Hot code updates in Cordova applications

Jonas Semeelen[1]

[1] *Univeristy of Leuven, Ghent, Belgium*

January 11, 2018

In this dissertation, an alternative and flexible update system was developed for Cordova applications. To achieve this, a qualitative study was done on Cordova update plugins. The study determined the most suitable plugin which was then used as a foundation for the rest of the update system. The functionality of the plugin was expanded by adding an authorisation and authentication server to the system. This made it possible to identify users and thus send them personalised updates.

1 Introduction

Imagine a mobile application which has a wide variety of resources at its disposition. And let's say these resources are wireless sensors in a homecare environment. In order for the app to be able to receive data from the sensors, it will need specific code for each different type of sensor. Now try to imagine all of the different and unique homecare settings. Bear in mind that each homecare setting is designed accordingly to the needs of the resident. This results in a huge amount of different settings.

Before moving on, a bit more context is provided in order for the reader to be able to fully understand the goal of the thesis. The previously stated application is meant for relatives or acquaintances of the patient in need of homecare. The app will assist in monitoring the patient when the caretaker is unable to be physically present at the house. For instance, a trip sensor will alert the caretaker immediately if the patient has tripped somewhere in the house.

Of course the app and the use of the sensors won't come freely. A caretaker will have to pay in order to use the app's functionality. However, the caretaker will only want to pay for the functionality needed by his or her patient. Since the different amount of homecare settings is so high, the app provides different kinds of packages from which the caretaker can choose. This way the caretaker will only pay for the necessary set of functions.

Now we don't want the app itself to contain all of the functionality of all the different available packages. This would make the app too large and de app users would often have to download updates for pieces of code they can't even use because they don't pay for the package. We also don't want to distribute the different packages as seperate applications. This would only complicate matters if a caretaker decides to upgrade to another package. This would require the installation of a whole new app.

Conclusion: we want one flexible and lightweight application which functionality is based on the user. The actual goal of the thesis is to develop a system which can provide update support for said type of application. The newly developed updatesystem will be accompanied by a prototype homecare application in order to demonstrate the update system.

1.1 What to expect from this paper

This paper will discuss all of the steps taken to accomplish the goal. The first chapter is about Cordova and its update plugins. These are two key technologies used during the thesis. They get a whole chapter because the update plugin will be the foundation of the whole system. Next, the design of the system will be thoroughly discussed. This is followed by the implementation of each component and how these implementations were chosen. Finally, an analysis is made of the accomplished system.

2 Process

This chapter will discuss the same steps that are taken in the thesis. At the end of each step, a summary is given about the contribution of the step in the grand picture. After reading this chapter, it should be clear what the system can do and how it's made.

2.1 The update plugin

As you may or may not know, Cordova[1] is a framework which allows the creation of cross-platform applications using standard web technologies such as Javascript, HTML and CSS. Cordova uses plugins as an interface between the Javascript and the native components. Because of the application's hybrid properties, the actual app solely consists out of platform independent code. To update the app, this code can simply be replaced by new code because there are no native components present, thus no recompilation of code is required.

This property makes it possible to update Cordova apps without the interference of an app store. We call these kind of updates 'hot code updates'. There are currently a lot of plugins available in the Cordova plugin store which can perform hot code updates. Some of the existing plugins are written by professionals and continue to be enhanced up until this day. It was then decided that an existing plugin would be used in the system instead of a custom designed one. This way, the update plugin will be able to benefit from continuous updates and bugfixes released by its authors.

After carefully analyzing several plugins, the CodePush plugin by Microsoft came out on top. This plugin offers secure communication with a cloud service where the apps and updates are stored. The plugin grants a unique deployment key to each application that is newly registered in the cloud service. When an app contacts the cloud service to check for updates, the cloud service will use this deployment key to search for the correct application in its database. On top of this, the plugin also comes with a command line interface (CLI) to manage the apps in the cloud service.

Although CodePush seems like a flawless update plugin, a security vulnerability was found during its analysis. When the connection is made with the cloud service right before the download, a parameter is wrongfully given the value *true*. This sets in motion a chain of method calls resulting in a situation where every server certificate is blindly accepted. This makes the user's device vulnerable to a man-in-the-middle attack. The vulnerability is carefully documented in the thesis and a solution to the vulnerability has been suggested.

2.2 The design

After the choice of the update plugin, it was possible to design the rest of the system around the plugin since the plugin itself can't fulfill the needs of a flexible update system. The additional components are supposed to complete the missing functionality. In order to determine which update should be sent to which app user, two additional elements are needed: an authentication and an authorisation element. Using these elements it is possible to identify the requester of the update and authorise the request based on the identity.

