An Intercepting API-Based Access Control Approach for Mobile Applications

Yaira K. Rivera Sánchez[1], Steven A. Demurjian[1] and Lukas Gnirke[2]
[1]Department of Computer Science & Engineering, University of Connecticut, Storrs, CT, U.S.A.
[2]Department of Computer Science, Oberlin College, Oberlin, OH, U.S.A.

Keywords: Access Control, Application Programming Interface (API), Authorization, Data Management, Mobile Application, Mobile Computing, Representational State Transfer (REST), Role-Based Access Control (RBAC).

Abstract: Mobile device users employ mobile applications to realize tasks once limited to desktop devices, e.g., web browsing, media (audio, video), managing health and fitness data, etc. While almost all of these applications require a degree of authentication and authorization, some involve highly sensitive data (PII and PHI) that must be strictly controlled as it is exchanged back and forth between the mobile application and its server side repository/database. Role-based access control (RBAC) is a candidate to protect highly sensitive data of such applications. There has been recent research related to authorization in mobile computing that has focused on extending RBAC to provide a finer-grained access control. However, most of these approaches attempt to apply RBAC at the application-level of the mobile device and/or require modifications to the mobile OS. In contrast, the research presented in this paper focuses on applying RBAC to the business layer of a mobile application, specifically to the API(s) that a mobile application utilizes to manage data. To support this, we propose an API-Based approach to RBAC for permission definition and enforcement that intercepts API service calls to alter information delivered/stored to the app. The proposed intercepting API-based approach is demonstrated via an existing mHealth application.

1 INTRODUCTION

In the last decade, the increase of mobile devices (e.g., smartphones, phablets, tablets) has led to the decrease in the usage of stationary devices (e.g., desktop computers). Mobile devices have taken over daily tasks such as reading a document, browsing the internet, managing emails, gaming, social media, health & fitness, e-books, banking, email, music, etc. According to (Lella et al., 2015), mobile application (app) usage is rapidly increasing among mobile device users, surpassing the time spent on a mobile device web browser as well as the time spent utilizing their desktop computers.

Mobile applications often contain dynamic data, which requires delivering data taken from a data source (e.g., repository, database, etc.) and/or storing data to/from the source, both at frequent intervals. In order to do these types of data transactions between a mobile app and a server/database, an Application Programming Interface (API) is utilized. The data that is displayed/obtained from an application can vary in sensitivity, ranging from seen by anyone (non-sensitive data) to custom subsets for specific users (highly sensitive data such as personally identifiable information (PII) and protected health information (PHI)). Access control mechanisms are commonly utilized to secure highly sensitive data in order to control which information each user can access/store in a particular system, with the proviso that disclosing the wrong information could lead to serious consequences (Rindfleisch, 1997). The three dominant access control models (Sandhu and Samarati, 1994) to achieve this are: role-based access control (RBAC) which defines roles with permissions on objects that are assigned to users; discretionary access control (DAC) where security policies are established based on the user's identity and authorization and can be delegated; and, mandatory access control (MAC) where sensitivity levels (Top Secret, Secret, Confidential, and Unclassified) are assigned to objects (classifications) and users (clearances) to control who can see what.

The work presented in this paper focuses on

Sánchez, Y., Demurjian, S. and Gnirke, L.
An Intercepting API-Based Access Control Approach for Mobile Applications.
DOI: 10.5220/0006354301370148
In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, pages 137-148
ISBN: 978-989-758-246-2

securing highly sensitive information (PII and PHI) that is present in many mobile applications and is accessible via an API, where the data transactions between a mobile app and a server are performed via calls to the services of the API. This is achieved via the utilization of RBAC in a two-phase process of definition and enforcement on a user of a mobile app with a given role to control which services of the app's API can be invoked. First, RBAC permissions are defined on a role-by-role basis on the API services of the mobile app to identify which services can be called by which user by role utilizing the mobile app. Second, RBAC enforcement is achieved at runtime by intercepting each of the mobile app calls of a user by role on authorized services in order to perform real-time permission checks. Our choice of RBAC to control APIs is motivated by its wide usage for securing highly sensitive data for: corporate data (West, 2015); and, healthcare data in electronic health records (Fernández-Alemán et al., 2013). Another motivation for the latter case is the emergent Fast Healthcare Interoperability Resources (FHIR) (FHIR, 2016) standard for exchangeable healthcare resources that are accessible via an API from EHRs and other health information technology systems.

Through the mobile app API, we seek to provide a means for a user playing a role to be constrained to deliver/store data by limiting access to API services when utilizing the mobile app via the interception of the API services. According to (Cobb, 2014), every API service should be verified to ensure that the user accessing the mobile app has the necessary permissions to manage the requested data. The Intercepting API-Based approach presented in this paper supports the interception of API services by generating a new API that mirrors the original mobile app API (in terms of signatures) and serves as a wrapper which includes calls to the original mobile app API to proceed based on RBAC checks that control the data that is displayed (delivered) and managed (stored). The larger intent of our research would be to define RBAC, MAC, and DAC permissions on API services and intercept calls for access control permission checks that determine the filtered information returned to the mobile app and control information that can be stored in the mobile application's server. For the purposes of this paper, we focus on RBAC.

The remainder of the paper has five sections. Section 2 reviews background on RBAC and APIs, motivates the need to securely control APIs in mobile apps, and describes the Connecticut Concussion Tracker (CT^2) mHealth app for tracking concussions from kindergarten to 12th grade. Section 3 introduces the Intercepting API-Based approach for RBAC definition and enforcement and, underlying infrastructure to secure data in mobile applications through a combination of API and RBAC in order to realize the intercepting API-based approach. Section 4 provides a proof-of-concept demonstration of the intercepting API-based approach into the CT^2 mHealth app and discusses the limitations of our work and possible solutions. Section 5 compares and contrasts related work to our approach. Finally, Section 6 concludes the paper and discusses ongoing work.

2 BACKGROUND, MOTIVATION, AND THE CT^2 MHEALTH APP

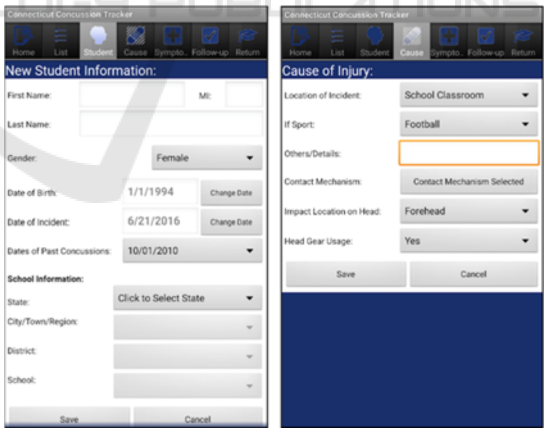
This section provides: background on RBAC and APIs; motivation on the increasing role of APIs and a need for security; and, a review of the Connecticut Concussion Tracker (CT^2) mHealth application.

