CENG 499

Final Project Report

air.auth

By:
Group Number: 6
Anubhav Mishra (anubhav@uvic.ca) - V00740087
Cole Bosmann (cboss24@uvic.ca) - V00722585
Conrad Foucher (conrad@foucher.ca) - V00721922

Supervisor: Dr. Kin F. Li
Submitted: August 1st 2014

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 What is a Password Manager

As technology continues to advance, it proliferates deeper and deeper into our everyday lives. As a result, the average user has an ever-increasing number of accounts for various sites and services. Each of these accounts requires a username/password combination, which is left to the user to remember. This is easily accomplished for a few accounts, however as the number of accounts increases, the task becomes more and more difficult. Users compensate for this increased difficulty in two main way:

- **Users select simple usernames and password.** In this context, "simple" represents short, dictionary words with few capitals, symbols, or numbers. This reduces account security by allowing attackers to more easily guess a user's credentials (brute force attack).

- **Users reuse usernames and passwords over multiple sites/services.** This reduces account security by allowing an attacker to compromise multiple accounts with one correct credential set.

Password managers were created to alleviate user strain, while maintaining account security. The general idea is to allow a user to store their various site passwords in a central location, and gain access to them using a single master password. This design allows users to maintain strong username/password combinations for their various sites/services, while requiring them to only memorize a single master password.

There are numerous variations of password managers. Two main distinctions include:

- **Password Generator vs Password Storage.** Early password managers attempted to increase site security by generating long, pseudo-random passwords for user's accounts. This increased security, because dictionary attacks were no longer effective, however it ensured that users could only access their sites through the password manager. If the service was temporarily down, the user's would lose access to all their accounts. Future iterations (password storages) corrected this by allowing users to still select their site passwords, which does not increase security, but increases the flexibility from the user's perspective.

- **Offline vs Online.** There exists both application based (offline) and web based (online) password manager services. Both variations have their advantages and disadvantages. Offline password manager's are considered more secure, as the data is held locally on a user's machine. In this scenario, the benefit of compromising a system is minimal for an attacker, as they will only gain access to one user's set of accounts. The downside of an offline approach is that users only have access to their passwords on their personal devices. This accessibility is ameliorated in a web-based approach, where users can access their passwords from any device with an internet connection. However, due to their very nature, web-based managers must be exposed to the internet, making it potentially vulnerable to attacks. Due to the valuable data contained in web-based managers, they are also attractive targets for attacks.

The system implemented in this paper is a password storage, web-based system. These distinctions were chosen in order to provide the best usability for the end user. The inherent security concerns of these choices and our proposed solutions will be addressed later in this report.

1.2 What is Leap Motion

Leap Motion, Inc. is a San Francisco based company that manufactures and markets a computer hardware sensor that supports hand and finger motion as input. This device is called a Leap Motion Controller. The Leap controller is a small USB device, designed to be connected directly to a user's computer and placed on the user's desktop. It uses two IR cameras and three IR LEDs to capture hand motion within an approximate 1 meter hemisphere surrounding the controller. To accomplish this, the LEDs project IR light onto a user's hand. A portion of this projected light is reflected back towards the IR camera, which captures approximately 300 frames per second.

The captured data is then sent to the host computer via USB for further processing and analysis. This processing and analysis is proprietary information held by Leap Motion, however the "complex math" of this step yields a 3D image of the user's hand(s). This 3D image provides information about the position and dimensions of various hand components (palm, fingers, joints, etc…). Leap Motion then exposes this data through an SDK, which allows developers to build applications using this revolutionary hardware as a basis. Leap Motion's 'Airspace' is an online app repository that hosts such apps and showcases the limitless possibilities that come when developing with Leap Motion. The system implemented in this paper is one such example of what is possible using Leap.

1.3 What is Air.Auth

1.3.1 Overview

Air.Auth is a fusion of the two previously discussed technologies. First and foremost, it is a web-based password manager. The main service it provides is to allow users access to all their site specific passwords using a single master password. Where Air.Auth differs from traditional managers is how this master password authentication is performed. Using the Leap Motion, Air.Auth authenticates using a user's hand dimensions and a combination of hand poses. Once this authentication is performed, Air.Auth will also allow users to launch sites using pre-assigned poses. Once a site is launched, a new tab is created, the site is loaded, the user's credentials are populated, and the user is logged in. From start to finish, Air.Auth allows a user access to their sites without needing to touch a keyboard or a mouse.

1.3.2 Goals

The goals of this project are as follows:

- To explore the Leap Motion V2 Beta, and construct a way to utilize a user's hand for authentication.
- To build a fully functioning system, ready for user interaction. This includes traditional web service components such as account creation, account recovery, account management, session management, and site usage support.
- To construct this service using industry standard security practices in order to properly protect user's information.
- To utilize cutting edge technology for Air.Auth's sub-components, to facilitate scalability should the user base grow.
- To share Air.Auth under an open source license, and craft it in a modular fashion to enable other developers to build upon it.

This report will enumerate on each of these goals and show how the Air.Auth team has strived to achieve them.

2. System Architecture

The system was built entirely using javascript. Javascript is known as the "Ultimate Scripting Language" amongst its supporters and allows for asynchronous execution. It is supported by all of the main browsers, and writing both the frontend and backend in the same language brings a sense of standardization and fluidity to the system. Figure 1 below shows a high level view of the components that compose Air.Auth.

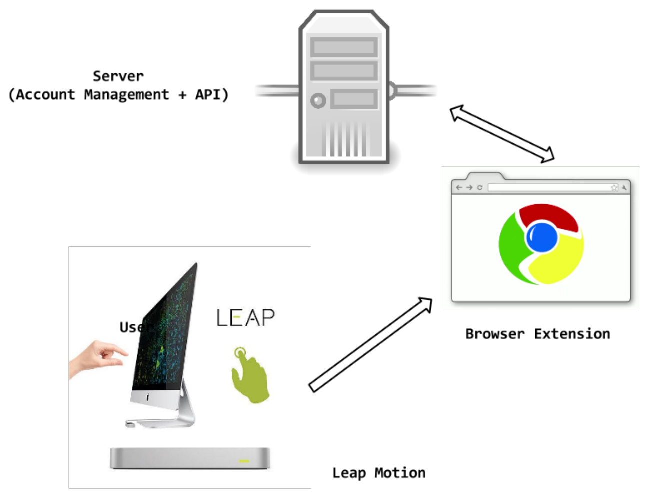


Figure 1: Top Level System Architecture

2.1 Frontend:

The frontend is everything that a user sees and interacts with. The Air.Auth frontend is built using javascript, jQuery and Twitter's bootstrap CSS framework. jQuery is used to implement client side HTML scripting and allows easy asynchronous HTTP requests to Air.Compute (API). Twitter bootstrap offers ready-made grid systems that allow for rapid frontend development. Bootstrap is also completely customizable. The grid system is responsive so it can be used across all screen sizes such as mobile devices, tablets and laptops. Bootstrap also comes with its own javascript class that allows modals, alerts, tooltips and button manipulation.

2.2 Backend:

The backend encompasses the system component that power the Air.Auth chrome extension. It is divided into the following unique components:

- Node.js application
- Nginx Webserver
- MySQL (Relational Database)
- Redis (NoSQL Database)

2.2.1 Node.js Application

Most of the Air.Auth backend is powered by Node.js using express.js as the web application framework. Express.js has a minimal set of robust features that allows for rapid web application deployment. It allows server-side scripting using javascript and gives huge scalability advantages. Node.js is built for real-time systems and uses an asynchronous non-I/O blocking event model. The Node.js app runs on port 3000 which is not the default web server port. As a result, there was a need to use Nginx to forward all the default web server requests to Node.js.

2.2.2 Nginx Webserver

Nginx is very popular web server software used by multiple large-scale companies to handle a large amount of concurrent requests. As mentioned above Node.js app runs on port 3000, which is not the default web server port. This was done for added security reasons as the most denial of service attacks target default web ports. To allow access to the application Nginx was configured to forward all the default web server requests to Node.js. If an attack were to occur it would be very simple to disable the forwarding and move the site to a new port.

2.2.3 MySQL (Relational Database)

MySQL is one of the most popular open source relational databases. It is also easily configurable and has been battle tested for years by various large-scale companies. The benchmark tests for MySQL are positive and improve with every release. In this project, MariaDB was used as the primary database. MariaDB is a fork of MySQL and has all the features that are supported by MySQL. It also possesses optimizations in terms of performance, making it an ideal choice.

2.2.4 Redis (NoSQL Database)

Redis is an In-Memory Key-Value store that is made to increase large scale application performance. It has variety of data types that can be used for various purposes. These data types include strings, hashes, lists, sets and sorted sets. Redis is single threaded, meaning only one process will run on one core and will not utilize any other cores concurrently. Since Redis is an in-memory database, all of its data is stored in the RAM. Redis was used for session storage for users logged into Air.Auth and using the chrome extension or the account management system.

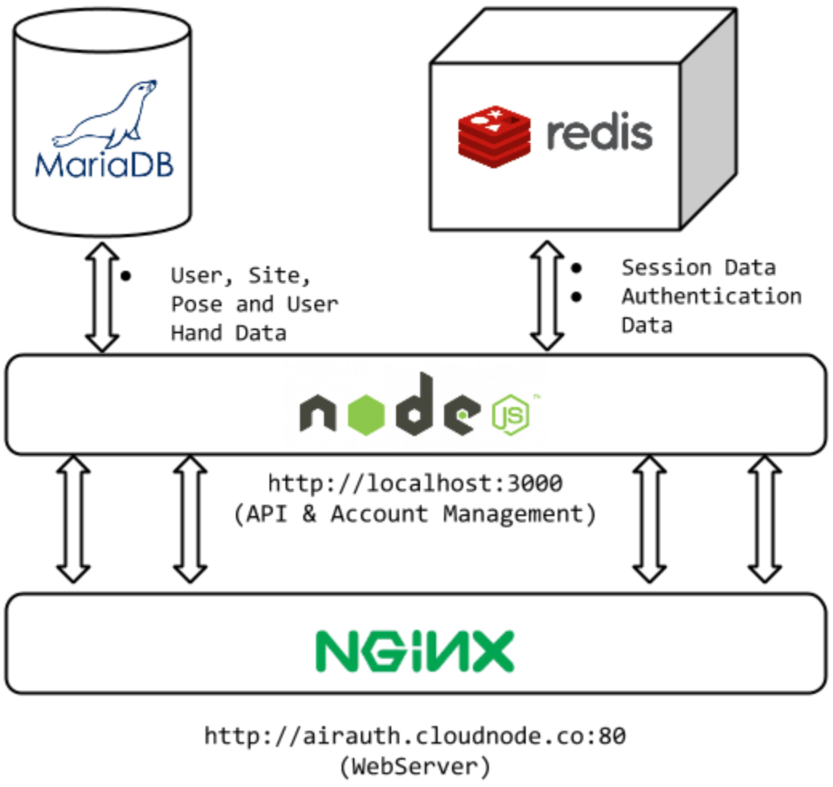
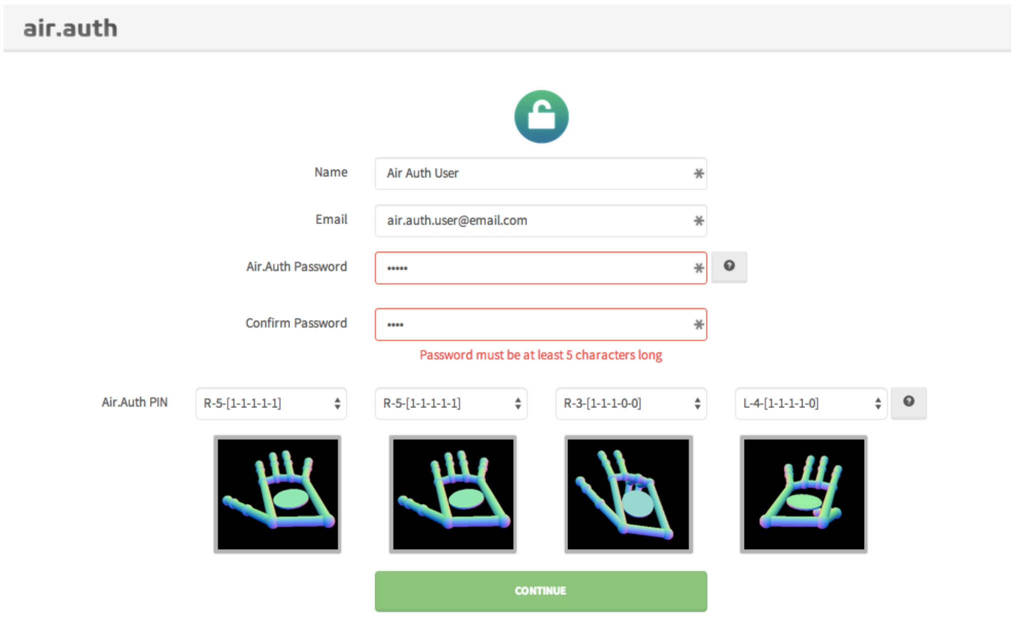