[1]https://cordova.apache.org/

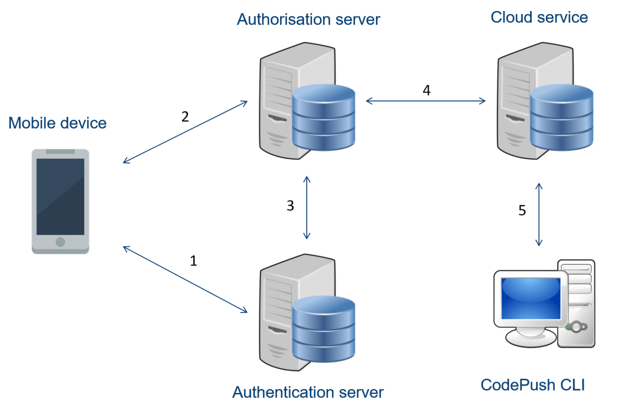

Figure 1: *Design of the system*

Several designs were considered but the design given in figure 1 proved to be the best. The biggest design decision that was made is the use of separate servers for the authentication and authorisation component. In the previous subsection it was already stated that the CodePush plugin comes along with a cloud service and an interface to use this cloud service. The main reason for the introduction of the extra servers is the preservation of the CodePush web server as a whole. By keeping this server unmodified it can be used as a black box and it can keep on receiving updates and bugfixes by the Microsoft development team. A second argument for using this design is the central location of the authorisation server. All the requests from the apps will be intercepted by this server. This allows the authorisation server to be a policy enforcement point (PEP). This a point in a network which executes decisions based on a certain policy. This part is the core of the whole flexible update system.

With each incoming request, the identity of the sender is extracted. Based on this identity, a deployment key is dynamically added to the request before forwarding it to the cloud service. By dynamically adding the deployment key instead of having it statically linked to the application, it is possible to dsitribute updates in a flexible manner. Because the decision policy is based on the user's identity, it is important that every request is properly authenticated. This is done by the authentication server.

The authentication server has two jobs. The first one is to authenticate users when they log in to the application. If the login is succesful, an access token will be granted. This token is used by the app in all subsequent requests made to the authorisation server. There, the token is extracted from the request and is sent to the authentication server for validation. This allows the PEP to do its job correctly.

2.3 The implementation

This subsection will clarify how each component of the system was implemented. It became clear in subsection

2.1 that the system will use the CodePush plugin. If the plugin is used alongside its intended cloud service and CLI, three out of five elements on figure 1 are already explained. Only the authorisation and authentication server remain.

By placing the authorisation server in the center of the system, it was possible to use an existing server which provides authentication mechanisms. The server that is used is an Oauth2 server[2]. Oauth is an open standard for authorisation. It allows users to grant permission to third parties to access their private data. This Oauth2 server already owns the functions needed to authenticate users. Because of this the Oauth2 server can also be used as a black box, just like the cloud service.

The authorisation server acts as an interface between the other components in the network which allows those components to be used as a black box. This was only possible by custom creating the authorisation server. Because of this, the authorisation server doesn't benefit from constant updates and bugfixes like the other elements. But this little disadvantage does not outweigh the benefits of the other components.

The authorisation server is a Node.js[3] web application which uses the Express[4] framework. The combination of these two technologies results in a lightweight and flexible authorisation server. The complexity of the server is relatively low since its only three jobs are verifying accesstokens, adding deployment keys to requests and forwarding these requests to the cloud service.

3 Analysis

3.1 Risks

The analysis of the system focuses mostly on the security aspect. The updatesystem has to be safe to use by apps which handle sensitive information. For this reason, every part of the system is checked to see if there are any vulnerabilities. The next paragraph will discuss the possible attacks, where they might happen in the system and what the risk involved is. Each connection in figure 1 is given a number to make the discussion easier.

The first connection is used to validate the user's login data. If an attacker is listening on this connection, he might find out the username and password. He will then be able to impersonate the user and fool the system.

The second connection is used whenever the update plugin sends a request to the authorisation server. This request contains the user's access token which is used for authentication purposes. If an attacker gets hold of this token, he can impersonate the user even without his or her login data. This is not quite as bad as the previous issue since the validity of an access token has a limited duration. This connection is also the prime target for a man-in-the-middle (MITM) attack. An attacker might try to inject a malicious update into this connection when the user is performing an update.

The third connection is susceptible to the same type of accesstoken-related risk, but on a larger scale. This connection is used when the authorisation server sends an access token to the authentication server to check its validity. This means that every token of every user gets sent over this connection. This could result in a large scale abuse of the system if the tokens have a long validity duration.

The fourth connection can also be targeted by a MITM attack. It's, just like connection two, a point in the system where malicious code might be injected.

Last but not least, the dangers to the fifth connection. This connection is used to upload and manage apps and updates in the cloud server. If an attacker were able to inject code into this connection, or if an attacker got hold of the logindata of an app developer, he would be able to directly upload malicious code into the cloud service. This would result in a global distribution of the infected update to all app users. This simply cannot happen.

3.2 Security measures

The previous section listed numerous security risks which could happen if the system isn't properly secured. Luckily measures were taken in order to secure the system and its users. In fact, there's one measure that would provide security versus all the risks from the previous section. This measure is to ensure all communication between all the components happens by HTTPS. Attackers would not be able to listen to the connections because now all data sent is encrypted and malicious code injections will be detected by the servers and the mobile device since HTTPS supports tamper detection.

To enforce this measure, all the servers in the system are configured as HTTPS servers. This entails secure communication on all connections.

4 Conclusion

The goal was to create a flexible update system for Cordova applications. In the first step, a qualitative research was done about Cordova update plugins. This research resulted in the qualification of the CodePush update plugin as the foundation of the system. Next, two servers were added for authentication and authorisation purposes. The correct combination of these elements results in a system which can provide individualised updates to Cordova app users.

[2]https://www.npmjs.com/package/node-oauth2-server
[3]https://nodejs.org/en/
[4]https://expressjs.com/