First, access control mechanisms are utilized to manage which permissions should be granted or denied in regards to the resources of a system or application. One of the most popular mechanisms is role-based access control (RBAC), proposed in (Ferraiolo and Kuhn, 1992) and established as a standard (Ferraiolo et al., 2001) in 2004. In RBAC, *users* are assigned *roles* and each role contains different *permissions*, which contain the policies of which operations and objects a user with a particular role can have access to. Note that each user is limited to one assigned role per session. For the purposes of our paper, we apply RBAC concepts at the API level of the mobile app in support of the proposed approach to define by role which services of the API can be called at which times and under which conditions that are then enforced when a service is invoked by a user/role combination.

Second, in order to do data transactions between a server/database and a mobile application, many developers utilize the *Application Programming Interface (API)* concept. This consists of different tools, protocols, and libraries used to interface data to an application (Beal, 2014). Basically, the client sends a request through the means of a URL, the API receives the URL and interprets it, and then sends this to the data source. The data source will then execute the request and send back a response to the API. The API encodes the response in a human-readable format (e.g., JSON, XML) and sends the response in this format to the client. Some advantages of APIs are: utilized in several applications as most of them are modular (e.g., Facebook Graph API (Facebook, 2014)); useful in applications that contain dynamic data; facilitate the sharing of data or processes between two systems; and, are highly interoperable (Developer Program, 2012; Flanders et al., 2012). The concept of API originated with traditional desktop devices and is now being heavily utilized in mobile applications. The proposed intercepting API-based approach is aimed towards APIs that are built under the REST architectural style (REST API Tutorial, 2012) and that use HTTP as a transfer protocol (Rouse, 2006).

In terms of motivating the ideas in the paper, we acknowledge one of the most recognized options to display (deliver) and manage (store) dynamic data in a mobile app is to utilize the concept of API. However, before attempting to implement an API, one must evaluate their security risks and their effective management (Collet, 2015). For example, consider the recent security breaches in Snapchat and Instagram APIs. Snapchat, a mobile app that enables users to view and send self-destructive pictures and videos (Snapchat, 2011), had a data breach that affected 4.6 million users (Snapchat, 2013). The company quickly posted a statement revealing that the vulnerability allowed individuals to compile a database that contained usernames and phone numbers of users of the mobile app and, that this problem came from their private API. To address this issue, Snapchat is attempting to identify which third-party applications offered in the iTunes store and Google Play store are accessing their private API and any application that uses it is accessing Snapchat's information without their permission (Zeman, 2015). Instagram, a mobile app that allows users to take pictures and share them with family and friends (Instagram, 2010), had a password breach in 2015 (Dellinger, 2015). The breach allowed a third-party application to steal more than 500,000 usernames and passwords, and used the information to post spam on Instagram accounts without permission. To remedy this, Instagram is now reviewing all of the applications that utilize their API and adding new usage policies (Larson, 2015). Clearly both public and private APIs need to be continuously secured and monitored to prevent disclosure of restricted information from occurring. To address this issue, a number of companies have added security and associated management mechanisms to APIs.

Lastly, to serve as an example throughout the paper and in support of the prototyping of Section 4, the Connecticut Concussion Tracker (CT^2) mHealth application, database, and its server are utilized. CT^2 is a collaboration between the Departments of Physiology and Neurobiology, and Computer Science & Engineering at the University of Connecticut and Schools of Nursing and Medicine in support of a new law passed in the state of Connecticut to track concussions of kids between ages 7 to age 19 in public schools (CT Law HB6722) (Connecticut General Assembly, 2015). The CT^2 application is for Android and iOS devices and utilizes a REST API in order to manage its data. The CT^2 mHealth app contains seven tabs ('Home', 'List', 'Student', 'Cause', 'Symptoms', 'Follow-up', and 'Return') where: the 'Home' tab allows the user to add a concussion, to retrieve an open case, or to find a student by name; the 'List' tab which contains the list of students the user has permission to view and, for each student gives him/her the option to add a concussion or edit an existing one; the 'Student' tab (left screen in Figure 1) allows the user to input the student's general information (e.g., name, birthdate, school) and the date of concussion; the 'Cause' tab (right screen in Figure 1) allows the user to specify how an where the concussion occurred; the 'Symptoms' tab allows users to record the symptoms the student had within 48 hours and other pertinent data; the 'Follow-up' tab allows users to record the status of the student over time; and the 'Return' tab allows users to specify when the student can return to various activities at school.



New Student Information:

First Name:
MI:
Last Name:
Gender: Female
Date of Birth: 1/1/1994
Date of Incident: 6/21/2016
Dates of Past Concussions: 10/01/2010
School Information:
State: Click to Select State
City/Town/Region:
District:
School:

Cause of Injury:

Location of Incident: School Classroom
If Sport: Football
Others/Details:
Contact Mechanism: Contact Mechanism Selected
Impact Location on Head: Forehead
Head Gear Usage: Yes

Figure 1: Two Screens of the CT^2 mHealth App.

3 INTERCEPTING API-BASED APPROACH

In this section, the *Intercepting API-Based* approach is explored. This approach offers the versatility of intercepting original API service calls and has no impact on the source code of the mobile application. We differentiate between three types of APIs in the discussion: the *original mobile app APIs* that are used by the mobile app; the *intercepting mobile app APIs* that have the same signatures as the original mobile app APIs to replace these and provide permission checks; and, the *renamed mobile app APIs* (former original mobile app APIs) that are wrapped by the intercepting mobile app APIs. The remainder of this section provides details of our proposed intercepting API-based approach by: defining the architecture of a mobile app extended with our work in Section 3.1; classifying the services of an API so that they can be assigned to users by role in Section 3.2; explaining the interactions and infrastructure of the intercepting API-based approach in Section 3.3; and, by reporting the programmatic changes needed in order to apply the approach to a mobile app in Section 3.4.

3.1 Proposed Architecture

For the general architecture of a mobile app we employ a client mobile app (Microsoft Corporation, 2008) augmented with the intercepting API-based approach. We focus on client applications since these are easier to maintain and assume that the app is always fully connected to the Internet. This assumes that all of the data is processed server-side and does not contain cache and local data. The architecture consists of four main layers as shown in the left side of Figure 2: the *User Layer* which symbolizes the users of the mobile application; the *Presentation Layer* which consists of the UI components of the mobile application; the *Business Layer* which contains the logic of the mobile app (e.g., libraries, APIs, source code); and, the *Data Layer* which contains all of the data the mobile app manages (e.g., retrieves, inserts).

