Abdelaziz FOUNAS
Rémi GATTAZ
Marwan HALLAL

Project report :
Extended services for iPOPO Python-based component model

I) Introduction

I.1) Context

iPOPO is a Python-based Service-Oriented Component Model (SOCM) based on Pelix, a dynamic service platform. They are inspired on two popular Java technologies for the development of long-lived applications: the iPOJO component model and the OSGi Service Platform.

The development of iPOPO is supported by IsandlaTech, in the context of the thesis of Thomas Calmant in the ERODS team, at the Laboratoire Informatique de Grenoble.

Pelix brings the concept of bundles in Python. A bundle is a module with a life cycle. It can be installed, started, stopped, updated and uninstalled. A bundle can declare a class acting as a bundle activator using the @BundleActivator decorator. This class will be instantiated by the framework and its start() and stop() methods will be called to notify the bundle about its activation and deactivation. When it is active, a bundle can register services. A service is an object implementing a specification and associated to a set of properties. A component will then be able to select and consume a service according to the specification(s) it provides and to its properties.

The components are a concept brought by iPOPO. A component, or component instance, is an object managed by a container. The container handles the interactions between the component and the Pelix framework. That way, the component contains only the code required for its task, not for its bindings with the framework. A component is an instance of a component factory, a class manipulated by iPOPO decorators.

I.2) Goal of the project

In IPOPO exists a component called “Remote shell”. Its activation in an application allow users to remotely connect to an the IPOPO shell and therefore manage the application. Using the remote shell, it is possible namely possible to install bundles, start components and stop components.

The connection to this shell is however clear. Using simple tcp sockets, anyone listening to the connection can see what is transmitting between the server and the client. Moreover, any user can connect to this shell as now password is required.
The goal of this project is to create a TLS version of the remote shell. This component will then allow the authentication of users and encrypt every messages exchanged between the client and the IPOPO application. This encryption will then allows the implementation of several commands in the shell. For example the upload and download of file. Not authorized as of today for security reasons.

II) Prototyping

When we started this project, we knew we would need to design and implement three components in IPOPO. A component to create a TLS Remote shell, a component to handle a set of certificates and a component to add certificate generation commands to the IPOPO shell. Among these three components, it was decided that we would start by the TLS Remote shell.

The reason behind that choice was simple. The component to add commands in IPOPO can not be created before the certificate handling one. And at that point, we didn't know for sure what this component needed to do as we didn't know what the TLS remote shell would require. Doing a prototype of this component was a good way to find out. It also allowed us to understand the way IPOPO works a bit better before trying to write more complicated components for the framework.

Designing this component and implementing it took us nearly 4 weeks which was a lot longer than we first expected. The main reason behind this length is that we had to learn how to use python at the same. Still, the design we made for the prototype was good so the exact same design will be used for the component that will be in the delivered version of the project.

The prototype was presented during the mid-presentation of the project and is a fully working TLS version of the remote shell.

Figure 1 : Network capture of a TLS remote shell message

III) TLS Remote shell and IPOPO Remote Client

A component “remote shell” already exists in IPOPO. Therefore, it was not needed to write a lot of code to implement the TLS version of the remote shell. By extending the current remote shell and overriding a few methods, we were able to create the TLS remote shell. The design we used can be found in the *“Annexe A : Simplified Class Diagram of TLS Remote”*.

Because we extend existing classes, our implementation of the TLS remote shell is very similar to the remote shell already in IPOPO. It is a TCP server, waiting for new connections and creating a thread for every new client. Each clients connecting to the IPOPO application are given a pelix shell and gain access to a prompt. Using that shell, it is then possible to run commands on the ipopo application. The only major difference between the remote shell implementation and our TLS remote shell is that we are using ssl wrapped sockets rather than simple sockets. Therefore, every messages between the client and the IPOPO application are encrypted.

To reduce as much as possible repetition of code, we however had to slightly change the implementation of the component IPOPO remote shell. A decision approved by our tutor.

To connect to the IPOPO remote shell, users could use any tcp client. Using netcat and telnet were both viable options. With our tls version, they however weren’t anymore. As the tcp sockets are ssl wrapped, classic tcp clients can not be used to connect to the TLS remote shell. Therefore, at the same time we were writting the TLS remote shell component, we were writing an application called “IPOPO Remote Client”. Written in python as well, this application can be used to connect to the IPOPO Remote shell or the TLS remote shell.