MariaDB
redis
• User, Site, Pose and User Hand Data
• Session Data
• Authentication Data
node JS
http://localhost:3000
(API & Account Management)
NGINX
http://airauth.cloudnode.co:80
(WebServer)

Figure 2: Backend Component Interactions

3. User Functionality

3.1 Account Creation

Like any other service, the first task a user must complete to use Air.Auth is registration. Registration is done locally within the Chrome extension and consists of two steps. In the first step which is shown below in Figure 3, the new user provides their name, contact email address, Air.Auth password (master password), and identifying PIN. Javascript runs locally on this page to ensure that information is entered correctly, such as the password hints shown below in red. Potentially confusing fields, such as password and PIN are also given hint buttons. Once all fields have been filled and checked by the local javascript, the greyed out "Continue" button will be enabled and the user can proceed to the next step.



Name: Air Auth User
Email: air.auth.user@email.com
Air.Auth Password: •••••
Confirm Password: ••••
Password must be at least 5 characters long
Air.Auth PIN: R-5-[1-1-1-1-1] R-5-[1-1-1-1-1] R-3-[1-1-1-0-0] L-4-[1-1-1-1-0]

Figure 3: Registration UI (Step 1)

After completing the first registration step, the user is now ready to submit their biometric hand information. This procedure requires the user to place their hand at a specified location over the Leap Motion. This page is interactive, and aids the user by showing them a shadow of their hand in real-time and hints on how to move their hand into the collection zone. In order to complete this page a user must successfully scan their hand 5 times, with their progress being actively displayed. Once 5 scans have been completed, the user has completed registration.

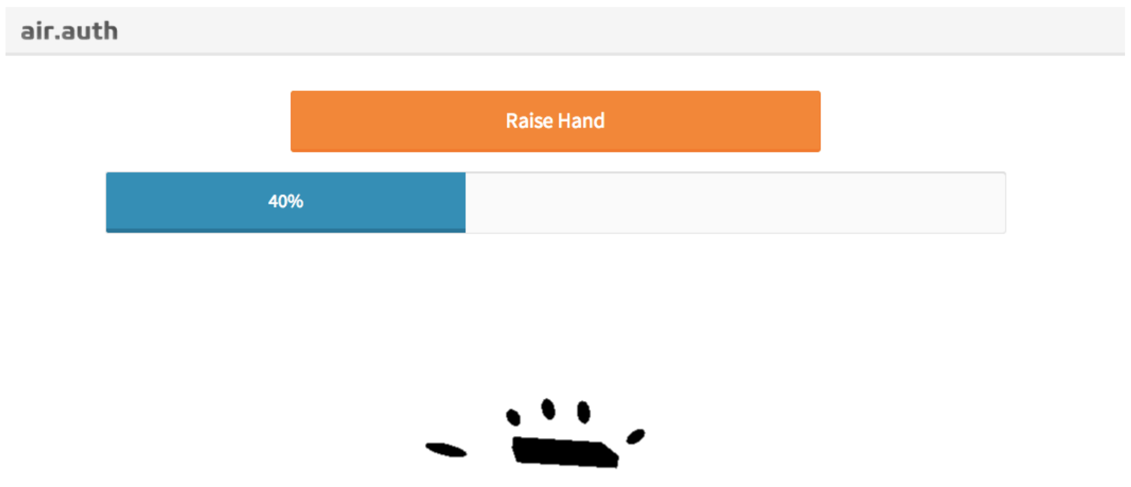


Figure 4: Registration UI (Step 2)

3.2 Account Management

Once a user has successfully registered, they are automatically redirected to their dashboard. The Air.Auth dashboard, shown below in Fig 5, resides on a remote server. A user can access this dashboard with their email and master password from any internet enabled device. This dashboard allows users to view and manage their saved sites, and manage their account.

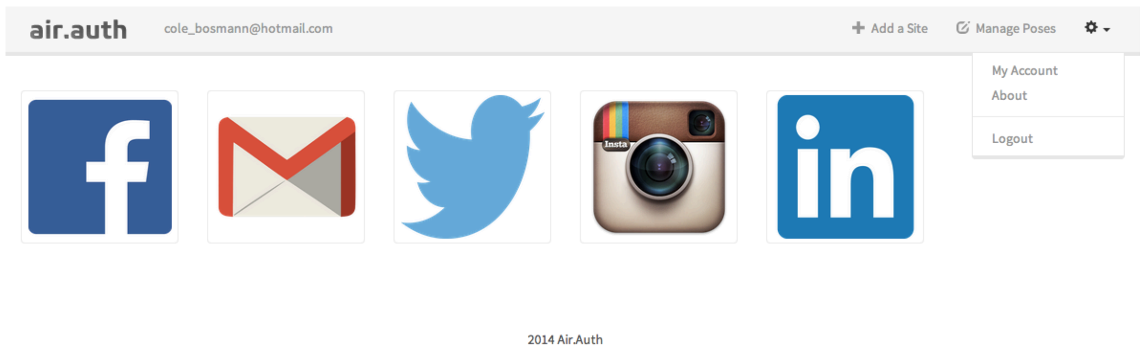
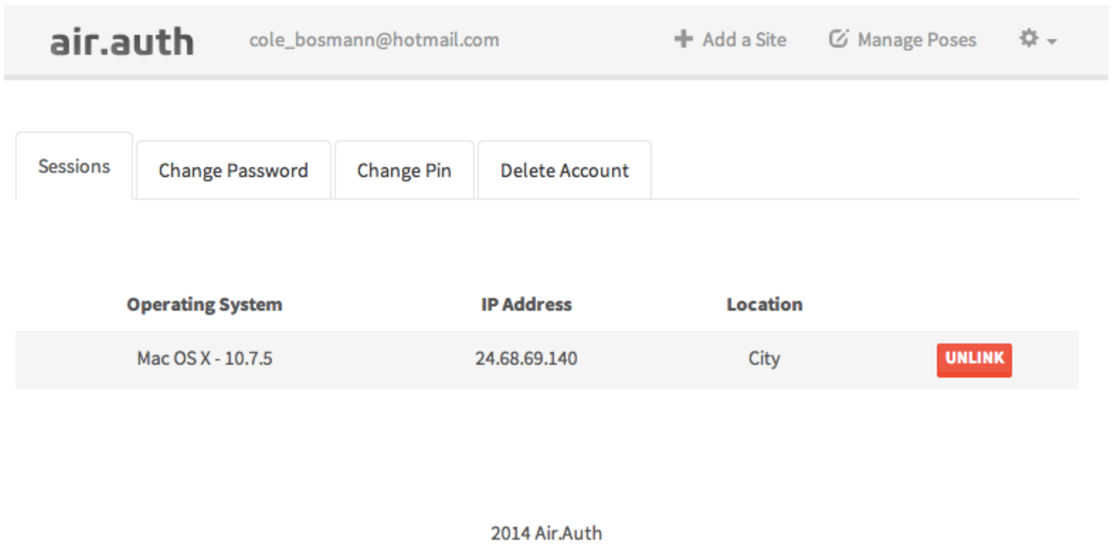


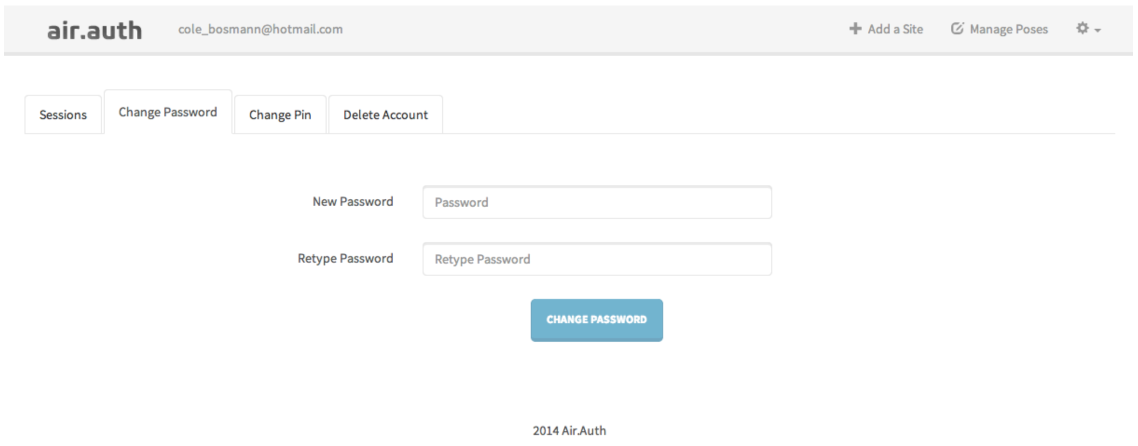
Figure 5: Dashboard

In order to manage their account, a user can either click on their email address in the top left of the dashboard, or on “My Account” in the settings dropdown. This will direct user’s to the “My Account” page. Within this page a user can view/terminate their active sessions (Figure 6), change their master password (Figure 7), change their authentication PIN (Figure 8), or delete their account (Figure 9). Once again, there is javascript running locally on this page, ensuring that all fields are correctly filled before an action can be submitted.