The right side of Figure 2 details the architecture of the intercepting API-based approach across the four layers in three groups. The first group, Role Assignment, involves the user layer and contains the users of the mobile app and their assigned roles. This is achieved via a separate user interface the security administrator will have access to in order to manage users and roles (not shown or discussed). The second group, Define RBAC Permissions on API Services by Role, spans the presentation and business layers and contains the original mobile app API services to retrieve/insert data from/into the data source. This group is utilized to define RBAC permissions on a role-by-role basis on which mobile app API services are authorized to each role, which in turn will be assigned to different users. Once permissions by role on the mobile app API are defined, our approach can intercept API services utilized by the mobile app in order to perform security and permissions checks. To transition from the second to third group, our intercepting API-based approach utilizes the data layer as a pass via the renamed API service calls, and as a result, does not require modifying the source code of the mobile app in order to achieve. Lastly, the third group, Enforce Permissions on API Services by Role, contains the RBAC policies that need to be incorporated in the original data source(s) so that they can be enforced. This includes a new set of intercepting API services that must be defined and then utilized to replace the original mobile app API services to enforce the defined RBAC policies to control the data that is displayed (delivered) and managed (stored) on a user/role combination.

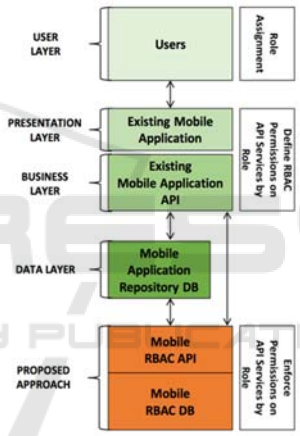
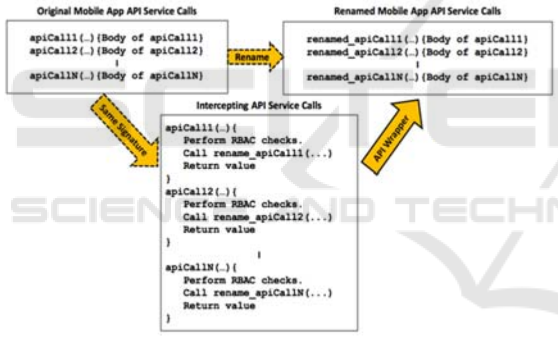

USER LAYER
Users
Role Assignment
PRESENTATION LAYER
Existing Mobile Application
BUSINESS LAYER
Existing Mobile Application API
Define RBAC Permissions on API Services by Role
DATA LAYER
Mobile Application Repository DB
PROPOSED APPROACH
Mobile RBAC API
Mobile RBAC DB
Enforce Permissions on API Services by Role

Figure 2: Intercepting API-Based Approach Architecture.

To illustrate the third group, Figure 3 details the modifications of the original API services that are needed for interception. Specifically, for a mobile app, there is a set of original mobile app API services, as shown in the left side of Figure 3. To maintain the functionality of the mobile app and provide an ability to continue to invoke services by name, the original mobile app API services are renamed (as shown on the right side of Figure 3) in order to reuse the original name of the original service for the new intercepting API services so that services from the mobile app remain unchanged (would now be occurring against the intercepting services). For each original mobile app API service, we define a corresponding intercepting API service, as shown in the bottom (middle) part of Figure 3, that is able to: perform RBAC security checks for the user/role combination; call the corresponding mobile app API service (if it is allowed); and then return either filtered data (retrievals) or a success/failure (inserts, updates, or deletes) status.

The mobile app is still able to invoke the same APIs by name and signature, which are now the intercepting API services (with the same signature) that are able to step in and interrupt the process. As a result, the intercepting API services act as a wrapper that adds a security layer to the original API services. The dashed arrows in Figure 3 indicate that the process of renaming the original API services as well as the process of creating the intercepting file needs to be done only once. Therefore, the developer only needs to create these files once and after that security administrators can manage the RBAC policies without modifying the server-side portion of the mobile app through the means of a separate user interface. The solid arrow indicates the way that the API behaves when a user makes a request through the mobile app; first, the request is intercepted in order to be evaluated with the pertinent access control policies and then, depending on the result, we either proceed to execute the request or send an error message to the user who sent the request.


Original Mobile App API Service Calls
Rename
Renamed Mobile App API Service Calls
Same Signature
Intercepting API Service Calls
API Wrapper

```
apiCall1(…){Body of apiCall1}
apiCall2(…){Body of apiCall2}
        ⋮
apiCallN(…){Body of apiCallN}
```

```
renamed_apiCall1(…){Body of apiCall1}
renamed_apiCall2(…){Body of apiCall2}
        ⋮
renamed_apiCallN(…){Body of apiCallN}
```

```
apiCall1(…){
   Perform RBAC checks.
   Call rename_apiCall1(...)
   Return value
}
apiCall2(…){
   Perform RBAC checks.
   Call rename_apiCall2(...)
   Return value
}
        ⋮
apiCallN(…){
   Perform RBAC checks.
   Call rename_apiCallN(...)
   Return value
}
```

Figure 3: Conceptual API Process.

3.2 Classifying Services of APIs

This section discusses the way that the API of a mobile app is viewed from a RBAC security perspective in order to control who can invoke which service(s) of an API at which times, and the way that each service is viewed from a security standpoint. From a RBAC perspective, we partition the services of an API into two broad categories: secure and unsecure services. *Secure services* are a subset of the API that require control from a security perspective and can be assigned to individual roles. Not all of the API services need to be in the secure category; for example, API services to load drop downs, display web content, etc., may not need to be secure. The secure API services are the ones that will lead to data that is stored/edited/displayed that must be controlled by role. *Unsecure services* need not be assigned and are available to any user.

The following four definitions formalize a mobile app, a role, and permissions for services.

Defn. 1: A mobile application MA has an API $\alpha = \{\alpha_1, \alpha_2, \dots, \alpha_k\}$ where each α_j is a service that has a service name, set of service parameters, and a return type. Note that services are either web or cloud APIs.

Defn. 2: A role r is defined as a two-pair $r = < r_{ID}, r_{Name} >$ with unique r_{ID} identifier and name r_{Name}.

Defn. 3: The API α of a mobile application *MA* can be partitioned into two disjoint sets Secure API S_α and Unsecure API U_α in regards to the services that are to be assigned by role:

- Secure API $S_\alpha \subseteq \alpha$ are the services of MA that need to be controlled.
- Unsecure API $U_\alpha \subseteq \alpha$ are the services of MA that do not need to be controlled where $\alpha = S_\alpha \cup U_\alpha$ and $S_\alpha \cap U_\alpha = \emptyset$ (e.g., $U_\alpha = \alpha - S_\alpha$).

