be confirmed giving the reset a minimal-effort 2FA.



*In the week this report was written I saw a similar technology of sending an SMS to reset a password applied as authentication API. After a lot of searching on Github and through Facebook groups I was unable to find it. It was done by a Brown CS undergraduate, please let me know if you find it.

Discussion

What is this?

This project was not created to be a complete solution to authentication and sign-in for all web-apps, but as I built it, DS-i gained more and more features, and at this point acts as a secure library for PHP authentication. By no means should you download the code, run it and expect a scalable, hyper-functional auth system, but it provides different perspective on hands-on, reliable, user-owned, app-controlled user management.

The code in the Github is not fully commented, nor organized and I plan on adding releases in the near future with accessible design and a more focused UX/UI (there is none at the time of writing this).

In a perfect reality this project is a stepping stone to showing an authentication system that stays out of the way, but gives users control and security as we head towards a more interconnect world.

Why not just use …?

Why not use Authy, or Laravel or Auth0? See "**Motivation.**"

There are many authentication systems out there, and some even *fully* decentralized. One great example is the Ethereum (https://www.ethereum.org) system. It is great, but I have found that it requires a heightened technical know-how and software beyond what the average internet user has at their fingertips in order to be wholly applicable in the common day web app usage.

A new standard of user-owned data has arrived, largely supported by independent initiatives such as Solid (https://solid.mit.edu), a personal favorite. I hope DS-i shows how we can integrate such a standard into our existing web app systems.

How to hack it?

To the best of my knowledge there are two methods for hacking a DS-i system:

1. Hack the server. Without the server it is impossible(ish) to gain access to a user's Drive and the subsequent Auth files in order to impersonate the user.
2. Compromise the user's phone and Google account. Short of that, there are minimal ways to socially engineer the DS-i system short of coercing the password out of a subject.

What about zero knowledge proofs, or another new cryptography standard?

I'm all for it! Feel free to hack away at my source code! Submit a PR, I'd love to see the communities ideas on how to improve and change the design of DS-i.

Technology

Web tech

- Ubuntu Droplet on Digital Ocean (https://www.digitalocean.com)
- PHP 7.0.8
- See composer.json for dependencies

Encryption

- pbkdf2 (sha256) for pass hashing with openSSL generated salt
- openSSL for encrypting user Drive token

SSL

- Let's Encrypt (http://letsencrypt.org)

Data storage

- Heroku PostgreSQL (https://www.heroku.com/postgres) [FREE]

SMS

- Twilio (http://twilio.com)
- Zapier (http://zapier.com)

Email
- PHPMailer
- Mailgun (http://mailgun.com)

Usage

The usage of this library is currently pretty involved. It requires setting up developer accounts on Google and Twilio at least. Mail can certainly be run through the DO server. The database may also be run on the DO server but this might make it more susceptible to attack as more of the app data is in one location. Once you set up these accounts the only code you have to change is the place the php scripts in a login/ directory, run composer and update the .env.php file.

Any page you want to make secure you simply have to insert the following code at the top of:

```
<?php
require PATH.'login/login-
engine.php';
$user = authorizeSession($db);
?>
```

Current Features

In building a demonstration of DS-i I wanted to experiment with the usability of the authentication in a realistic system, and thus nearly created a full-fledge sign-in and profile management application. These are the current functionalities in the current library (0.1):

- Account creation
- SMS confirmation
- Email confirmation
- Password reset
- Edit profile
- Sign in / out
- Bot detection (reCAPTCHA)
- CSRF protection
- Simplified user features (similar to Eloquent models in Laravel)

Future

I would like this project to head towards an easily implementable system. The following is currently on the TODO list:

- Timeout of SMS's
- Class everything
- Remove $db from function calls
- For use in Electron, Ionic
- full CSRF implementation
- Mailing list
- Sanitize inputs