Operating System	IP Address	Location
Mac OS X - 10.7.5	24.68.69.140	City

Figure 6: Session Management



New Password: Password
Retype Password: Retype Password

Figure 7: Change Master Password

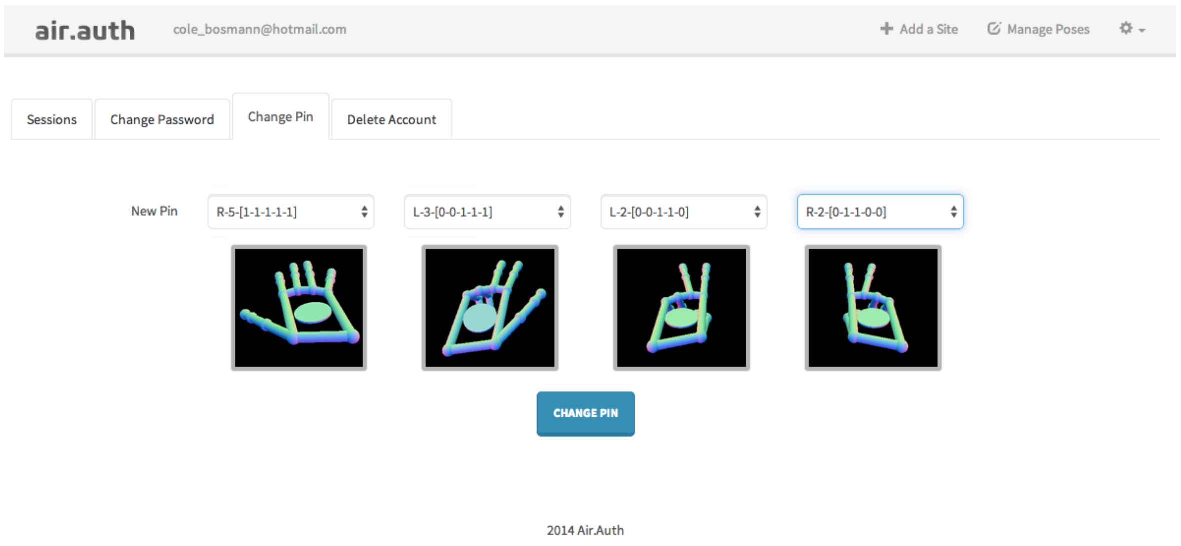
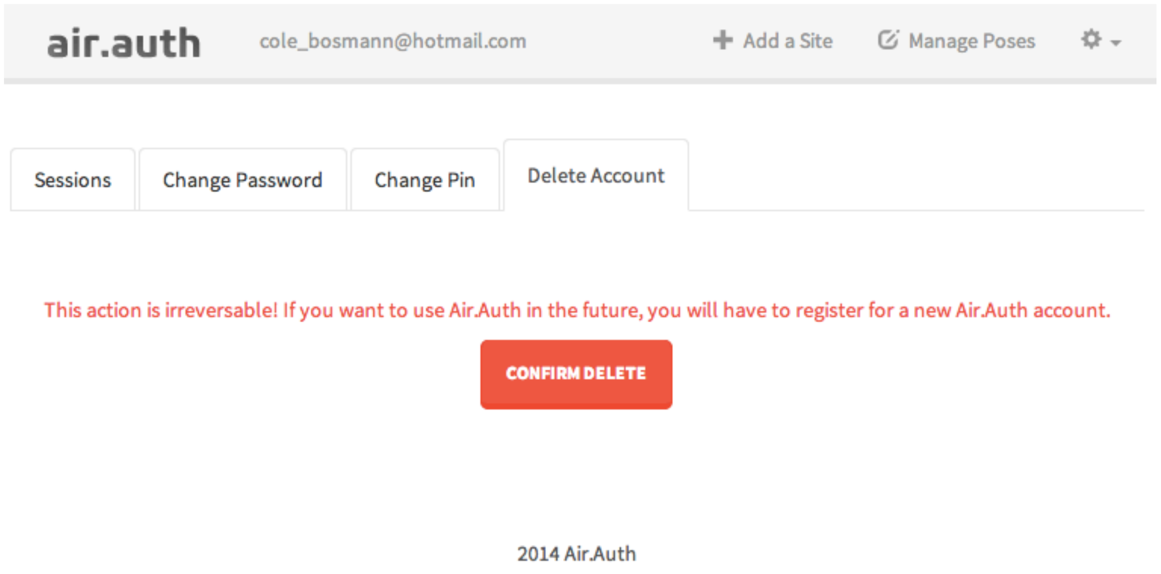


Figure 8: Change PIN

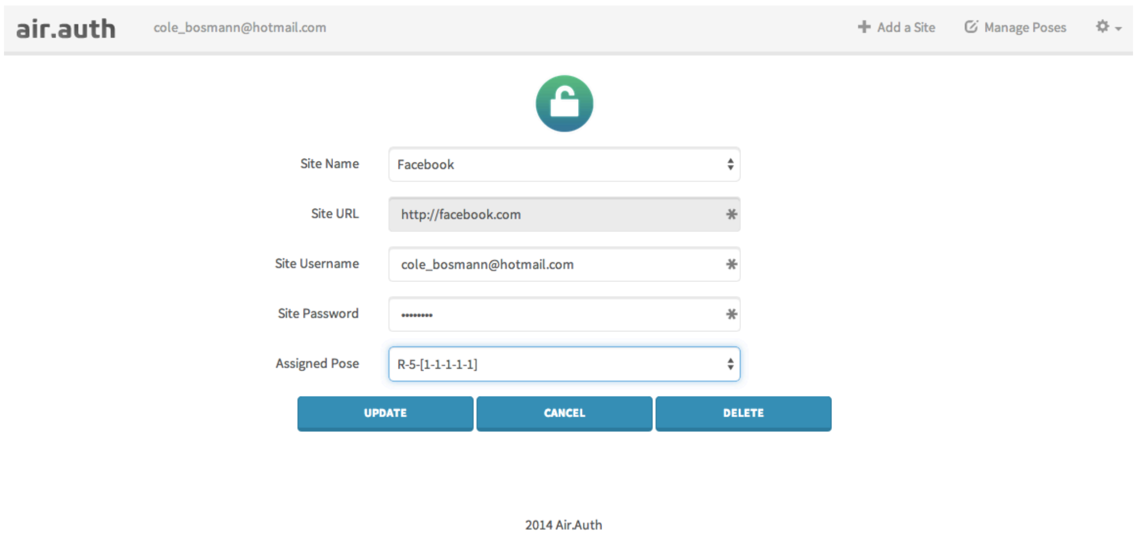


This action is irreversable! If you want to use Air.Auth in the future, you will have to register for a new Air.Auth account.

Figure 9: Delete Account

3.3 Site Management

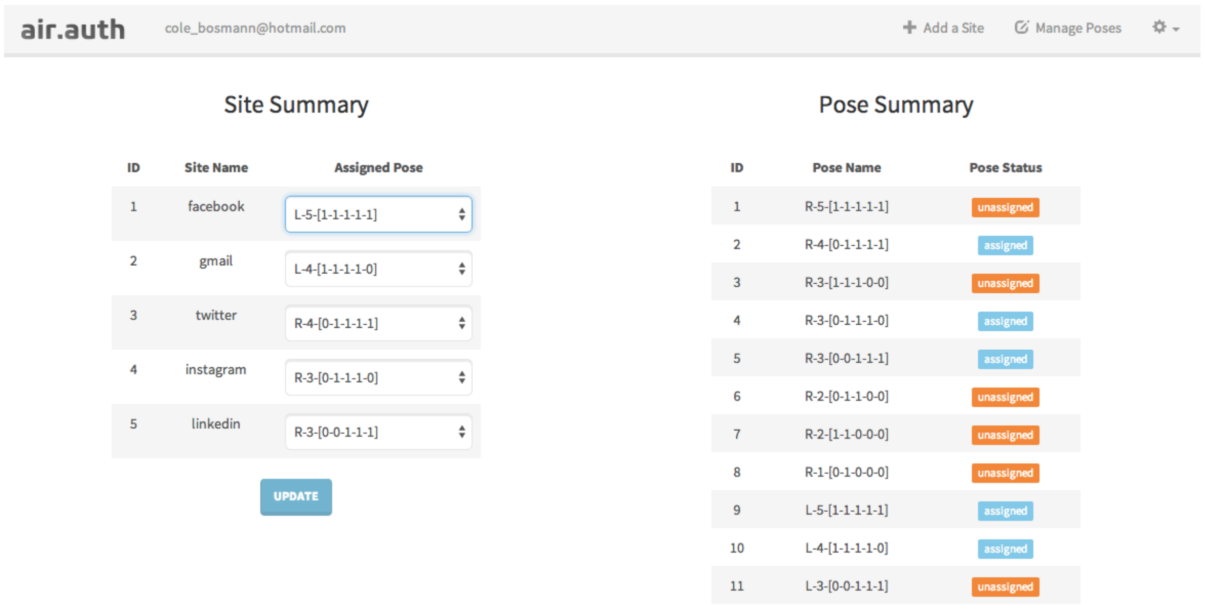
Through the Air.Auth dashboard, a user is also able to add and edit their list of saved sites. To edit a site, a user simply has to mouse over the site icon on the dashboard. To add a site, a user must select the "Add Site" button from the upper navigation bar. Both add and edit actions direct a user to the same screen, shown below in Figure 10. However, in the edit case, all the fields will be populated with their prior values. Once a user has selected a site, provided a username/password, and chosen a launching pose, they are able to add that site to their account by selecting the "Save" or "Update" button. At any time during the process, a user can also choose to discard their add/edit by selecting the "Cancel" button, or delete this site from their account by selecting the "Delete" button.



Site Name	Facebook
Site URL	http://facebook.com
Site Username	cole_bosmann@hotmail.com
Site Password	••••••••
Assigned Pose	R-5-[1-1-1-1-1]

Figure 10: Add/Edit Site

Once a user has a added some sites, it becomes necessary to facilitate easy re-assignment of poses to those sites. This feature can be accessed by selecting the "Manage Poses" button in the upper navigation bar. Selecting this button will reveal the page shown in Figure 11. On the right hand side of this page is a summary of which poses are assigned or unassigned for this user. On the left hand side of the page, a user is able to quickly reassign which poses are assigned to each site. As a user interacts with the left side of the page, the right hand side is refreshed dynamically. Javascript is also running locally to ensure that the drop down lists for each site contain only unassigned poses. Once a user is satisfied with their assigned poses, they can submit their changes by selecting the "Update" button.


air.auth cole_bosmann@hotmail.com

Site Summary

ID	Site Name	Assigned Pose
1	facebook	L-5-[1-1-1-1-1]
2	gmail	L-4-[1-1-1-1-0]
3	twitter	R-4-[0-1-1-1-1]
4	instagram	R-3-[0-1-1-1-0]
5	linkedin	R-3-[0-0-1-1-1]

Pose Summary

ID	Pose Name	Pose Status
1	R-5-[1-1-1-1-1]	unassigned
2	R-4-[0-1-1-1-1]	assigned
3	R-3-[1-1-1-0-0]	unassigned
4	R-3-[0-1-1-1-0]	assigned
5	R-3-[0-0-1-1-1]	assigned
6	R-2-[0-1-1-0-0]	unassigned
7	R-2-[1-1-0-0-0]	unassigned
8	R-1-[0-1-0-0-0]	unassigned
9	L-5-[1-1-1-1-1]	assigned
10	L-4-[1-1-1-1-0]	assigned
11	L-3-[0-0-1-1-1]	unassigned

Figure 11: Manage Poses

3.4 Account Linking

Once a user has successfully created an account and added sites, they are now ready to begin using Air.Auth. The first step is to link their account to their current computer. This is done locally by clicking the Air.Auth icon in Chrome. This action will cause the page displayed in Figure 12 to be shown. On this page a user must login with their email and master password. Once this is complete the account is linked. This page also enables users to retrieve a forgotten password, which triggers a password reset email to be sent. A linked account shows Air.Auth that this computer has permission to attempt to launch that user's sites. A linked computer will also appear in the Active Sessions tables shown in Figure 6.