To facilitate permission assignment, we define:

Defn. 4: Secure API Permission Assignment: Each role r_p can be assigned *Secure API role permissions* which represents a subset of services in the Secure API S_α (Defn. 3) that denote those services that can be invoked by a user playing role r_p.

To illustrate Defns. 1 and 2, let MA = CT[2] and four roles be defined: the *Nurse* role, which has access to all seven tabs (see Figure 1) for a school nurse to manage a student's concussion incident from its occurrence to its resolution; the *Athletic Trainer* (*AT*) role which has access to home, list, student, cause, and symptoms tabs (see Figure 1) to do a limited preliminary assessment if a concussion incident occurs at the event; the *Coach* role, which has access to home, list, student and cause tabs (see Figure 1) to report a concussion incident at an athletic event with very limited information on the student; and, the *Parent* role, which has access to home, list, and student tabs (see Figure 1) to view and edit his/her children's general information (e.g., name, date of birth) in addition to being able to track the current status of his/her children that have ongoing concussions.

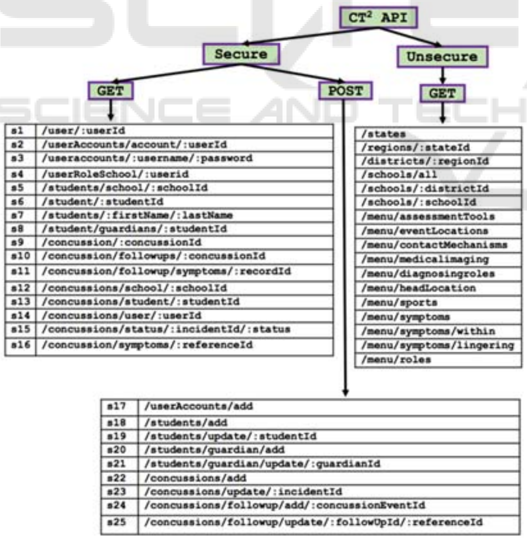
The access to the different screens are reflected in the services of the Secure API from Figure 4 that are assigned to each role. The subset of the Secure APIs for the Nurse, Athletic Trainer (AT), Coach, and Parent roles have the following Secure API Permission Assignments:

- Nurse: Assigned to s17 to s25

- AT: Assigned to s18, s22, and 24
- Coach: Assigned to s18 and s22
- Parent: Assigned to s19

Notice each of the roles has been assigned a subset of the Secure API from Figure 4 by referring to the unique IDs assigned to each service. In addition, note that we are only assigning a subset of the services s17 to s25 to the four roles. These are the Secure services whose HTTP method is POST; we are assuming that all the roles have access to the Secure services whose HTTP method is GET (services s1 to s16 in Figure 4).

Now, for Defs. 3 and 4, there are forty-two REST API services to retrieve, insert, and update information. Figure 4 shows the way that the available API services are classified in the CT^2 mobile app based on the Secure and Unsecure APIs. Basically, we have divided the services into the ones that need to be secured (the Secure API) and the ones that do not need to be secured (the Unsecure API) as they do not contain confidential data. Identifying the services beforehand will benefit us in reducing the overhead of achieving the intercepting API-based approach since we only need to apply additional security policies in the intercepted services that are part of the Secure API.



s1	/user/:userId
s2	/userAccounts/account/:userId
s3	/useraccounts/:username/:password
s4	/userRoleSchool/:userid
s5	/students/school/:schoolId
s6	/student/:studentId
s7	/students/:firstName/:lastName
s8	/student/guardians/:studentId
s9	/concussion/:concussionId
s10	/concussion/followups/:concussionId
s11	/concussion/followup/symptoms/:recordId
s12	/concussions/school/:schoolId
s13	/concussions/student/:studentId
s14	/concussions/user/:userId
s15	/concussions/status/:incidentId/:status
s16	/concussion/symptoms/:referenceId

/states
/regions/:stateId
/districts/:regionId
/schools/all
/schools/:districtId
/schools/:schoolId
/menu/assessmentTools
/menu/eventLocations
/menu/contactMechanisms
/menu/medicalimaging
/menu/diagnosingroles
/menu/headLocation
/menu/sports
/menu/symptoms
/menu/symptoms/within
/menu/symptoms/lingering
/menu/roles

s17	/userAccounts/add
s18	/students/add
s19	/students/update/:studentId
s20	/students/guardian/add
s21	/students/guardian/update/:guardianId
s22	/concussions/add
s23	/concussions/update/:incidentId
s24	/concussions/followup/add/:concussionEventId
s25	/concussions/followup/update/:followUpId/:referenceId

Figure 4: CT^2 API Service Classification.

3.3 Interactions and Infrastructure

Figure 5 depicts the detailed interactions of the *intercepting API services* approach. The top portion of Figure 5 embodies the core RBAC model that we are utilizing as a basis for our approach. The middle portion of Figure 5 represents, at a high level, the various steps and the associated process that needs to be performed in order to authorize a user playing a role to manage certain fields of a mobile application. Note that a user's role has been set administratively. The steps from the user's perspective from left to right are: log in to his/her mobile app account; for successful login, extract the user's role that is part of the login credentials; store the extracted user role in a secure access token in order to use it in future API services; utilize the mobile app which results in multiple mobile app API calls and are intercepted (data processing step in Figure 5); and, the intercepted API service interacts with the RBAC permissions and policies to enforce the defined security before calling the original mobile app API service. This final step involves each role having a specific set of API permissions (see Defn. 4 in Section 3.2) as a subset of the Secure API (see Defn. 3 in Section 3.2).

There are two possible requests that can occur as an end result of the interactions in the middle portion of Figure 5. In the insert/update/delete request (via an intercepting mobile app API service in the upper portion of the RBAC API rounded square), the request will be intercepted to perform the RBAC checks against the Secure API and Secure Permissions for that role, and depending on the response, the action will either get done (the original mobile app API service is allowed) or not. In the retrieve request the user is trying to retrieve data (via an intercepting mobile app API service in the lower portion of the RBAC API rounded square), the data source will perform this action but the mobile app API is intercepted to allow RBAC checks to be performed to verify whether the requested service is in the Secure API for that user. This will allow the intercepted API service to determine if the user has access to all/some/none parts of the data with the resulting original API service returning data (all/some case) or null/error message (none case). In both types of requests, the security policies utilized to perform the RBAC checks in the intercepting API service that involves checking the permissions to the Secure API are stored in the database in Figure 6.

