# GeoAuth Design Document

Jacob Okamoto, Andrew From, Jason Carter, Christian Gerrard

April 3, 2013

*The Execution Plan is at the end of this document*

## 1 Introduction

This is the design document for GeoAuth, a location-based authentication service. Its objective is to provide a strong identity verification service using dynamic location data, instead of using traditional, static data such as biometrics (i.e., fingerprint and iris scans), which can be simply integrated into websites' authentication mechanisms as an additional authentication factor.

## 2 Purpose

GeoAuth seeks to provide a more flexible and less sensitive alternative to traditional authentication strengthening techniques employed by websites to validate identity. Due to recent high-profile account compromises at major websites such as Twitter, websites have begun to implement multi-factor authentication schemes, which follow the principle of "something you have and something you know."

### 2.1 Existing solutions

Traditional multi-factor authentication systems have traditionally relied upon keyfob-style systems, such as those from RSA or SafeNet. These have the disadvantage of requiring that users carry with them the device that produces their authentication codes. They have also been subjected recently to increased scrutiny due to security breaches at RSA, which may have potentially exposed all of RSA two-factor customers' systems.

Recently, some sites have begun implementation of their own multi-factor authentication schemes, often taking advantage of the ubiquity of cellular phones. The largest-scale example of this trend is Google, whose multi-factor authentication scheme sends six-digit numeric codes to a user's phone when they login to their Google account from an unknown device. They are then presented with the option of "remembering" the device, which drops the requirement that they type in a code when logging in from that device.

### 2.2 GeoAuth's solution

GeoAuth seeks to provide an alternative means of providing multi-factor authentication. Location-aware smartphones are quickly becoming commonplace; combining GPS and Wi-Fi scanning, they can pinpoint their location extremely accurately. This highly accurate location information is what GeoAuth seeks to exploit. By collecting a user's location over time, and combining that with user-created location names, it is possible to construct reasonable challenges which would require an attacker to know both the user's location at the time the challenge was created and how the user named that location.

#### 2.2.1 Advantages over existing solutions

GeoAuth's advantages over existing solutions differ for each existing solution. Compared against keyfob-based systems, it removes the requirement that the user carry the relatively fragile keyfob around with them (not carrying the keyfob defeats the purpose of "something you have"). Compared against Google-like authentication schemes, GeoAuth has the user send authentication information (location data) to the GeoAuth service, instead of sending authentication information to the user, helping reduce the potential impact if a user's smartphone is stolen.

#### 2.2.2 Disadvantages over existing solutions

GeoAuth's primary disadvantages are its reliance on a relatively diverse set of location data for effective generation of queries, and the potential ability for attackers to "guess" answers to queries. The first may be solvable through examination of the kinds of queries which can be produced, which may yield more effective techniques for secure query generation. The second is harder to prevent, as it can be caused by users doing two things: first, following almost invariant location patterns over time (i.e., leaving for work at 8 A.M., driving to pickup children at 3:30 P.M., and going home for the remainder of the day, every weekday); second, choosing extremely predictable location names, such as "home," "work," or "school". The former may be mitigated by more extensive location analysis looking for anomalous behavior; the latter by encouraging varied location naming.

## 3 Use Case

GeoAuth is being designed to be used as a secondary authentication factor in multi-factor authentication systems; that is, it is to be used to strengthen identifying information such as passwords or biometrics. Because location, while private in a sense, is necessarily to a degree public information (that is, your location is often known to other people, such as family, friends, or coworkers), we believe that it cannot be used as a standalone authentication factor. Websites wishing to use GeoAuth will use GeoAuth's website API (detailed in section **TODO**) to request location challenges which may be presented to users as part of their authentication workflow, and to validate the responses users provide. They will also be able to redirect users to register their devices with GeoAuth and connect them to the service.

# 4 Architecture

GeoAuth consists of four primary components. The first is the mobile client, which runs on users' smartphones and uploads location and naming data to the GeoAuth service. The second is the device API server, which services mobile clients' upload requests, performing verification of the device's registration status and storing that location information to the server. The third is the website API and interface server, which is responsible for receiving websites' requests for challenges, producing the challenge/response pages, and returning the user to the website when they have successfully (or unsuccessfully) authenticated. The fourth is the interface for handling user registrations, device management, API access, and other administrative functions.

## 4.1 Mobile Client

The mobile client is the most important part of GeoAuth. It is responsible for collecting all of the location data and naming information which GeoAuth uses to construct challenges.

Its primary task, location data collection, will be achieved through the use of a long-running background service, which will periodically request the device's current location and send it to GeoAuth. The secondary task, location naming, will be performed by notifying the user that they should name a location if they have spent a substantial amount of time in the same place, and it will be aware of existing named locations so as to prevent duplicate or overlapping regions.

The exact implementation will be platform specific, and the fullness of the implementation will take into account each platform's limitations. Clients will be produced for all major platforms (Google Android, Apple iOS, and Microsoft Windows Phone).

### 4.1.1 Apple iOS Client

The Apple iOS client (authored by Jacob Okamoto) implements the core background location tracking capabilities of the client and the device registration interface. On launch, it presents an interface for registering the device with the server; if unregistered, it will use a testing key until registration is completed. The device identifier returned by the server during registration is stored using the `NSUserDefaults` capabilities of iOS, and the stored key is used for any subsequent calls. Location updates are performed when movements greater than 100 meters are detected; the resolution of the location measurements is set to the highest possible level to allow for Wi-Fi based positioning. Future changes may include using the iOS Significant Location Change service to decrease power consumption, and integration of the configuration interface into the iOS settings menu.

