OSP MVP XMLtoSQL Design Documentation

Theodore Olsauskas-Warren – U5195918
August 2014

Introduction

This document serves to outline the design choices made in the creation of the XMLtoSQL component of the Oh SCAP Please (OSP) minimum viable product. It aims to help future developers, whether external or internal to the project, understand why certain decisions were made, where the programs strengths and weaknesses are, and where the included extension points are. Hopefully this document will sere to guide continued developed on the component of OSP, so that further releases such as the planned extended edition make effective use of the ground work laid here.

UML Primer

For those so inclined, a diagram of the overall UML structure of the XMLtoSQL component is included as an appendix to this document. It uses a very small subset of the available UML notation, and serves only as a conceptual guide, rather than a strict reference materiel. There are many reasons for this, firstly any concept of type has been omitted from the diagram. This is owing to the way in which Python handles types. The name of the variable in most cases should be enough to generate a good guess of the expected type.

Function names in italics are “abstract” functions (no such concept exists in Python, instead they are simply functions that throw exceptions should they ever be called and are thus expected to be overridden in the sub-classes). Function parameters that include the self parameter first are object instance specific, whilst those without it are declared as static. The exact nature of the “uses” relationship outlined by a filled in black arrow are omitted, in most cases they simply represent some kind of use relationship. The direction of the arrows indicates single direction knowledge about.

General Architecture

The overall process by which the XMLtoSQL component processes through can be broken down into two primary components. The extraction of audit result information from user specified XML files (generated by openSCAP) into Python object data structures, then the storage of that information into a PostgreSQL (PSQL) relational database. Controlling this process is the aptly named controller class, which also serves as the API of sorts that the XMLtoSQL component presents internally to the OSP program.

To begin this process the controller class is handed an options object, which includes information regarding the files to be input, as well as the database to connect to and whom to connect to it as. In the current release this options object is created by a simple parsing module. This module is not part of the XMLtoSQL component and instead presents a point of integration for the rest of the OSP project.

Once the controller object has received the options object it may be asked to begin the process of extraction and insertion, at this point it begins iterates over the input files one by one. With each file it attempts to match the type of file to an input file type it is aware of (for the MVP this consists of Open Vulnerability and Assessment Language (OVAL) and Extensible Configuration Checklist Description Format (XCCDF) files). It does this by asking each of the different collection classes it is aware of if it is capable of parsing the file. The abstract collections object, from which different file format implementations extend from, contains a function for performing this test. Each implementation then overrides it with an appropriate test. Once the controller object has found a collection that has indicated it can handle the input file, it asks that collection to extract the relevant data.

The data to be extracted is defined by the definitions class. Each collection object has a list of definitions to which they must append definition objects they have created and filled out with the relevant information. It is here that a level of consistency is enforced; it doesn't matter what the input file looks like, or how the collections object handles that file, as long as it creates the appropriate list of definitions it can be stored in the database correctly. Once the collection object has generated its list of definition objects from the input file its job it complete.

Once all files have been handled in the manner, the controller object then moves to phase two, database insertion. It performs this by creating a SQLInterface object which serves to both create the relevant relations as well as insert data into them. The SQLInterface class uses the psycopg2 library to interface with the PSQL database. At creation, the object is informed of the details of the database for which it is to connect to (the controller object is of course aware of these via the aforementioned options object).

The controller then asks the SQLInterface object to create the database schema, it does this by handing a copy of the Definition class to the SQLInterface object. Contained within the Definition class is a copy of the schema desired. This means that the shape of the database is contained in a single place, the definition class. After the relation has been created insertion begins, the Controller object hands each list of definitions from the collections objects to the SQLInterface for insertion. The interface then looks at the values contained within the definition and stores them in the database. The interface is aware of the values it is looking for as it again references the schema stored within the definition object. After all definitions have been inserted into the database the entire process is considered complete.

Points of Extensibility

Because of the nature of the input data, namely that it contains a large amount of information and may also be in different formats, the XMLtoSQL component of OSP has been designed with a focus of extensibility in two main areas. Namely, adding additional attributes to a definition, as well as including the capability to parse different input file formats.