The primary changes to support the intercepting API services approach are made in the backend of the mobile app (server-side – bottom portion of Figure 2) and include the addition of RBAC security policies on the permitted subset of Secure API by role, in a permission database to create the mapping from the original mobile app API services to the corresponding new intercepting API services. Figure 6 shows the three database tables that need to be added in order to realize the RBAC security policies on the server side: the *roles* table that contains the list of available roles; the *api_calls* table that contains the list of the API service calls that are available along with an id for each one of these; and, the *api_calls_map_access* table that defines whether a role has permission to a specific API service call or not. To support these tables, the *role_id* foreign key has been added to the existing *user_accounts* table in order to reference and link the user to a role. Note that there are two new additional tables that are placed in the database as well (*api_calls_parameters* and *http_methods* tables) nevertheless, these are not part of the security policy but hold part of the content that could be utilized to generate the intercepting code. More details about both of these tables are discussed in Section 3.4.

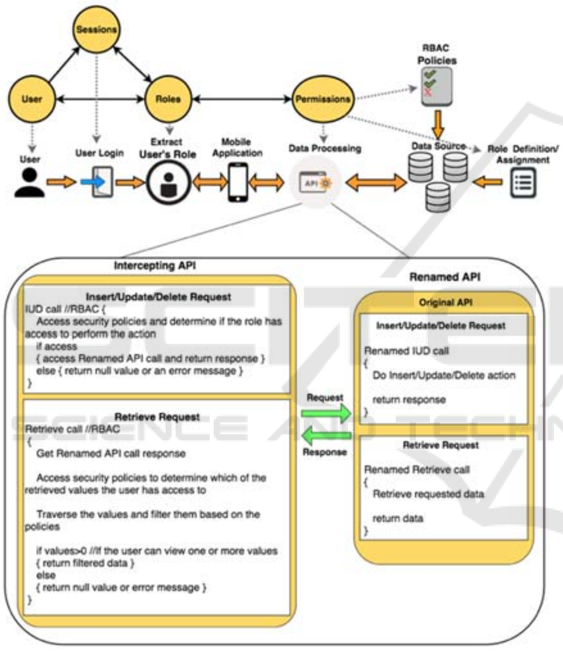

Sessions
User
Roles
Permissions
RBAC Policies
User
User Login
Extract User's Role
Mobile Application
Data Processing
Data Source
Role Definition/ Assignment
API
Intercepting API
Insert/Update/Delete Request
IUD call //RBAC {
Access security policies and determine if the role has access to perform the action
if access
{ access Renamed API call and return response }
else { return null value or an error message }
}
Retrieve Request
Retrieve call //RBAC
{
Get Renamed API call response
Access security policies to determine which of the retrieved values the user has access to
Traverse the values and filter them based on the policies
if values>0 //If the user can view one or more values
{ return filtered data }
else
{ return null value or error message }
}
Request
Response
Renamed API
Original API
Insert/Update/Delete Request
Renamed IUD call
{
Do Insert/Update/Delete action
return response
}
Retrieve Request
Renamed Retrieve call
{
Retrieve requested data
return data
}

Figure 5: Interactions of Intercepting API Calls with RBAC Model as Base.

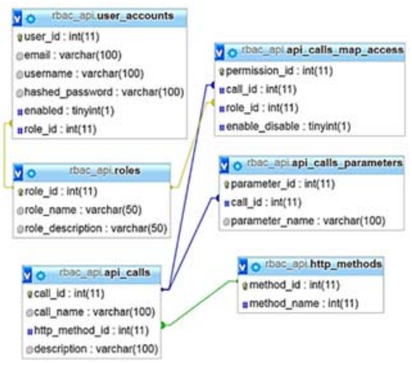
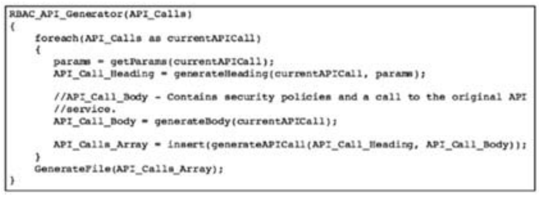


Figure 6: Security Policy Tables in the Database.

The database tables as given in Figure 6 provide the infrastructure that is needed to link the original mobile app API service to its corresponding new intercepting API counterpart. Each new intercepting API service has the same signature (same address and parameter) as its original mobile counterpart, so that the intercepting API service can substitute for the original API of the mobile app to allow the aforementioned security checks for retrieve and insert/update/delete requests. As a result, the intercepting API services effectively wrap the original mobile app API services. The mobile app now seamlessly invokes the intercepting API services. These intercepting API services contain the appropriate RBAC security checks on the subset of the Secure API by role, adding a layer of security to enforce the policies. The renamed mobile app API services will be invoked based on the outcomes of the security checks. The end result is that the mobile app will appear differently based on the user/role combination, to limit information that is delivered (retrieve request) or that impacts the data that is stored (insert/update/delete requests).

3.4 Algorithm Generation

The intercepting API-based approach utilizes an algorithm to automatically generate the intercepting code. In order to achieve this, we need to create a file that contains the same API services as the original mobile app API via the generate function *RBAC_API_Generator* (depicted as a pseudocode in Figure 7) which has a parameter that contains an array of all the API services available in the mobile app (line 1 of Figure 7). The API services reside in the api_calls table in Figure 6, which also contains the respective HTTP method (e.g., GET, POST) from both the Secure and Unsecure APIs. Note the generation of the Unsecure API services is trivial since they simply pass through from the new intercepting services to the original API services without any required security checks.



```
RBAC_API_Generator(API_Calls)
{
    foreach(API_Calls as currentAPICall)
    {
        params = getParams(currentAPICall);
        API_Call_Heading = generateHeading(currentAPICall, params);

        //API_Call_Body - Contains security policies and a call to the original API
        //service.
        API_Call_Body = generateBody(currentAPICall);

        API_Calls_Array = insert(generateAPICall(API_Call_Heading, API_Call_Body));
    }
    GenerateFile(API_Calls_Array);
}
```

Figure 7: General Idea of Code for Generating the Body of an Access Control Service.

For each of the API services in the array, we obtain the parameters (if any), which are stored in the api_calls_parameters table in Figure 6, and store these in an array (line 5 of Figure 7). Once we obtain the parameters of the API service that is being evaluated, we can generate the heading of the intercepting API call function by using the current API service as well as its parameters (if any) (line 6 of Figure 7). After generating the heading for the intercepting API call function, we then generate the body of the API service, which contains the security policies (i.e., the security permission to a subset of the Secure API by role) for that specific service and calls the original mobile app API service if the user has access to it (line 9 of Figure 7). The resulting heading and body of the current API service will be stored in an array (line 11 of Figure 7). Once all of the intercepting services have been created, we traverse the array in which they are stored in order to generate the intercepting file (line 13 of Figure 7). This approach was achieved with the assumption that the API we want to intercept was created with Slim (Slim, 2015), which is a PHP micro framework that allows people to write web applications as well as APIs. Therefore, we consider that our approach is useful for those mobile applications/web applications that contain an API developed with a PHP-based framework.