Source code is available at <*https://github.umn.edu/CSCI5221-G06/geoauth-client-ios*>.

### 4.1.2 Android Client

There are two Android clients (one authored by Christian Gerrard, the other by Jason Carter).

**Gerrard Client:** This Android client implements the same set of functionality as the iOS client. It also implements development functionality to request device keys to register the device (this functionality would be handled server side in production). Due to issues with Android's background services engine, it will send placeholder values when backgrounded.

Source code is available at <*https://github.umn.edu/CSCI5221-G06/geoauth-client-android*>.

**Carter Client:** This Android client supports interfacing with the device API; however, it has not yet implemented any further functionality at the present.

#### 4.1.3 Windows Phone Client

The Windows Phone client was written by Andrew From. A local database stores location region information. Currently the database is populated whenever a region is successfully added, and is checked to prevent duplicate location requests of the same name. A future enhancement would be to query the local database and display if the user is in defined region. A UI to manage regions could also be created.

On location change it checks if the GPS distance (3200 meters from last check in) or the GPS last check in time (15 minutes since check in) have been exceeded, if so it will automatically checkin to the server. There is currently no UI for a user to modify these settings, but it could be created in the future.

Source code is available at `<https://github.umn.edu/CSCI5221-G06/geoauth-client-wp7>`.

### 4.2 Device API server

The device API server is responsible for receiving location updates and location names from mobile clients and storing that data in GeoAuth's backend database. It is also responsible for servicing device registration requests. This API will be used over HTTP and will conform to REST concepts; the details of the API are located in the "Device API" document.

### 4.3 Website API server

The website API server is responsible for receiving websites' requests for authentication challenges, providing challenge tokens which websites can use to redirect their users to the challenge presentation system, presenting the challenge and validating its response, and sending the outcome of the challenge back to the website. It is also responsible for providing websites with appropriate binding information to connect the website's identity system to GeoAuth's; the details of the API are located in the "Service" document.

### 4.4 Data View server

The data view server provides read-only access to the data stored on the server for development purposes. In future it will support management and user self-service functionality.

### 4.5 Authentication workflow

The following describes the authentication process using GeoAuth:

1. The website requiring authentication sends a request to GeoAuth's website API for a challenge request identifier and a challenge response identifier.

2. The website sends its API key and the GeoAuth identity it is authenticating, and receives a challenge token from the GeoAuth server
3. The website presents the challenge to the user and collects their response
4. The website sends the response value and the challenge token to the GeoAuth server
5. The website receives a `200` with the content "success" or a `403` with the content "failure"
6. The website handles a success or failure.

Note that the website handling is not explicit. Because of the nature of GeoAuth's challenges, a website may choose to use a failed validation as simply a "this is suspicious" flag, and send notification to the user (similar to Facebook's device identification system); or it may choose to deny access.

### 4.6 Authentication challenges

Authentication challenges for the initial prototype will be generated in the form "where were you at {time} on {day}, {date}." The answers to these responses will be based on the named locations provided by the user. While these are relatively simplistic challenges, constructing more complex challenges will require data analysis to determine what sorts of challenges are possible.

### 4.7 Platforms & technologies

GeoAuth's APIs are based on HTTP with REST conventions. This greatly reduces the complexity of implementation, especially with regards to security, since secure HTTP communication over TLS is fully supported on all mobile and server platforms which GeoAuth will use. It also eliminates potential issues designing communications protocols which might otherwise arise using binary protocols over TCP or UDP.

Mobile platforms will use the appropriate technologies for each platform: the Android mobile client will be written in Java; the iPhone client will be written in Objective-C; and the Windows Phone 7 client will be written in C#. Appropriate libraries will be used as necessary to facilitate well-written code.

The servers will be written in Python on the Flask framework. This provides a minimalistic but powerful platform to use as much shared code as possible between the three server side components. A suite of tests will be provided to validate the functionality of the server side, and a test-mode server with fixed data will be provided to allow for simple mobile client testing.

## 5 API references

API references for both the device and website APIs are provided with this document. They are not quite complete as the exact scope of the APIs for the prototype is still in flux.

# 6 Execution plan

## 6.1 Functional objectives

We anticipate completion of a functional prototype to be well within our capabilities. This prototype, at the very least, will implement the core functionalities of:

- Device location updates and location naming, with interfaces to each: **COMPLETE**
- Persistent storage of location updates and processing of relationships between location updates and named locations: **PARTIAL**
- Generation of appropriate authentication challenges: **COMPLETE**
- Fully functional website authentication flow implemented in the website API and demonstrated with a simple test website: **PARTIAL**

### 6.1.1 Completion Notes

Device location update functionality is complete across all platforms. Region support is as complete as possible, but will require further exploration of methods for determining when a new region should be created.

Persistent storage of updates and regions is fully supported. Relationship processing will also require further exploration (research topic).

Generation of authentication challenges is complete at a basic level.

Authentication flow is implemented in the API and demonstrated in the test framework. There is no demo website as of yet.

## 6.2 Division of work

- Andrew From implemented the Windows Phone 7 client
- Christian Gerrard implemented the primary Android client
- Jason Carter implemented a partially complete Android client (API interface, but no location support)
- Jacob Okamoto implemented the API servers and tests, as well as the Apple iOS client.
- The design document was initially written by Jacob Okamoto; the information regarding the implementation of individual clients in the final document was provided by each client's author and tweaked to match the writing style of the existing document.

## 6.3 Source code

All source code will be stored and version controlled on the University of Minnesota's GitHub Enterprise server at `<https://github.umn.edu/>`. All documentation will be version controlled and stored there as necessary as well.