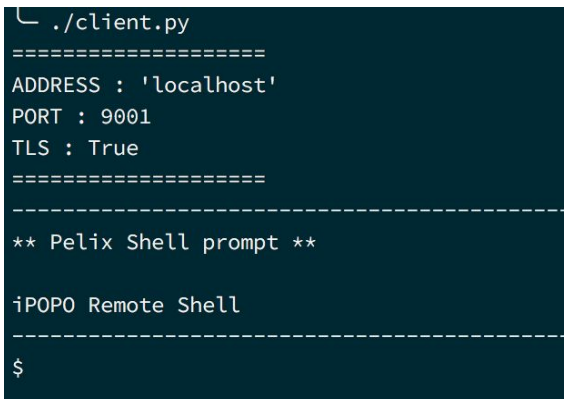

Figure 2 : IPOPO Remote Client

IV) Private Key Infrastructure

IV. 1) Choosing a library

The choice of an encryption library was necessary to advance the project. After doing some research of available solutions, we found two possible candidates:

- ssl : a python core module that is compatible with python 2 and 3.
- pyOpenSSL : a third-party wrapper of the OpenSSL crypto library.

The major limitation of ssl is that it doesn't handle in-memory storage of keys and certificates (keystores). A simple solution would be to use the pyjks module which provides keystore support. But this module is still in alpha and only supports python3. That's why we chose pyOpenSSL that has the inconvenient of adding a dependency to IPOPO.

IV. 2) Wrapping pyOpenssl

We decided to expose the features of pyOpenSSL through a set of wrapping modules. The idea behind this choice was to allow the future replacement of the library with another without affecting the rest of the application.

There are 4 wrapper modules:

- **Key** represents a pair of cryptographic keys (a private and a public key).
- **Certificate** is a wrapper of an X509 certificate
- **CSR** is a certificate signing request
- **Entity** is an organisation that uses a certificate; It can be either the issuer or the subject

These modules provide a public interface that is completely independent of the underlying logic. finally , the **PKI** module uses these wrappers to handle certificate authorities, including adding and revoking of certificates.

IV.3) Keystore

The interface KeyStore defines the needed methods to manage a KeyStore. A KeyStore manages Certificates and is configured from a path on the disk. It should give several primitives concerning the insertion, retrieval and removal of a Certificate and the dump and load of persistently saved KeyStore. Moreover, we need to

BasicKeyStore is the implementation of the latter interface. It defines how to manage a BasicKeyStore. Basically, it keeps the certificates in a dictionary and it dumps them on disk - thanks to the Pickle package - to save them persistently. It gives primitives to load a BasicKeyStore saved on disk.

V) Results

We had the time to do the prototype of a TLS client and the TLS secured remote shell. This prototype is working properly. A test with the Wireshark, a software capable of listening to network messages, shows a well-encrypted connection between the client and the remote shell.

We have also done the design and implementation of 6 different modules :

- Key
- Certificate
- CSR
- Entity
- KeyStore
- PKI

The goal of these modules was mainly to have an abstraction to handle certificates that could be used for the TLS remote shell. All of them have been tested by unit tests, and by the integration tests provided by TRAVIS.

But we are missing several points to complete the project. First, we did not manage to make the integration of our new modules in iPOPO itself. Since we didn’t do a component allowing the usage of PKI in the IPOPO shell, we cannot use the new functionalities in the iPOPO shell. It also involves the fact that the TLS remote shell component is still in its prototype state using static variables.

Also, it was first thought that we could have the time to design an access control solution for the remote shell. The goal here was to prevent or allow the use of certain

commands in the remote shell to specific users. But this objective has not been reached. But when the project will be finished, the implementation of such a functionality would be very useful for security issues.

VI) Conclusion

As stated in the previous part, we haven't reached our objectives. But since finishing this project is important, we are going to keep working on it to complete it. Furthermore, since the beginning of the project, a lab has approached Thomas CALMANT, the main developer of IPOPO, and asked for the inclusion of more security into IPOPO. This fact makes completing this project even more important than it already was.

So even though this project is still in an unfinished state, we will deliver everything we produced and provide all the help we can to Thomas CALMAN to finish this project.

Annexe A : Simplified Class Diagram of TLS Remote