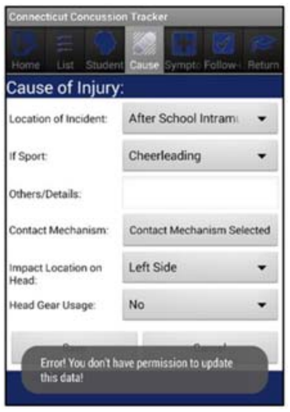
4 PROOF-OF-CONCEPT AND LESSONS LEARNED

This section presents the proof-of-concept demonstration of the intercepting API-based approach reviewed in Section 3 coupled with lessons learned. Section 4.1 explores the usage of the intercepting API-based approach on the Connecticut Concussion Tracker (CT^2) Android mHealth application, database, and server. Section 4.2 examines lessons learned including the limitations found when implementing our approach and an alternate proposed approach to address them.

4.1 Implementation

Programmatically, we have source code for the Android version of the CT^2 app and a REST API that accesses the CT^2 MySQL database. The realization of the intercepting API-based approach is achieved without any modification to the mobile app UI and is intended to allow fine-grained access control on the information that is displayable and/or storable of the authorized tabs for each user/role combination via controlling access to the services of the Secure API on a role-by-role basis. There is a very clear mapping from the process described in the previous section and the accompanying figures to its realization in CT^2. The database in CT^2 was augmented with a table that contains a list of all the API calls available along with a call_id (similar to the *api_calls* table shown in Figure 6), and a table that contains the security policies that determine which calls the roles have access to (similar to the *api_calls_map_access* table shown in Figure 6). Given these database changes, we then take the original CT^2 REST API calls and rename these using an analogous process to the one shown in Figure 3. Afterwards, a set of new CT^2 intercepting REST API calls were generated using the algorithm in Figure 7 that will perform a series of RBAC checks based on the services of the Secure API assigned to each role, and if successful, invoke the corresponding renamed original CT^2 REST API calls.

From a process perspective, the steps in CT^2 follow the middle portion of Figure 5. The user logs on to the CT^2 mHealth app and a combination of his/her user id with his/her role is stored in a JSON Web Token (JWT) (JWT, 2015) in the session in order to support the class that manages the API services as presented and discussed in Section 3 and in Figure 4 for roles, the Secure API, and Permissions (Defns. 2, 3, and 4, respectively). Figure 8 illustrates the impact of the intercepting API services (error message) and associated process for a user with the role of *Coach* which has access to only the home, list, student, and cause tabs. This role can add basic information on the 'Student' tab, can add information in the 'Cause' tab and, after adding the information, the user can view but not edit. The error in Figure 8 indicates that a user with the Coach role tried to update information on the Cause tab.



Cause of Injury:

Location of Incident:	After School Intram
If Sport:	Cheerleading
Others/Details:	
Contact Mechanism:	Contact Mechanism Selected
Impact Location on Head:	Left Side
Head Gear Usage:	No

Error! You don't have permission to update this data!

Figure 8: CT^2 mobile app screen.

The original mobile app CT^2 API services support the insert of information in the database and the intercepting API service in this case allows that first save to occur. At a later point in time, if the user with the role of Coach attempts to edit and perform another save, the intercepting API service in this case performs the RBAC check that does not allow the edit. As a result, the intercepting API service alerts that he/she does not have permission to perform that action. This checking process consults the Secure API Permission Assignments of the Secure API by role, to verify if the Coach role has access to the desired action. Specifically, for the Coach role, all of the POST services (s17 to s25) are intercepting services and as a result each call performs an RBAC check. If the service is s18 or s22, the RBAC check succeeds and the corresponding original API method is called. Otherwise, if the service is s17, 19, 20, 21, 23, 24, or 25, then the RBAC check fails and the original API method is not called. The other tabs of CT^2, 'Symptoms', 'Follow-Up' and 'Return', are still visible within the app. However, when a user with the Coach role attempts to access one of these tabs, the app will try to obtain the pertinent data via the former original CT^2 API service that has been replaced by a new CT^2 intercepting API service that checks for permissions and returns that the specified role does not have permission to retrieve the data for those screens. Attempting to access other tabs with services other than those authorized for the Coach role (s18 and s22) results in a failed RBAC check that denies the action.

To evaluate if the approach affected the performance of the mobile app in any significant way, we tracked the time the app took to perform services with and without interception. Table 1 contains the time average (in seconds) the mobile app took to process the services requested by the clients without the intercepting API-based approach versus the time it took to process the services requested by the clients with interception. For this evaluation, we assumed multiple users were utilizing the mobile app simultaneously and that each of these users made one request only, therefore, we emulated the calling of 25, 50, 100, 250 services concurrently to determine the time variation. By looking at the results in Table 1, we observe that the time taken to process services in their original state is smaller than the time elapsed when the services are processed with the use of our approach. Nevertheless, we consider that even though adding time to process services is not desirable, our intercepting API-based approach is still quick since it takes only a matter of microseconds to analyze the service requested by a user and return a secured response. Therefore, in terms of overhead, the mobile app did not take a significant amount of additional time to execute a user's request.

Table 1: Original vs. Intercepting API service Performance Evaluation (in seconds).

#of services	25	50	100	250
Original	3.34×10^{-6}	3.41×10^{-6}	3.54×10^{-6}	3.98×10^{-6}
Intercepting API	1.08×10^{-4}	1.11×10^{-4}	1.18×10^{-4}	1.19×10^{-4}

4.2 Limitations and Discussion

The intercepting API-based approach performs security checks to determine whether the API service can occur based on the user and his/her role that has been given permission to call a subset of the services of the Secure API, thereby controlling what is returned to the user. The approach as described in this paper essentially creates a replica of the API that the mobile app uses so that the mobile app utilizes our intercepting API which can perform security checks and then pass the call through to the original mobile app API if the security permissions are met (i.e., the role is authorized to the service of the Secure API attempting to be invoked). This was accomplished by renaming the calls to the services of the original mobile app API. There are two possible issues with this approach. First, we may not have access to the mobile app API. Second, even if we do, then the renaming would require changes to the service names of the original mobile app API. As a result, we believe that it is possible to realize a solution to eliminate these two issues by proposing an intercepting server that does not modify the original mobile app API file but contains the original service calls that the mobile app has access to and then forward each call from the intercepting API service of the same name to the original API service.

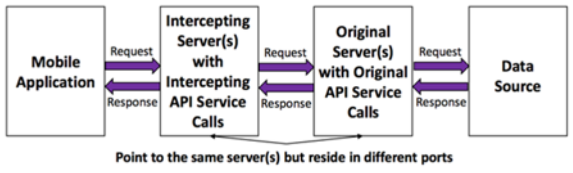

Mobile Application
Request
Response
Intercepting Server(s) with Intercepting API Service Calls
Request
Response
Original Server(s) with Original API Service Calls
Request
Response
Data Source
Point to the same server(s) but reside in different ports

Figure 9: Alternate Process for the Intercepting API-Based Approach.