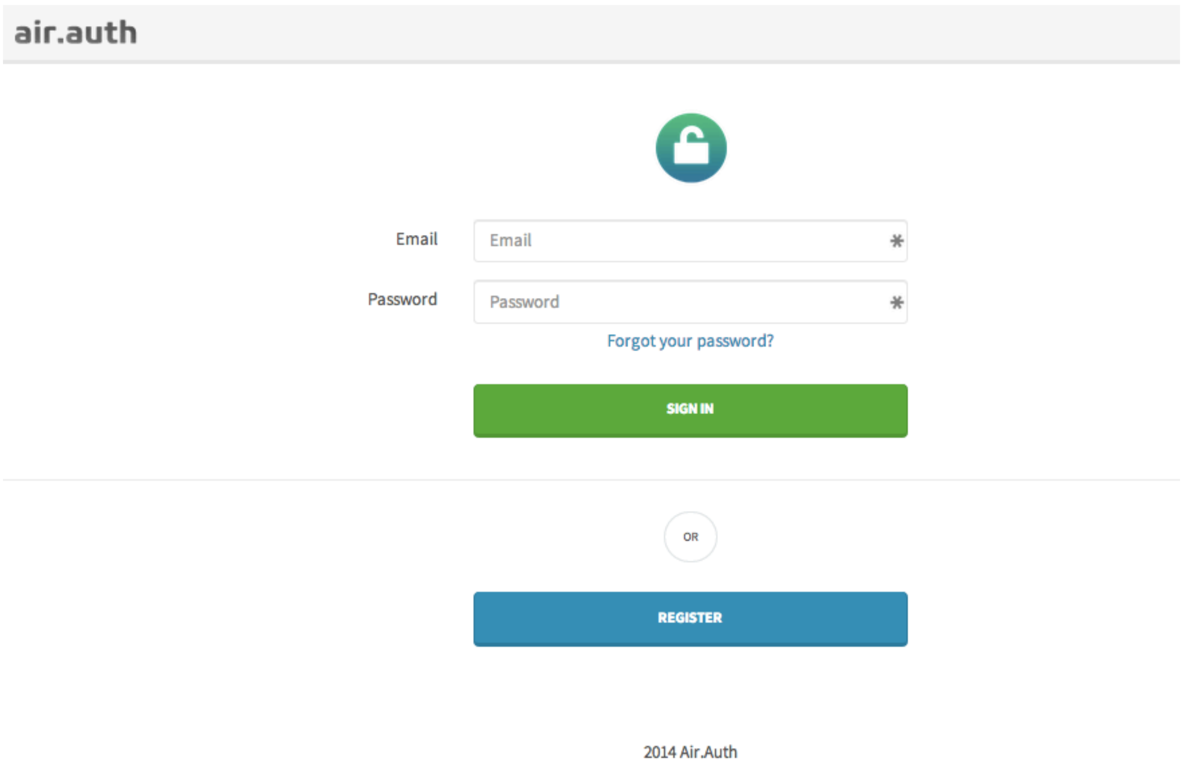
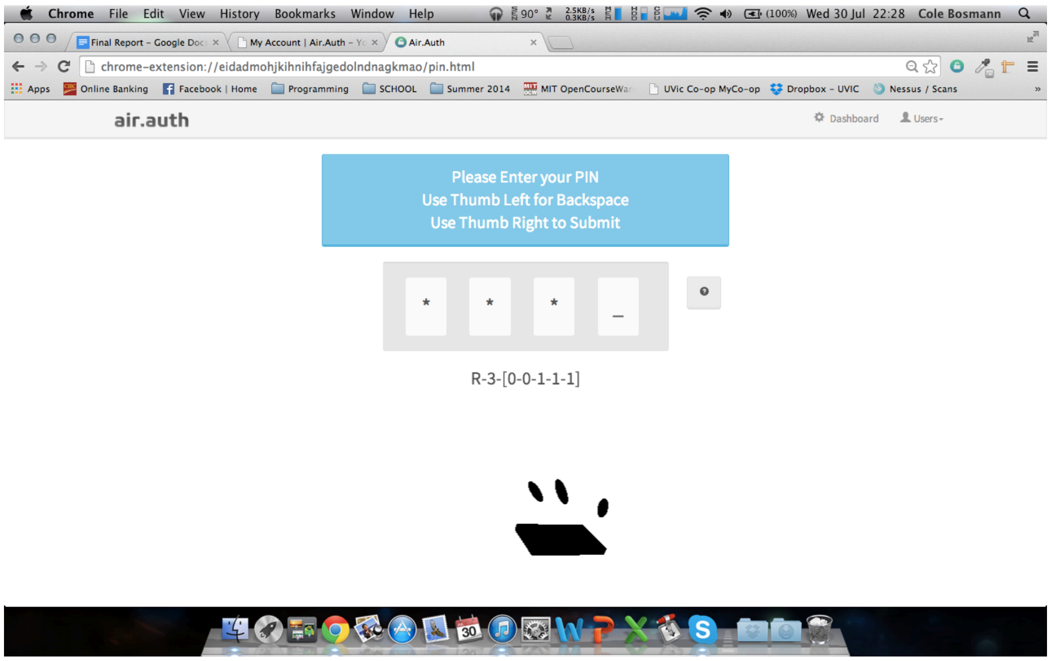


Figure 12: Account Linking

3.5 Site Launching

Once a user has linked their account to a computer, they can begin accessing and launching their sites. This is a three stage process. In the first step, a user performs their pose PIN code. This is done by sequentially holding each pose for 1 second over the Leap Motion. If an error is made, the user can make a "thumb left" pose to delete the previously entered PIN digit. Once a user has entered their 4 PIN digits, they perform a "thumb right" pose to submit. This step is shown below in Figure 13. In this Figure, a user has already entered 3 digits of their PIN, and has just had their final digit recognized.


Chrome File Edit View History Bookmarks Window Help
Wed 30 Jul 22:28 Cole Bosmann
Final Report - Google Docs
My Account | Air.Auth
Air.Auth
chrome-extension://eidadmohjkihnihfajgedolndnagkmao/pin.html
Apps Online Banking Facebook | Home Programming SCHOOL Summer 2014 MIT OpenCourseWare UVic Co-op MyCo-op Dropbox - UVIC Nessus / Scans
air.auth
Dashboard
Users

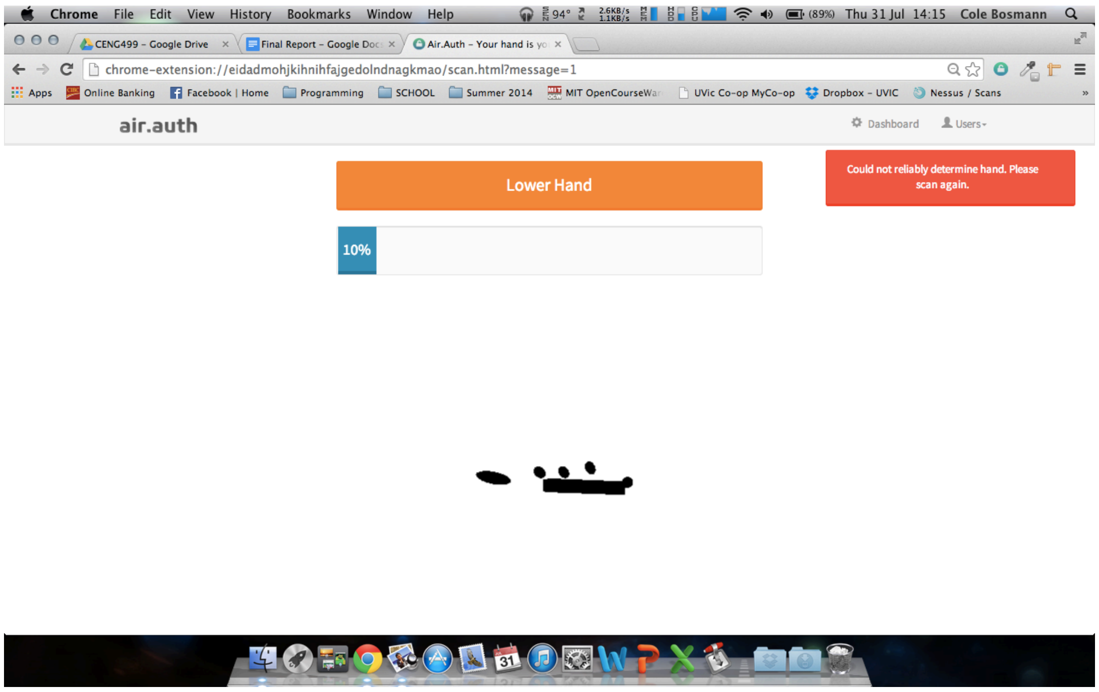
Please Enter your PIN
Use Thumb Left for Backspace
Use Thumb Right to Submit

* * * _

R-3-[0-0-1-1-1]

Figure 13: Pin Entry

The next step in launching is to perform a hand scan. This is conducted similarly to the initial hand registration. A user can view their hand shadow on the screen, and is provided with prompts as to where their hand must be placed. Once a sufficient amount of data samples have been collected, the user is moved to the next step. This step is shown below in Figure 14.


