Security Oracle - Developer Manual

July 7, 2016

Contents

Part I
About this manual

This manual discusses some of the implementation details of Security Oracle. It is intended for developers who wish to extend the original implementation and assumes that the reader is familiar with the intended use and installation of the individual components covered. More information on the subject can be found in the Security Oracle User Manual.

Security Oracle is built for use with Cloud Foundry applications and both this guide and the former assumes that the reader is familiar with Cloud Foundry. This guide also assumes basic familiarity with the structure of Cloud Foundry buildpacks.

Security Oracle Service and Security Dashboard are both written in NodeJS. This guide assumes that the reader is familiar with NodeJS application development.

Additional technologies may be pertinent on a project-by-project basis. These prerequisites will be discussed in each project's background and prerequisites section as needed.

Part II
Security Oracle Service

1 Background and Prerequisites

The Security Oracle Service is a Swagger compatible API, and is based on an automatically generated NodeJS server. You should be familiar with Swagger and the Swagger generator, or at least the structure of a generated NodeJS Swagger server before proceeding. You may read more at `http://swagger.io` or even generate your own Swagger server from a Swagger configuration file at `http://editor.swagger.io/#/`.

Security Oracle also uses Promises extensively. Promises are an abstraction over asynchronicity in JavaScript that offer more legible and composable semantics than traditional callbacks. If you're not familiar with Promises, it's recommended that you get familiar with them before approaching the Security Oracle codebase. More specifically, Security Oracle uses Bluebird - a fast, safe implementation of the Promise specification, that also comes with a great selection of utility and higher-order functions to help convert conventional callback-based functions to functions that return Promises, and to compose Promises in meaningful, powerful ways. It is highly recommended that you consult the Bluebird documentation (`http://bluebirdjs.com/docs/getting-started.html`) before proceeding or as you interact with the code.

The core solution finding algorithm has a concept of compatible packages that is based on the Semantic Versioning specification. If you are not familiar with this specification it may be helpful to consult `http://semver.org` although it may suffice to know that NPM packages follow a convention that makes it easy to know if two versions of a package are compatible with each other, and that we use this convention to select what we call "semver-compatible" (or simply "compatible") versions of packages.

2 File Structure

Security Oracle is based on a standard Swagger-generated NodeJS server. Like most other servers of this nature, the core business logic resides in Service files. Specifically the AppService.js, ReportService.js and SolutionService.js files under the controllers directory. Additionally, these services may use utility functions under the utils directory. These include the database layer in db.js, the solution finding algorithm in findSolutions.js and dependency tree transformation in processDeps.js.

3 The Core Algorithm

Security Oracle Service not only identifies security vulnerabilities in the user's application's dependency tree, but can suggest solutions to these problems. The main constraint is that a proposed solution should never break the user's application and any breakage that does occur should be easy to fix. Therefore, any new package version that is proposed as a solution must be, as far as we can tell, compatible with the version it replaces. Additionally, even as we minimize the risk of installing an incompatible package version, we must try to maximize the user's ability to work around incompatibilities if they arise. In concrete terms, this would mean we can only alter direct dependencies installed by the application developer, as these are packages that the application developer interacts with directly. The algorithm is as follows:

```
1. Set Vulnerabilities = {}
2. Traverse the dependency tree
3. For each node in tree
3.1     If node (package@version) is listed as vulnerable
3.1.1    Vulnerabilities = Vulnerabilities U { pathDownTreeTo(node) }
4. Set Solutions = {}
5. For each vulnerability in Vulnerabilities
5.1     Let directDependency = vulnerability[0]
5.2     Let solution = latestCompatibleVersion(directDependency)
5.3     If solution not vulnerable*
5.3.1    Solutions = Solutions U { solution }
```

In practice, steps 1-3 are performed by the third party NSP library. The dependency "tree" is not strictly a tree, as several packages may depend on a single other package, and if that package is vulnerable, it will have several different paths leading to it. LatestCompatibleVersion is implemented by querying the NPM package registry for the latest semver-compatible version, and checking if the solution is vulnerable (at step 5.3) entails traversing it's entire dependency tree and checking if any of those dependencies are vulnerable. The formats returned as output or expected as input for the various libraries involved are different, so additional transformations must be performed between steps.

This algorithm is implemented fully in utils/findSolutions.js

4 Implementation Notes

4.1 The App Endpoint

4.1.1 GET /:appId

Returns the list of all report identifiers associated with the application identified by appId, as stored in the database. This list is maintained by the POST method that follows.