Figure 9 illustrates an alternate process for the Intercepting API-based approach that establishes an intercepting server that has an API that will mimic what the mobile app is expecting but is our intercepting API (second box from the left in Figure 9). The intercepting API server must also be able to mimic multiple APIs since the mobile app may interact with several of these (depicted in third box from the left in Figure 9). The original and intercepting servers would need to be run on different ports if on the same host with the intercepting server being accessible publicly while the original might only be accessible locally. The intercepting server(s) could forward any allowed calls to the original server(s) and filter the results as needed before returning to the client. This is facilitated by using the login credentials (user/role combination) to determine the security level in each of the API Services. Currently, we manage to pass on the user id and the user's role between calls by storing these server-side in a JSON Web Token (JWT) (JWT, 2015). This is done to secure the user's data and to verify that the user has access to the action he/she requested by utilizing his/her role.

5 RELATED WORK

There are many efforts that propose access control mechanisms to secure mobile applications by limiting the permissions and resources a mobile app can access in different areas of the mobile device/app. In this section, we discuss several existing proposed approaches that attempt to apply access control mechanisms on different locations on a mobile device and, we explain the way our approach compares and contrasts.

The first area of related work involves sensor management on smartphones that is commonly addressed by applying access control mechanisms to the sensors of a mobile device so that mobile apps obtain fine-grained permissions. This facilitates the managing of sensor data in mobile apps (e.g., user's location, use of Bluetooth) (Cappos et al., 2014; Xu and Zhu, 2015). BlurSense (Cappos et al., 2014) and SemaDroid (Xu and Zhu, 2015) allow users to define and add privacy filters to sensor data, through the means of a user interface, that is being used on their mobile applications. In contrast to these efforts, our work presented in this paper focuses on API access control management for the API services that are utilized within a mobile app to populate data in the app and to add/edit data and store it in a data source. In other words, instead of focusing on modifying the operating system to filter sensor data we modify the backend of a specific mobile app and filter the data that a user can have access to according to his/her role, which can include sensor data as well if there was an API service included in the intercepted API that managed this.

The second area of related work involves permission control in Android in which access control can be applied on the mobile device itself. There are many existing approaches (Beresford et al., 2011; Benats et al., 2011; Wang et al., 2014; Jin et al., 2015; Hao et al., 2013; Backes et al., 2014) that focus on applying fine-grained access control policies to mobile devices that contain Android as their operating system. This is due to the fact that Android contains a coarse-grained access control mechanism when it comes to allowing permissions in mobile applications. In other words, in order for a user to install a mobile app he/she needs to accept all of the permissions that the app requires. This may disregard the fact that some permissions may not be necessary for the app to function and that some of the permissions may not make sense for app that is being downloaded and could result in using the allowed component for malicious purposes (e.g., a flashlight app tells user it needs permission to get the user's location). Adding fine-grained access control to the APIs that Android uses for the device and apps to function properly has been addressed by: mocking the values that an app receives in order to function (Beresford et al., 2011) (e.g., mocking latitude and longitude coordinates); extending the security policies of the mobile device (Benats et al., 2011; Wang et al., 2014; Jin et al., 2015); by rewriting the bytecode of the mobile device (Hao et al., 2013); and by adding security modules to the mobile device (Backes et al., 2014). In contrast to this effort, our work presented in this paper focuses on applying access control mechanisms to the APIs that are not part of the mobile system itself. In addition, most of these works are specific for Android OS/API while ours can be implemented for any type of application (even though we focus on the mobile setting) since our access control approach is enforced server-side.

The third area of related work involves role-based access control and extensions that expand RBAC with context-aware techniques in order to provide finer-grained access control security policies to those systems that contain highly sensitive data. One effort does this by proposing an RBAC model with a spatiotemporal extension for web applications (Aich et al., 2009) and another effort proposes a similar approach but for mobile applications (Abdunabi et al., 2013). The proposed access control system made for web applications (Aich et al., 2009) can be applied to an existing system as a dll component. Another approach proposes a dynamic

RBAC approach for Android devices (Rohrer et al., 2013). That approach focuses on modifying the Android framework to provide a uniform security policy to mitigate security risks in mobile devices that are utilized by users who are part of an enterprise. Finally, an effort (Fadhel et al., 2016) proposed a model that extends RBAC to generate RBAC conceptual policies. Nevertheless, the aforementioned effort does not provide details of which specific application domain(s) the approach could support. Our proposed framework could easily be extended to support other types of access control, can be applied to mobile web applications and, it is not domain-specific; this contrasts to discussed related work.

6 CONCLUSION & ONGOING WORK

This paper presented and discussed an intercepting API-based access control approach for mobile applications achieved via role-based access control defined and enforced on services. Specifically, the work in this paper demonstrated the way that RBAC can be incorporated into an intercepting API that wraps the original mobile app API in order to manage the data displayed in a mobile application. To begin the presentation, Section 2 reviewed background on RBAC and APIs, motivated the usage of RBAC in mobile applications, and described the Connecticut Concussion Tracker (CT^2) mHealth app. Using that as a basis, Section 3 detailed our proposed intercepting API-based approach, which included the transition from the original mobile app API to a renamed mobile app API that is then wrapped with the intercepting API. To demonstrate the feasibility and utility of our work, Section 4 explained the realization of the intercepting API-based approach into the CT^2 mobile application and discussed a way to address the limitations of the proposed approach. Lastly, Section 5 reviewed and contrasted related work to our effort.

As part of our ongoing work, we are researching on the way to incorporate RBAC, MAC, and DAC into the Fast Interoperable Healthcare Resources (FHIR) specification and infrastructure (FHIR, 2016) to facilitate information exchange between a mHealth app and multiple EHRs and health information technology systems. So far, we have published a paper (Rivera Sánchez et al., 2017) with our findings on incorporating RBAC in FHIR through the means of the HAPI FHIR reference implementation (HAPI FHIR, 2014). Our solution utilizes unique capabilities in HAPI FHIR that allow our code to intercept the FHIR server API service calls so that the service calls that the mHealth app makes against the RESTful API service calls are checked against defined RBAC policies. We believe that this in turn will aid us in the development of a general approach for securing data that is managed through the means of APIs.

REFERENCES

Abdunabi, R., Ray, I., & France, R., 2013. Specification and analysis of access control policies for mobile applications. *18th ACM Symposium on Access Control.*

Models and Technologies (SACMAT '13). ACM, pp. 173–184.