Chrome File Edit View History Bookmarks Window Help
Thu 31 Jul 14:15 Cole Bosmann
CENG499 - Google Drive
Final Report - Google Docs
Air.Auth - Your hand is yo
chrome-extension://eidadmohjkihnihfajgedolndnagkmao/scan.html?message=1
Apps Online Banking Facebook | Home Programming SCHOOL Summer 2014 MIT OpenCourseWare UVic Co-op MyCo-op Dropbox - UVIC Nessus / Scans
air.auth
Dashboard
Users

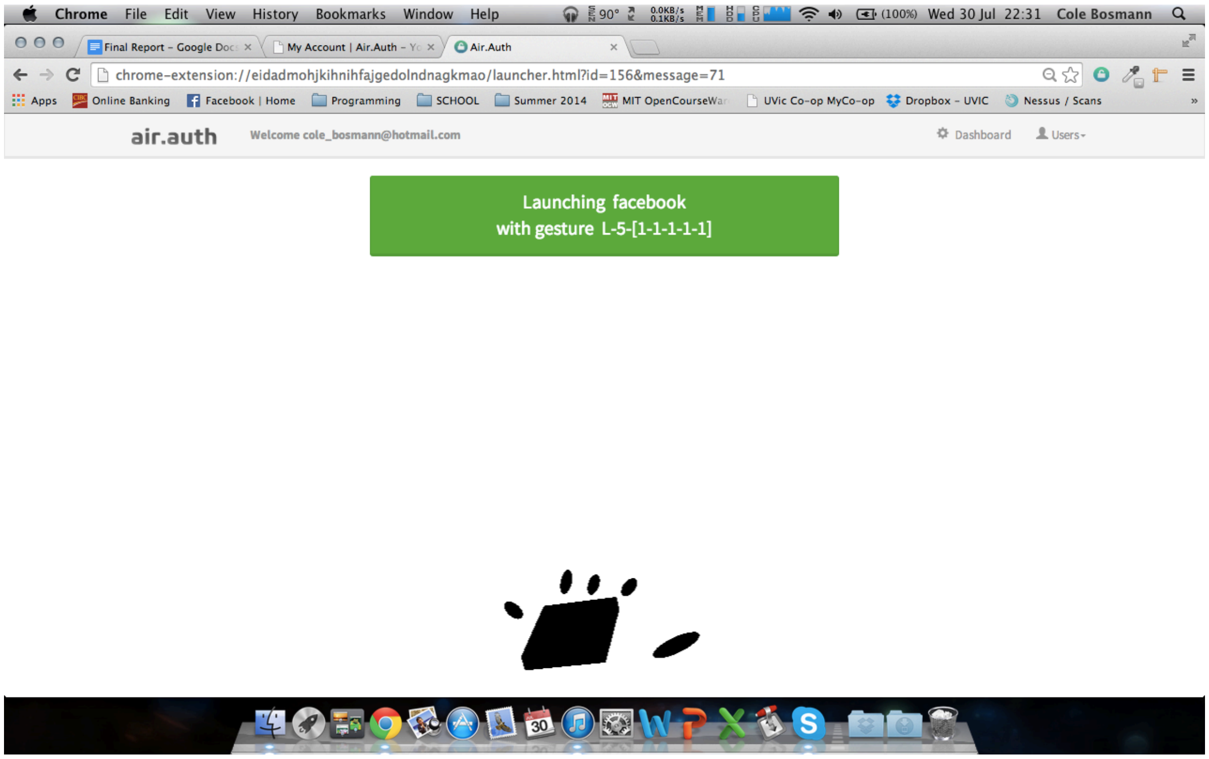
Lower Hand

10%

Could not reliably determine hand. Please scan again.

Figure 14: Hand Scan

Finally, once a user has entered their PIN and scanned their hand, they are ready to launch. The user can accomplish this by performing the pose they previously assigned to the site they wish to launch. This step is shown below in Figure 15, where a user has performed their “Facebook” pose. This action will cause a new tab to be created and directed to Facebook, populated with their Facebook credentials, and logged in.


Chrome File Edit View History Bookmarks Window Help
Wed 30 Jul 22:31 Cole Bosmann
Final Report - Google Doc
My Account | Air.Auth
Air.Auth
chrome-extension://eidadmohjkihnihfajgedolndnagkmao/launcher.html?id=156&message=71
Apps
Online Banking
Facebook | Home
Programming
SCHOOL
Summer 2014
MIT OpenCourseWare
UVic Co-op MyCo-op
Dropbox - UVIC
Nessus / Scans
air.auth
Welcome cole_bosmann@hotmail.com
Dashboard
Users

Launching facebook
with gesture L-5-[1-1-1-1-1]

Figure 15: Launch Pose

4. Leap Motion

The beta version of the Leap Motion api enables the measurement and tracking of individual bones. The beta v2 “Bone” api is available for multiple languages C#, C++, Java, Python, Objective C, and Javascript. For use in the chrome extension, Javascript was selected as the development language.

The beta “Bone” api is the basis for Air.Auth’s biometric hand authentication. It has the ability to measure the width and length of the Distal phalanges, Intermediate phalanges, Proximal phalanges, and Metacarpals for each finger, as well as the dimension of the palm and the ability to differentiate between left and right hands.

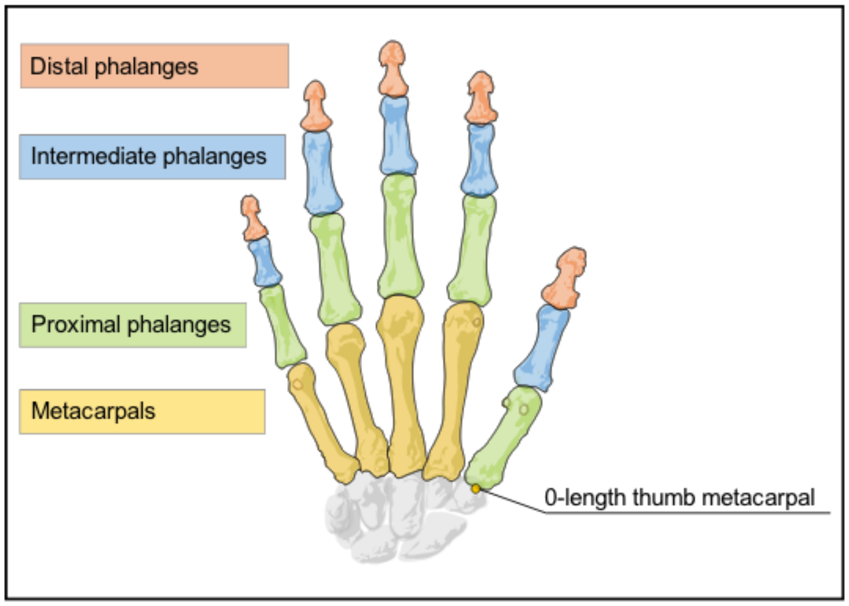


Figure 16: Hand Model

To analyze the accuracy of Leap Motion some initial testing was performed. 50 samples of a users hand dimensions were taken, then used to calculate the variance and standard deviation for each measurement. For example, the standard deviation for the index finger proximal phalange length was calculated to be 0.82952mm, and the standard deviation for the index finger distal phalange length was calculated to be 0.32992mm.

A pattern quickly emerged, smaller standard deviations were seen for measurements closer to the extremities of the hand. Further research revealed that the measurements provided by the Leap Motion originated from the tips of the fingers. Therefore the measurements closer to the extremities were more accurate while the measurements closer to the wrist had more variance.

Leap works by latching onto identifiable features of a user’s hand. When a hand enters the Frame (the area above the Leap Motion Device where it is capable of tracking a hands), Leap locates the finger tips and other noticeable hand features. All the data provided by the Leap device regarding that hand is then derived from these points. Once the Leap device has identified these points, it continues to track them very accurately until the hand completely leaves the frame.

This results in the data being outputted from the Leap device to remain incredibly stable once it has located and begun tracking a hand, but vary significantly (for the purposes of biometric identification) between instances of the same hand entering and leaving the Frame. This variance is due to Leap latching onto slightly different points, resulting in all derived measurements from these points varying accordingly.

To account for the varying measurements of a user's hand between scans (and instances of a user's hand entering and leaving the Leap Frame) multiple scans could be performed, then an average for each desired measurement calculated. Significant effort was put into emulating a user's hand entering and leaving the Frame by forcing the Leap device to forget the hand it was currently tracking, requiring it to relocate the points it latches onto. If this could be performed quickly, a large sample set could be gathered then averaged resulting in an accurate measurement of a users hand.

After significant research and discussion with the both Leap Motion support and development teams, it was concluded that this was not currently possible with the Javascript API, as it did not connect directly to the Leap device but rather through a web socket. Had the project been written in a language that provide more low level access such as C, or C++ the result may be different although more analysis would be required.

Having the user remove then replace their hand from the Frame ten or more times to gather sufficient data is cumbersome and defeats the purpose of the Air.Auth. As emulating the hand entering and leaving the Frame could not be achieved and the majority of the dimension and measurements provided by Leap were insufficiently accurate to be of use for authentication, a different approach was needed which would hopefully be independent of the variance of measurements.

As the variance of the hand measurements tended to simply shift all measurements up or down (i.e affected all measurements evently) two alternative approaches to identifying a user's hand were explored. Firstly the difference between various bone lengths were explored followed by the angles made between each joint.

The first alternate approach was to take each bone measurement then calculate the magnitude of the difference between it and all other bones. This would be independent of the variance output from Leap due to the nature of the calculation. For example if the variance in the Index Finger Distal Phalange (D_3 Figure 17) measurement was 0.3 mm, it would likewise be the same for the Ring Finger Distal Phalange (D_9 Figure 17). Calculating the magnitude of the difference between the two:
$|(D_3 + \text{Variance}) - (D_9 + \text{Variance})| = |D_3-D_9|$

We can see that the variance has no effect on the result. This calculation was performed for each bone pair (e.g. $|D_3-D_{13}|$, etc) for a total of 13+12+11+...+1 = 91 results (when only considering the lengths of each bone and not the width or other measurements).

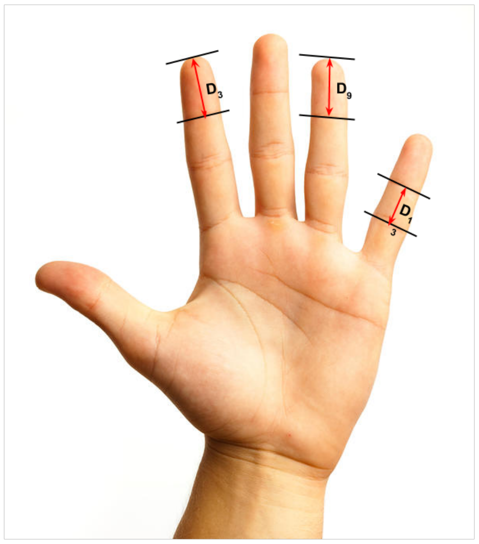


Figure 17: Bone length differences

This appeared to have great potential, however after testing on numerous different hands it became apparent that most hands tend to have very similar ratios. Even for hands which appear to be quite different (a very small hand and a large hand) the results from the magnitude calculations were extremely comparable. These results were therefore not useful in authentication as almost all hands returned similar values.

The second approach taken after it became apparent that the magnitude of the bone length differences were not sufficiently unique to each user, was to calculate the angles between joints and finger tips.