Extending the definition to include extra information is a straightforward process as the definition class not only defines what information the collection classes should generate, but also how that information should be stored in the database relation. Thus extending it to include additional information is a simple matter of updating its schema and adding additional class wide variables. Of course doing this requires that every collection class in the program now include those additional variables. There is not concept present of only including some of the definition variables, all must be included, this is reflected in the schema defined within the definition; every field explicitly disallows a null entry.

This disallowing of partial definition constructions was an obvious choice given the small initial data collection it represents, but it is also intended that any extensions also follow this format. This requirement brings a level of homogeneity to the database. This means that the component of OSP responsible for generating a visualisation and exploration of the data stored by the XMLtoSQL component can rely on the simple idea that all fields are equally populated by all entires. It is expected that meeting this requirement will be simple enough for a significant number of extension, owing to the fact that the input specification files are so incredibly rich with available data. It should also be noted that different specifications of files share a large common information subset, it is primarily the presentation of that subset that differs between them.

The second considered point of extensibility is that of the addition of extra input file formats. This is facilitated by extending the “abstract” collections class, as the OVALCollection and XCCDFCollection classes do now (this is visible in the UML diagram available in the appendix). Enabling XMLtoSQL to handle extra file types is as simple as overriding the two functions, the first being a simple true or false identification function that lets the controller class know whether the collection object can handle a specific input file. This must obviously be a unique way of identifying the type of file a new collection extension can handle, so as to avoid any overlaps. It is expected that with significant structure differences between file inputs developing a function to uniquely identify a new one should be a trivial task.

The second function to be overridden is that of generating the definitions. As discussed earlier the type of information extracted from each type of file is static and defined in the definitions class. That is different collection objects must still generate the same type of information. The collection object is handed the root of the XML tree and is simple asked to generate a list of definitions. This can of course be achieved by any method. The included examples simply walk through the tree to pre-determined points defined by their specification to pull the required information out.

Deficiencies

The primary source of deficiencies in the MVP version of the XMLtoSQL component of OSP lie in two places; the front end input and database connectivity options, as well as in the logging and handling of errors. These deficiencies have both arisen from the same limitation, namely the isolation of development from the rest of the OSP components.

At the front end only a basic parsing system exists and thus this limits the connectivity options the module exposes. Currently only the database name and user can be specified, missing is any concept of a password, or any ability to customise the name of the relation created in that database (currently hard-coded to OSP). It is expected that when the modules are combined together into a coherent product, an extension to the input options will trickle down through the program, addressing these limitations.

The XMLtoSQL component is also quite basic in the way it handles errors or erroneous inputs. Errors are simply defined as belonging to a fatal or non fatal type of error. For example being unable to connect to the specified database is a fatal error and results in an error being printed to the standard output and the exit function being called (no error code is given). A non-fatal error, like the relation OSP already existing, or a primary key violation on a definition insertion, simple result in an (informative) error being printed to standard out and the program continuing. At the moment I believe this strikes a good balance between the ability to keep going despite errors, as well as maintaining consistency.

Obviously simply outputting errors to standard out is a non-optimal solution. Again, it is hoped that in the combination of different OSP modules that this will be addressed. Hopefully with the inclusion of a program wide logging module that can more gracefully handle errors, such as by recording them in a file for user perusal. For the MVP however, it was decided that such a module was not a strict requirement and the current plain method of informing the user “good enough”

Conclusion

In conclusion, this document outlined the basic operation of the XMLtoSQL component of the OSP program in the hop of documenting it so that developers, both internal and external may better understand its functionality, as well as the architectural designed aimed at facilitating increased functionality in regards to both the type of data collected, and the source of that data. Deficiencies were also addressed, along with a discussion of their source and the future plans to address them.

Appendix

The appendix of this document contains a (basic) UML diagram of the XMLtoSQL component of OSP. It should serve as a visual reference for the relationships discussed in this document.

OSP MVP XMLtoSQL UML Diagram