Aich, S., Mondal, S., Sural, S., & Majumdar, A. K., 2009. Role Based Access Control with Spatiotemporal Context for Mobile Applications. *Transactions on Computational Science IV: Special Issue on Security in Computing.*

Backes, M., Bugiel, S., Gerling, S., & von Styp-Rekowsky, P., 2014. Android Security Framework: Extensible multi-layered access control on Android. *30th Annual Computer Security Applications.*

Conference, pp. 46-55.

Beal, V., 2014. *API - application program interface.* [Online] Available.

at: http://www.webopedia.com/TERM/A/API.html.

Benats, G. et al., 2011. PrimAndroid: privacy policy modelling and analysis for android applications. *In Symposium on Policies for Distributed Systems and Networks (POLICY '11).* IEEE.

Beresford, A., Rice, A., Skehin, N., & Sohan, R., 2011. MockDroid: trading privacy for application functionality on smartphones. *12th Workshop on Mobile Computing Systems and Applications*. Phoenix, Arizona.

Cappos, J., Wang, R., Yang, Y. & Zhuang, Y., 2014. *Blursense: Dynamic fine-grained access control for smartphone privacy.* [Online] Available at: DOI=10.1109/SAS.2014.6798970.

Cobb, M., 2014. *API security: How to ensure secure API use in the enterprise.* [Online] Available at: http://searchsecurity.techtarget.com/tip/API-security-How-to-ensure-secure-API-use-in-the-enterprise.

Collet, S., 2015. *API security leaves apps vulnerable: 5 ways to plug the leaks.* [Online] Available at: http://www.csoonline.com/article/2956367/mobile-security/api-security-leaves-apps-vulnerable-5-ways-to-plug-the-leaks.html.

Connecticut General Assembly, 2015. *Substitute for Raised H.B. No. 6722.* [Online] Available at: https://www.cga.ct.gov/asp/CGABillStatus/CGAbillst atus.asp?which_year=2015&selBillType=Bill&bill_num=HB6722.

Dellinger, A., 2015. *This Instagram app may have stolen over 500,000 usernames and passwords.* [Online] Available at: http://www.dailydot.com/technology./instaagent-instagram-app-malware-ios-android/

Developer Program, 2012. *Benefits of APIs*. [Online] Available at: http://18f.github.io/API-All-the-X/pages/benefits_of_apis.

Facebook, 2014. *Facebook Graph API*. [Online] Available at:https://developers.facebook.com/docs/graph-api.

Fadhel, A., Bianculli, D., Briand, L. & Hourte, B., 2016. A Model-driven Approach to Representing and Checking RBAC Contextual Policies. *CODASPY 2016*. ACM, pp. 243–253.

Fernández-Alemán, J., Señor, I., Lozoya, P. & Toval, A., 2013. Security and privacy in electronic health records: A systematic literature review. *Journal of Biomedical Informatics*, 46(3), pp. 541-562.

Ferraiolo, D. & Kuhn, R., 1992. Role-Based Access Control. *NIST-NSA National (USA) Computer Security Conference*, pp. 554-563.

Ferraiolo, D. et al., 2001. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC),* Volume 4, pp. 224-274.

FHIR, 2016. *Welcome to FHIR.* [Online] Available at: https://www.hl7.org/fhir/index.html.

Flanders, D., Ramsey, M., & McGregor, A., 2012. *The advantage of APIs*. [Online] Available at: https://www.jisc.ac.uk/guides/the-advantage-of-apis.

Hao, H., Singh, V. & Du, W., 2013. On the effectiveness of API-level access control using bytecode rewriting in Android. *8th ACM SIGSAC symposium on Information, computer and communications security*. Hangzhou, China.

HAPI FHIR, 2014. *HAPI.* [Online] Available at: http://hapifhir.io/

Instagram, 2010. *Instagram*. [Online] Available at: https://www.instagram.com/

Jin, X., Wang, L., Luo, T. & Du, W., 2015. Fine-Grained Access Control for HTML5-Based Mobile Applications in Android. *16th Information Security Conference (ISC)*, pp. 309-318.

JWT, 2015. *Introduction to JSON Web Tokens*. [Online] Available at: https://jwt.io/introduction/

Larson, S., 2015. *Instagram restricts API following password breach, will review all apps going forward.* [Online] Available at: http://www.dailydot.com/technology/instagram-api-restrictions/

Lella, A., Lipsman, A. & Martin, B., 2015. *The 2015 Mobile App Report.* [Online] Available at: https://www.comscore.com/Insights/Presentations-and-Whitepapers/2015/The-2015-US-Mobile-App-Report.

Microsoft Corporation, 2008. *Mobile Application Architecture Guide.* [Online] Available at: http://apparch.codeplex.com/releases/view/19798.

REST API Tutorial, 2012. *Learn REST: A RESTful Tutorial.* [Online] Available at: http://www.restapitutorial.com/

Rindfleisch, T., 1997. Privacy, Information Technology, and Health Care. *Communications of the ACM,* 40(8), pp. 93-100.

Rivera Sánchez, Y. K., Demurjian, S.A., & Baihan, M., 2017. An Access Control Approach for FHIR. *5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (IEEE Mobile Cloud 2017).*

Rohrer, F., Zhang, Y., Chitkushev, L. & Zlateva, T., 2013. DR BACA: dynamic role based access control for Android. *29th Annual Computer Security Applications Conference*. New Orleans, Louisiana, USA.

Rouse, M., 2006. *HTTP (Hypertext Transfer Protocol).* [Online] Available at: http://searchwindevelopment.techtarget.com/definition/HTTP.

Sandhu, R. & Samarati, P., 1994. Access Control: Principles and Practice. *Communications Magazine*, 32(9), pp. 40-48.

Slim, 2015. *Slim a micro framework for PHP*. [Online] Available at: https://www.slimframework.com/

Snapchat, 2011. *Snapchat.* [Online] Available at: https://www.snapchat.com/

Snapchat, 2013. *Finding Friends with Phone Numbers.* [Online] Available at: http://blog.snapchat.com/post/71353347590/finding- friends-with-phone-numbers.

Wang, Y. et al., 2014. Compac: enforce component-level access control in android. *4th ACM conference on Data and application security and privacy*. San Antonio, Texas, USA.

West, A., 2015. *5 Roles of Role Based Access Control.* [Online] Available at: https://www.itouchvision.com/5-roles-of-role-based-access-control-the-software-security-guard/

Xu, Z. & Zhu, S., 2015. Semadroid: A privacy-aware sensor management framework for smartphones. *5th ACM Conference on Data and Application Security and Privacy*. ACM, pp. 61-72.

Zeman, E., 2015. *Snapchat Lays Down The Law On Third-Party Apps.* [Online] Available at: http://www.programmableweb.com/news/snapchat-lays-down-law-third-party-apps/2015/04/07.