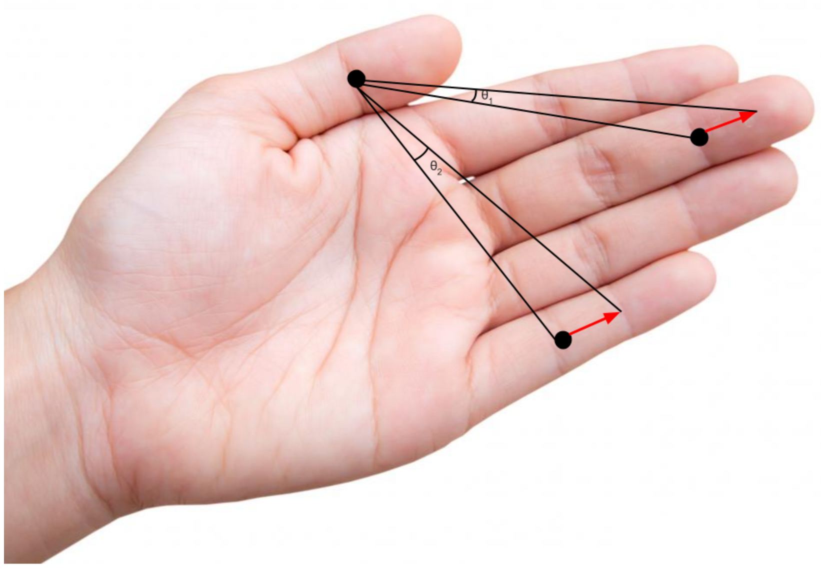


Figure 18: Angles between fingertips and joints

This also appeared to have great potential, but was quickly determined to be infeasible. In order to ensure the same angle was calculated between different scans, a user's hand would need to be closed as in Figure 18 compared to Figure 17. If the angles were calculated when the user's hand was open, as in Figure 17, there would be no guarantee that a user's hand would be in the same position on subsequent scans, resulting in erroneous data. Unfortunately the Leap device has significant difficulty tracking fingers and joints when there is no clear separation. Obtaining accurate position of joints and finger tips to extract vectors then subsequently calculate the angles proved impractical as the Leap could not reliably track a user's hand in the closed arrangement (seen in Figure 18).

This left few options for the biometric authentication of a users hand using purely the measurements provided by the Leap Motion. In order to get a functional biometric authentication running, the authentication measurements were reduce to those that were near the tips of fingers and determined to be sufficiently accurate. These include the length and width of the Distal Phalanges and Intermediate Phalanges for all fingers (see Figure 16).

While these measurements are fairly accurate and stable between scans, they could only determine whether your hand is likely yours, though without certainty. To overcome this, and ensure the authentication is certain the hand presented belongs to the user in question, a second authentication mechanism was added.

Many methods were explored for the secondary authentication method including:

- Having the user draw their signature in the air
- Having the user perform a predefined authentication gesture that could then be analysed
- Having the user perform their own defined gesture for authentication
- Having the user perform a series of poses defined by Air.Auth
- Having the user perform a series of poses defined by them for authentication.

The open source Leap Motion gesture and poses recognition library, Leap Trainer.js [1], was utilised to explore the various techniques mentioned above. The above techniques for authentication were analysed for accuracy, feasibility, flow, and simplicity. The optimal option was having the user perform a series of selected but predefined poses. This amounts to a PIN of defined poses selected by the users. To keep the flow of the application smooth a PIN comprising of four poses was chosen.

Poses that could accurately be determined by Leap and the gesture library were selected resulting in 8 unique poses for both the left and right hand. The user was able to select a combination of any of these poses resulting in a keyspace of 16^4 = 65536 combinations. As this is larger than the minimum keyspace for both credit and bank cards (PIN of 4 numeric digits, 10^4 = 10000), this authentication technique is satisfactory in proving the identify a user. By combining the PIN authentication with the biometric hand authentication, the security of a user's Air.Auth account and therefore their various username and passwords is ensured.

While the authentication of a user based purely on their biometric hand data from a Leap Motion was not successful, a few of the limitations encountered are expected to be removed in the future. The api used with the Leap Motion is currently only in beta, and while it is expected to improve and provide more accurate data there are also plans to allow direct access to the cameras within the Leap device. If this access was provided the techniques utilized in "Robust Hand Geometry Measurements for Person Identification Using Active Appearance Models" [2] could be implemented. As these techniques have already been proven to accurately authenticate a hand, pure biometric authentication using a Leap Motion should be possible, with a future version of the api.

5. API

The API was built using a number of backend components listed in the system architecture section. It was built using Node.js with express.js as the framework and written purely in javascript. It was implemented for supporting the Air.Auth chrome extension. Its function is to accept HTTP requests from the chrome extension and respond with relevant data. The API also communicates with the two databases (MySQL and Redis) to obtain data for manipulation. It supports both GET and POST methods for requests, which allows for a lot of flexibility when it comes to incoming requests.

An API has HTTP endpoints that the client application, in this case the Air.Auth Chrome extension, can communicate with, submitting some data in order to obtain some response data. The API endpoints correspond to an express.js route. The routes direct an incoming request to an express.js function that takes query parameters and responds with some data. The data can be in a number of formats such as JSON, Plain Text or XML. For this project JSON was chosen to be the format for the response from the API. The various routes in Air.Compute are listed and discussed below.

5.1 Index Route

Request Format:
GET /

Express.js Function:
app.get('/', routes.index);

This is the index route or endpoint that displays the default landing page for Air.Auth. This page contains information about the chrome extension and documentation.

5.2 Sign Up Route

Request Format:
GET
/api/user/signup?name={user.name}&email={user.email}&password={user.password}&pin_select_1={user.pin.first.digit}&pin_select_2={user.pin.second.digit}&pin_select_3={user.pin.third.digit}&pin_select_4={user.pin.fourth.digit}

Express.js Function:
app.get('/api/user/signup', routes.signup);

The signup route takes in user's name, email, plain text password and the PIN code from the query parameters. An initial database lookup is done in order to find out whether a user exists with the same email address as the one requesting to signup, if so then the route returns a 500 error code along with a message saying that the user already exists in the system. Otherwise, the user's plain text password and PIN code is hashed using an one way hashing function. A random hash is also generated for password reset purposes. All the data generated is then stored in the application's session store (ie. Redis) and a 200 success code is passed along with the user's email.

5.3 Login Route

Request Format:
POST /api/user/login

Data:
email - User's email
password - User's password
user_ids - An array of user ids that are linked to the computer
auth_ids - An array of authentication ids that are linked to the users

Express.js Function:
app.get('/api/user/signup', routes.signup);

The login route takes in user's email, password, array of user ids and auth ids that are linked to the computer making the request. The user's plain text password is then one way hashed and a database query is done to validate whether the user is a valid Air.Auth user. If the user cannot be found in the database then a 500 error code is returned with a message that the user is not valid. If the user is found in the database, then all the user details are obtained and the user's id and PIN are stored. Next, a random authentication id and key pair is generated and stored in Redis. The user's plain text password is then encrypted using the authentication key. The encrypted password that is generated can be decrypted on the client side using the authentication key. The user-agent and the ip address of the user is then obtained and associated with the user's id. This creates a valid Air.Auth session for that user.

Another key piece of logic that comes into play is when the user that is attempting to login has the same PIN as a user that is already present on that computer. The user_ids array is used to query the database and the users that have the same key as the user that is trying to login are logged out by invalidating their Air.Auth session in Redis.

After all the above operations are successfully performed, a response is generated that contains a 200 success code with the user's encrypted key, authentication id, user email, user id and an array of users that have the same PIN as the user that is trying to login.

5.4 Hand Compute Route

Request Format:
POST /api/hand/compute

Data:
user_hand - User's hand data object

Express.js Function:
app.post('/api/hand/compute', routes.hand_compute);

Session Data:
user_email - User's emails from session
name - User's Name from session
encoded_password - User's encoded password from session
encoded_pin - User's encoded pin from session
password_hash - User's random password hash

The hand compute route obtains user hand data from the POST request and the remainder of user's data from the session storage, which was stored in the sign up route. The user is then registered by inserting the user data that was obtained from the session into the user table in MariaDB.

The user hand data object obtained from the request contains five samples of data that are to be inserted into the database. There are two inserts that take place in this route that involve user hand data. First, the insert into the hand_data_average table and another in the hand_data_raw table. The first insert requires that hand data object to be parsed through and averaged. After the average has been computed, the data object is then prepared for the database insert into the hand_data_average table. Once this insert is complete the five raw readings are inserted into the hand_data_raw table.

After the user hand data is inserted into the database, user's email and id is stored in session. A random user token is then generated and stored in session. This is done in order for the application to recognize the user is valid. This helps in logging the user in after the signup process is complete. Finally, a response with a 200 success code and a boolean value true is sent back to the client computer.

5.5 Hand Authentication Route

Request Format:
POST /api/hand/authenticate

Data:
user_hand - User’s hand data object

Express.js Function:
app.post('/api/hand/authenticate', routes.hand_authenticate);

Session Data:
user_id - User’s id from session

In the hand authentication route, the user’s id is used to get all hand measurements that were stored in the database in the hand compute route. A database query is done that gathers all the hand data belonging to a user id from hand_data_average table, this returns a javascript array. This array is then compared with the user hand data object that was obtained from the POST data with a help of a special utility class called authentication.js. The class has a function called “auth” that takes in incoming user hand data object, database user hand data and various tolerance values for comparison. The tolerance values that are taken into consideration are as follows:

- Distal Length Tolerance
- Distal Width Tolerance
- Medial Length Tolerance
- Media Width Tolerance
- Palm Tolerance

These tolerance values are used to compare the difference in the incoming user hand data readings with respect to the user hand data readings obtained from the database. If the difference between the incoming user hand data and the database readings are within the tolerance values, then the incoming user is considered valid. There are a lot of tests that are done in a similar fashion for all of the hand dimensions that are part of the user hand data. The end product of all the comparisons is a score that is out of 72. This score is then turned into a percentage and passed back to the route by the “auth” function.

The percentage obtained from the “auth” function is then used to find out whether a user’s hand is valid or not. If the percentage is greater than 75 then the user’s hand data is valid, otherwise the user’s hand data is invalid and doesn’t match the user’s hand data obtained from the database. After the hand data is validated then a random hand token is generated and associated with the user id and stored in Redis with a very short time to live. This is done to provide easy access for the user that is using Air.Auth to launch websites. This allows the user to only enter the pin and launch sites without going through the hand authentication step. This hand token is valid in Redis for 10 minutes. The response contains a 200 success code, user id and a hand token that was generated earlier.

5.6 Validate Pin Route

Request Format:
POST /api/pin/authenticate

Data:
pin - User's pin
user_ids - Array of user ids
hand_tokens - Array of hand tokens

Express.js Function:
app.post('/api/pin/authenticate', routes.pin_authenticate);

The pin that is obtained from the POST request is hashed. The hashed pin and the user ids is used to look up user data from the users table. The idea is to find the user with the same pin as the one provided in the request. This search is narrowed down from the entire database by only requiring to lookup users defined by the user ids array provided. Using the user data obtained from the database query, a lookup is done in Redis to find out whether that user has a valid hand token. If a valid hand token is found, a response with 200 success code, user id and valid hand token is sent back. Otherwise a response with 200 success code, user id and empty token is sent back.