4.1.2 POST

When an application is posted to this endpoint, it is analyzed according to the Core Algorithm. The vulnerabilities, solutions and other application metadata is stored in a report that is assigned a unique identifier and stored in the database under that identifier. The list of all reports associated with the posted application is updated to contain this newly generated report and saved to the database. A summary that contains the reportId and brief vulnerability status is returned from this endpoint.

4.2 The Report Endpoint

4.2.1 GET /:reportId

The report identified by reportId is returned from the database.

4.3 The Solution Endpoint

4.3.1 GET /:appId

Gets the list of accepted solutions for the application identified by appId, as stored in the database. This list is maintained by the POST method that follows.

4.3.2 POST /:appId

When a list of proposed solutions is posted to this endpoint, they are marked as accepted by the user. This list of accepted solutions can be retrieved using the GET method that precedes this. A solution is an opaque string as it appears in the solutions field of a report. In practice, the string may not be so opaque and may be human-readable and understandable. Regardless, it must be posted to this endpoint exactly as it appeared in the report.

5 The Database Layer

All services operate atop a simple database layer. It is assumed that this database is a key-value store with methods getAsync(key) and putAsync(key, value) where value may be any object. These methods should both return a Promise with the value for key. In the current implementation, this is achieved by wrapping the Level (`https://github.com/Level/levelup`) module which wraps the LevelDB (`https://github.com/google/leveldb`) key-value storage engine. This implementation can be easily replaced with a more powerful one (like mongodb, or even a conventional RDBMS) as long as it exposes the same interface.

Part III
Security Oracle NodeJS Buildpack

6 Background and Prerequisites

The Security Oracle NodeJS Buildpack is a fork of the Cloud Foundry NodeJS buildpack (`https://github.com/cloudfoundry/nodejs-buildpack`). It extends the stock buildpack's functionality by adding additional stages to the compile script.

7 File Structure

The main entry points for the new functionality are two new steps in the bin/-compile script; apply_solutions and query_oracle. These functions call functions that are defined in lib/dependencies.sh. These functions also delegate to NodeJS programs located at lib/apply_solutions.js and lib/extract_modules.js.

8 Implementation Notes

The functions implemented in bin/compile interact with the Security Oracle via curl, but leave the heavy lifting to scripts implemented in NodeJS. Wrapper functions around these NodeJS scripts are defined in lib/dependencies.sh. Any change in URLs or other transport related details will have to be made directly in the compile script. Changes in the generated data structure will likely be made in the JavaScript programs.

8.1 Apply Solutions

A list of solutions is downloaded from the Security Oracle, according to the application identifier extracted from the buildpack environment. Each solution is comprised of a package name and version. If the package exists in the package.json file, it's version is updated to reflect the one in the solution. This happens before the dependency installation phase, so that by the time the buildpack installs NPM packages, it will install the version specified in the solution.

8.2 Query Oracle

The buildpack runs 'npm ls -json' to produce a full, serialized dependency tree. This data is fed into extract_modules.js, which transforms the dependency tree format to conform to the Security Oracle Service API and adds application metadata extracted from the buildpack environment. This transformed and augmented data is sent to the Security Oracle Service for analysis, and the results are displayed to the user.

Part IV
Security Dashboard

9 Background and Prerequisites

Security Dashboard is a set of two jobs, two widgets and a dashboard configuration for AtlasBoard (`http://atlasboard.bitbucket.org`). It's recommended that you get acquainted with the development of Atlasboard components if you are not familiar with them already.

10 File Structure

The jobs and widgets are under the default package (packages/default). The file structure within the package is the standard structure mandated by AtlasBoard.

11 Implementation Notes

11.1 Vulnerabilities Job / Widget

The vulnerabilities job continually queries the Security Oracle Service for reports related to a configurable application identfier. Any reports that are not already in the cache are downloaded and saved to the cache. The widget then displays a list of deployed application versions, color coded by the vulnerability status. Applications that have at least one vulnerability will be colored red, while applications that have no vulnerabilities will be colored green.

11.1.1 Latest-Report Job / Widget

The latest-report job continually queries the Security Oracle Service for reports related to a configurable application identifier, and downloads the latest report unless it is already in the cache. The widget then displays this report to the user, along with proposed solutions. Solutions that were accepted by the user and submitted to the Security Oracle Service are dispatched directly from the user to the Oracle (via AJAX) without the Security Dashboard's involvement.