5.6 Session Get Route

Request Format:
POST /api/session/get

Data:
auth_ids - Array of authentication ids
ids - Array of user ids

Express.js Function:
app.post('/api/session/get', routes.session_get);

The array of auth ids are looked up in Redis and the auth ids that are not found in Redis are returned back as invalid ids. The response is a 200 success code and the array of invalid ids.

5.7 Get Auth Key Route

Request Format:
POST /api/key/get

Data:
a_id - User’s authentication id
user_id - User’s id

Express.js Function:
app.post('/api/key/get', routes.auth_key_get);

In this route, the authentication id from the POST request is used to find if the id is valid in Redis. If the authentication id cannot be found, then the response is 500 error code with an invalid id message. If the authentication id is valid then a hand token is generated and stored in Redis with a time to live of 10 minutes. The result from the authentication id lookup is used to obtain the authentication key. The response is a 200 success code, authentication key and hand token.

5.8 Launch Route

Request Format:
POST /api/user/launch

Data:
pose_name - The name of the pose

Express.js Function:
app.post('/api/user/launch', routes.launch);

Session Data:
user_id - User’s id from session

The pose name is obtained from the POST request and the user id is obtained from the session. A complex data query is then done to obtain all site data for the given user id and pose from the database. If the there is no website that corresponds to the pose that was provided, the response is a 404 not found error code and the user id. If a website is found then the response is a 200 success code, all the details associated with the website, and a “found a site” message.

6. Database

6.1 MySQL

Air.Auth utilizes MySQL to store persistent data. This data is broken into 6 tables, the first being the “user” table. The user table is described below in Table 1. It is in this table that all information gathered through the first registration step is stored. All sensitive data like passwords and PINs are one-way hashed to protect users. Also, upon signup, a random hash is created for each user to be used for password recovery. Once a user requests a password recovery, a link containing this hash is sent in an email.

Column Name	Column Type	Description	Example Entry
id	smallint(11)	Primary Key, Auto-Incremented	156
name	varchar(255)	Null Allowed	Cole Bosmann
email	varchar(255)		cole_bosmann@hotmail.com
password	varchar(255)	One-way Hash	d90c8c4b8e8c4af4ddf3dde5ab47b6231e7c9078ced539846be25b937cfc437e
recovery_hash	varchar(255)	Random Hash	5e18aa9d9ddf6cbbe4f4b0a73df444641932ec93
pin	varchar(255)	One-way Hash	5cf6ef7a5fb3f1a257209520f8c17023d6c7068a6ef59709d4f79a27bd387d5c

Table 1: User Table

The second table in the MySQL DB is the pose table. This table contains all of the valid poses available to Air.Auth users. This table is summarized below in Table 2. All Air.Auth sites dynamically pull from this table, therefore in order to add a new pose, an admin must simply insert into this table and it will be made instantly available to all users.

Column Name	Column Type	Description	Example Entry
id	smallint(11)	Primary Key, Auto-Incremented	78
name	varchar(255)		R-1-[0-1-0-0-0]
json	text	Json that can be interpreted by Leap to describe a pose	{example: json}

Table 2: Pose Table

The third table in the MySQL DB is the site table. The site table contains all of the information for saved user sites. Relevant information includes the site username/password, which user this site belongs to, and the assigned launching pose. In order to mitigate against errors, both the user_id and pose_id fields are foreign keys. This means that users and assigned poses must be present in the user and pose table to be valid in the site table. This table is described below in Table 3.

Column Name	Column Type	Description	Example Entry
id	smallint(11)	Primary Key, Auto-Incremented	122
user_id	smallint(11)	Foreign Key	156
name	varchar(255)		linkedin
url	varchar(255)		http://linkedin.com
username	varchar(255)	Encrypted Object	{"ct":"MSAvAiZGr7WVkb4YYAgUCQ==","iv":"b94298be1e1af8f1c32b8f7ea962b428","s":"6fb106846222608a"}
password	varchar(255)	Encrypted Object	{"ct":"ziqqprRu3x5nyguMhG46Rg==","iv":"889711b2589cff5043854131e1ad7862","s":"5e760135358c8a2c"}
pose_id	smallint(11)	Foreign Key	74

Table 3: Site Table

The fourth table in the Air.Auth database is the pose_assigned table. This very simple table is used to link users with their sites, and sites with their assigned poses. In this table, all fields are foreign keys, ensuring that all relationships created between database entities are legitimate. This table is shown below in Table 4.

Column Name	Column Type	Description	Example Entry
user_id	smallint(11)	Foreign Key	156
pose_id	smallint(11)	Foreign Key	73
site_id	smallint(11)	Foreign Key	121

Table 4: Pose Assigned Table

The two final Air.Auth tables are used to store the biometric information obtained from hand scans. The first table contains raw dimension, with each row belonging to a specific user and representing the average values of one scan. The second table contains a rolling average of a user's average hand dimensions. Each time a user successfully scans their hand, a row is added to the raw table, and a new average is calculated and updated in the average table. Only the past 100 scans are included in the average, which helps compensate for gradual changes a user's hand may undergo (weight loss, weight gain, etc…).

6.2 Redis

Redis is known for its large variety of datatypes such as keys, strings, hashes, lists, sets etc. While building the express.js application for Air.Auth, Redis was used to do fast lookups for valid user data. This data included authentication id and key pairs, user's session data, user valid hand tokens and application session data. The format and the data type for the data stored in Redis is shown in the table below.

Data Access Name	Data Type	Data Structure
{authentication_id}	STRING	"{user_id}::::{authentication_key}"
userid::{user_id}	SET	"{authentication_id}::{user_agent}::{ip_address}"
hand_token_user_id::{user_id}	STRING	"{random_hand_token}"

Table 5: Redis Datastore for Air.Auth

The data that is stored in Redis has to be accessed using the Redis client for Node.js. The Redis client supports almost all types of commands that can be issues to Redis. The commands were issued to access the data from Redis for Air.Auth are shown in the table below.

Data Type	CLI Command	Client Function	Description
STRING	GET	db.get(key, callback function);	Get string value for a key
STRING	SETEX	db.setex(key, ttl, callback function);	Set a value in Redis for a key with a specific time to live in seconds
STRING	TTL	db.ttl(key, callback function);	Get the time to live for a key
SET	SADD	db.sadd(setName, memberData, callback function);	Add a member to a set. If set does not exist then this command creates a set
SET	SMEMBERS	db.smembers(setName, callback function);	Get all the members in a set
KEYS	DEL	db.del(key, callback function);	Delete any key with the keyname that is provided

Table 6: Redis CLI and Node.js Client Commands to Access Data

The SETEX, TTL and the GET operations are of complexity 1 (ie. O(1)). The SADD and SMEMBERS operations are of complexity O(N) where N is the number of members to add in a set.

The challenge about using Redis is that the data stored in it is mostly in string format, and since it is a NoSQL database the relationship between data objects is not maintained, hence there is a need to have additional logic in the code. The use of delimiters is essential and necessary for storing data into Redis. However, there is a huge performance gain in terms of lookups since most of the lookups are O(1) complexity, that in turn helps makes the API very responsive.

7. Security

7.1 User Validation

In order to manage their account and access/add/edit sites, users must authenticate using their master password. As a result, Air.Auth must store some form of a user's master password in order to authenticate. Storing plain text passwords constitutes a security risk, therefore Air.Auth hashes a user's password upon entry to the system. This hashing changes the password from a recognizable word or phrase to a seemingly random combination of letters, numbers, and symbols. This hashing is also irreversible, meaning if the system is compromised, attackers will not be able to reverse the hash to obtain the plaintext master passwords.

Although this hashing is not reversible, it is deterministic, meaning that any given input will yield the same output from the hashing algorithm. This is how Air.Auth authenticates, upon registration it stores a hashed version of a user's master password. On subsequent logins, Air.Auth again hashes the attempted login password, and compares with the stored hash password. An example of this scheme is shown below in Figure 19.

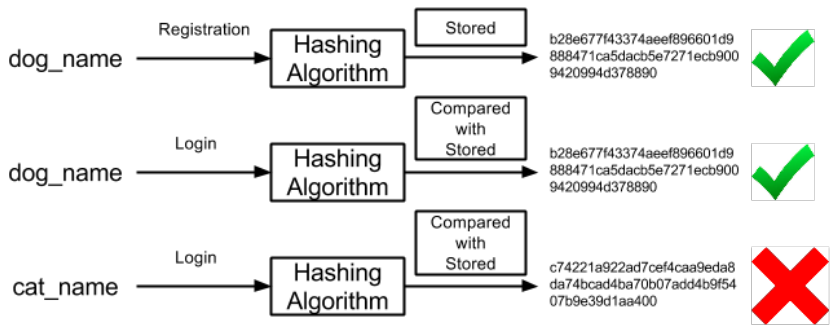

Registration
dog_name
Hashing Algorithm
Stored
b28e677f43374aeef896601d9888471ca5dacb5e7271ecb9009420994d378890
Login
dog_name
Hashing Algorithm
Compared with Stored
b28e677f43374aeef896601d9888471ca5dacb5e7271ecb9009420994d378890
Login
cat_name
Hashing Algorithm
Compared with Stored
c74221a922ad7cef4caa9eda8da74bcad4ba70b07add4b9f5407b9e39d1aa400

Figure 19: Password Hashing

The Air.Auth PIN is also stored and authenticated against in a similar fashion. Because the PIN is a series of JSON objects, they can be converted to strings and then hashed. Air.Auth utilizes the SHA-256 hashing algorithm. This is an algorithm designed by the NSA, and is the current industry standard for hashing sensitive information.

7.2 Site Password Encryption/Decryption

A main challenge when creating a password manager is keeping the stored passwords secure. In case the system is compromised, it is important that passwords are not stored in plaintext form. In the previous section, hashing solved this problem. However, with the site passwords it is critical that the transformation be reversible, as the plaintext is required to submit to the sites/services. For the desired level of security, it became necessary to implement encryption of the site passwords. Encryption represents a two-way transformation, allowing for a plaintext password to be temporarily scrambled (encrypted), and then reassembled (decrypted) using a unique key. This key was chosen to be the user's most unique and secure attribute, their master password.

Although encryption with the master password solved the original problem, it created a new one. Under this scheme users would be forced to supply their master password every time they needed access to an encrypted site password. This is a huge inconvenience and was deemed unacceptable. Instead, the fact that a user supplies their master password when linking their account was utilized. This master password was made persistent through a local cookie on the user's computer. This way, whenever decryption was required, this master password could be retrieved from the cookie, and used as the key.

This solution led to yet again another problem, was it acceptable to keep a plaintext master password in a local cookie? Because Air.Auth was designed to be used on shared computers, this was once again deemed unacceptable. The master password was then encrypted, using an authentication key. This auth key was paired with an auth id. These pairs were created each time a user linked their account to a computer, at the same time that the local cookie is set. The local cookie now contains the auth id, and the encrypted master password. Once a site password is required, the auth id is sent to the server, the auth key is returned, the master password is decrypted, which then allows the site password to be decrypted.

Although this security scheme appears convoluted, it was necessary to secure the Air.Auth system. This scheme ensures that keys to encryption are never stored in the same place, and that interaction between the client and server is required in all decryption steps. This way, for the system to be compromised, an attacker must compromise both the server and the client simultaneously. The encryption algorithm used was AES-256, which was developed by NIST, and is commonly used in industry. To further elaborate, two main use cases for encryption and decryption are shown below in Figure 20 and Figure 21.

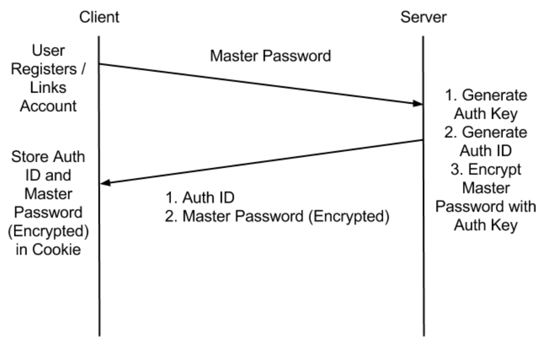

Client
Server
User Registers / Links Account
Master Password
1. Generate Auth Key
2. Generate Auth ID
3. Encrypt Master Password with Auth Key
Store Auth ID and Master Password (Encrypted) in Cookie
1. Auth ID
2. Master Password (Encrypted)

Figure 20: Account Creation / Account Linking

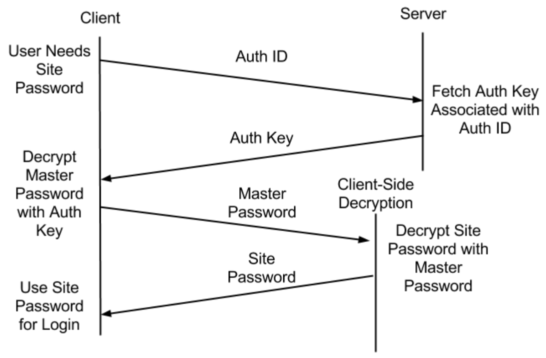

Client
Server
User Needs Site Password
Auth ID
Fetch Auth Key Associated with Auth ID
Auth Key
Decrypt Master Password with Auth Key
Master Password
Client-Side Decryption
Decrypt Site Password with Master Password
Site Password
Use Site Password for Login

Figure 21: Fetch Site Password

8. Session and Cookie Management

Session and cookie management are an integral part of Air.Auth. There are three types of sessions that are used within Air.Auth, each serving different purposes. They are as follows:

- Air.Auth User Session
- Air.Auth Account Session
- Air.Auth Hand Token Session

8.1 Air.Auth User Session Management

As mentioned earlier, when a user logs into a computer or links their account with a another computer, a pair of authentication hashes are created, an authentication ID, and an authentication key. The authentication ID is passed back to the extension and stored in a cookie, while the authentication key is stored in a Redis string with authentication ID as the index for fast lookups. The ID and key are linked pairs, which is essential in securing the username and passwords for the various user websites, but they also to serve in validating a user's sessions.

As the authentication key has a 7 day TTL, every time a user interacts with Air.Auth, the validity of all user sessions connected to the computer are checked. All of the cookies belonging to Air.Auth are inspected and the authentication ID extracted. These IDs are sent to the remote server (API) to check for matching and active authentication keys. If any of the users have authentication keys that have expired the corresponding cookie (i.e the one with the linked authentication ID) is then removed from the computer. As the cookie stored locally is the only place where the decryption key for user website credentials is stored, and as the credential decryption key itself requires the authentication key to be decrypted (see section 7.2), this ensures that no invalid sessions are able to access any user information. Once the authentication key is expired, the local cookie is invalidated and is useless to all people including attackers. This is because the credential decryption key itself will not be able to be decrypted, because it's key (the authentication key) will have been deleted.

8.2 Air.Auth Hand Token Session

A second independent session was implemented to facilitate the launching of multiple websites once a user is successfully authenticated. The first time a user authenticates they must enter their PIN then scan their hand. If they successfully validate both they may then perform a pose and launch their site. However if the wrong pose was performed and an undesired page launched, or the user simply wishes to launch another page shortly after the first, they would be required to perform both the PIN and scan again.

To eliminate some unnecessary steps in launching a website, a short lived "Hand Token" session was created. Once the user successfully enters their PIN and scans their hand a short lived cookied with a 10 minute TTL is created on the users computer. This cookie contains a Hand Token (a random hash) generated on the remote server and likewise stored in Redis with a 10 minute TTL as well. On subsequent PIN entries, the validity of all Hand Tokens is checked. If the entered PIN matches a valid Hand Token, the user is directed to launching of a site without scanning their hand as we can be fairly certain it is the same user sitting at computer. If a wrong PIN is entered or the Hand Token is expired, then the user must successfully pass both the PIN entry and the hand scan to be able to launch a website.

This secondary session significantly simplifies and speeds up interactions with Air.Auth while maintaining the high level of security we have strived to achieve.

8.3 Air.Auth Account Session Management

Lastly as the account management pages of Air.Auth contain very sensitive data, a third completely separate session was desired to ensure access was only granted to the valid user. The account session for Air.Auth users is purely application based. It is implemented using express.js session store and have a TTL of 10 minutes. Once a user logs in to the account management system, an unique user token is generated that acts as an identifier for a users session. The browser automatically creates a cookie in this case since the session is handled directly by express.js. The session is destroyed each time the user visits the login page and logout pages. This is done to deal with the multiple user scenario when one user tries to access the account of another user not knowing that there might be a user already logged in.

The account session is used to store user's email and user id. This implementation is very efficient since there are no database lookups that need to be done in order to gather user data for the required account management operations.

9. Chrome extension

As our project at its basis is a password manager for your various online accounts, incorporation with a browser was essential. As all modern browsers support some form of extensions (Add-Ons for Firefox, Extensions for Safari and Add-Ons for IE) Air.Auth could have been written for any of them. Eventually the goal is to have Air.Auth function with all major browsers, however due to time constraints, the scope of the project was limited to a single browser. With all members of the team using Chrome as their main browser it was the clear choice to develop for first. Chrome has a very well documented api for developing extensions and a vibrant community. This turned out to be invaluable as we learnt the intricacies of creating Air.Auth and reinforced our view of Chrome being a great platform to develop within.

One of the key dilemmas faced due to the nature of having a remote server backend and a local extension front end was deciding which elements of the application should be where (i.e. local or remote). As an example when a user enters a username and password for their Facebook account, should the encryption of the fields be handled locally or done on the remote server. Practices of well known companies such as Facebook and other extensions were investigated as well as the performance of Leap when hosted remotely vs locally. With the decision to have a separate session solely on the remote server when managing a users account, it was decided that all Leap related elements should be hosted locally to ensure the highest possible performance while all user management related elements would be hosted remotely using the separate sessions to ensure maximum security. There were, however, some exceptions, notably the decryption of a user's saved website username and password (i.e Facebook).

Arguably the only and most important reason for building Air.Auth as an extension rather than simply as a website is due to the Same-Origin Policy. This Policy states that scripts may only run on sites that originate from the same domain. As we require the ability to pass usernames and passwords to various sites, running Air.Auth as a website would conflict with the Same-Origin Policy resulting in not being able to launch other websites and log user in.

The Chrome extension eliminates this issue by giving direct access to various sites through the use of Content Scripts. Content scripts are scripts which can be embeded on any site specified by the extension. Though they are part of the extension they originate from, they are their own entities completely separated from the rest of the extension. To communicate with the content scripts, message can be exchanged using Chrome’s built in message passing. Only one content script was needed for Air.Auth but provided the key ability to pass the username and password to the desired websites, and then subsequently log the user into that site. For the purposes of this project, where to insert the username and password on the various sites was hardcoded into the content script. This is very limiting and can be easily broken if Facebook or Twitter were to change any input id name, however, it successfully proved the concept and allowed us to focus our effort on other more demanding aspects of the project. We plan to implement a more robust solution, which is discussed in Future Work.

10. Future Work

Firstly, while hashing of a user's master password and PIN eliminates the risk of exposing plaintext passwords, it is not a completely secure solution. Should the database be compromised, an attacker is able to feed dictionary words into the hashing algorithm, and compare the results with the list of hashed user passwords. If a match is found, the attacker knows the input that resulted in this match, therefore giving them the password. This is called a rainbow table attack. The main defense against this is to slow down the speed at which the attacker can hash their guesses. This is accomplished through key stretching, where a password is rehashed multiple times (100-10000 times). This increases the time a single guess takes to evaluate, slowing down a rainbow table attack and making it unfeasible. Future versions of Air.Auth will include this feature to mitigate against this attack vector.

Secondly, as the original plan for the project called for the ability to launch websites with gestures (e.g drawing an F to launch Facebook), but could not reliably be accomplished with currently available libraries, it is a functionality we wish to still implement. This may require the creation of our own library for gesture recognition, but we believe with the added accuracy provided by the new v2 beta "Bone" api (and future stable versions), improved third party libraries will become available that could accomplish our desired goals.

Incorporating these improved libraries would provide the possibility of not only launching websites through the use of gestures but also increasing the pose PIN keyspace by augmenting the number of poses that can reliably be detected, further securing the system.

If the gesture recognition libraries do improve significantly the possibility of utilising a user's signature, performed by tracing their finger through the air, could be added to enhance the security measurements or replace one of the current authentication methods.

Thirdly, we would like to expand the scope of sites that can be stored in Air.Auth. Currently this is limited to a few select sites as the ability to pass username and password to the desired sites needs to be hard coded into the extension. We realise that this is suboptimal, but designing a system that can accurately located and interact with all the various login forms across the internet was beyond the scope of this project. This, however, is not something we wish to leave uncorrected. Designing this system will be an interesting challenge and one we look forward to overcoming.

Lastly we wish to boost the abilities of the biometric authentication. We want to authenticate a user based solely on the scanning of their hand. As the v2 "Bone" api is improved, the currently implemented biometric authentication method will also improve. Though these improvements may be sufficient to positively identify a user's hand with sufficient confidence, they will likely continue to need a secondary authentication method to truly prove the identity of the user's hand. To overcome this the image process techniques analysed in Robust Hand Geometry Measurements for Person Identification

Using Active Appearance Models [2] could be implemented once access to the images obtained by the Leap device is given. Though no timeframe for this access has been given by Leap, it is something we will continue to keep our eyes on it as it would be the solution to the biometric authentication issues we have encountered.

This project has had many interesting challenges and obstacles. We feel the final product accurately reflects the work we have put into it, the quality we strived to achieve and functionality we intended to implement. We look forward to improving the project more and eventually releasing the project into the chrome app store.

11. References:

[1] O'Leary, Rob. Leap Trainer.js. Github, 1 Sept. 2014. Web. 20 May 2014. <https://github.com/roboleary/LeapTrainer.js/commits/master>

[2] Gross, R., Li, Y., Sweeney, L., Jiang, X., Xu, W., & Yurovsky, D. (2007). Robust Hand Geometry Measurements for Person Identification Using Active Appearance Models. *IEEE Conference on Biometrics*.