© 2020 by the authors; licensee RonPub, Lübeck, Germany. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/4.0/).

Open Access

Open Journal of Web Technologies (OJWT)
Volume 7, Issue 1, 2020

www.ronpub.com/ojwt
ISSN 2199-188X

Securing J2EE SOA Enterprise Applications with a Pattern-Based Approach

Antonio Navarro, W. Eduardo Parra, Eduardo Romero, Sergio Martín, Rodrigo de Miguel

Dpto. Ingeniería del Software e Inteligencia Artificial, Universidad Complutense de Madrid,
C/ Profesor José García Santesmases, 9, 28040 Madrid, Spain
{anama, willparra, eduarrom, semart12, rodrimig}@ucm.es

ABSTRACT

Security is a key issue in SOA J2EE applications. The literature and a considerable number of frameworks address security issues for this type of enterprise application. However, there are two significant problems in this body of knowledge: (i) it is hard to find an architectural approach for dealing with security threats to SOA J2EE applications; and, (ii) technologies are constantly changing, making it is difficult to have an abstract view of the problems that are solved using specific technologies. The Core Security Patterns (CSP) catalogue solves both problems because it provides a comprehensive architectural solution to J2EE security issues and abstracts specific security technologies into security patterns. However, the CSP pattern catalogue is huge (more than 1,000 pages) and there are three significant challenges to understanding it completely: (i) the integration of the CSP security patterns and the Core J2EE Patterns (CJP) for the software architecture of SOA J2EE applications is not evident; (ii) the high abstraction level of the CSP patterns, in some cases, obscures the security problems that the patterns solve; and (iii) the implementation of the CSP patterns involves the configuration of complex security frameworks, adding a layer of complexity to securing a J2EE application using a pattern-based approach. To address these issues, we have developed a SOA multitier application based on the patterns described in the CJP catalogue, and we have secured it by implementing the patterns described in the CSP catalogue. This paper describes the work carried out during these developments. The main goal was to relate the CSP patterns with: (i) CJP patterns; (ii) the security concerns that the CSP patterns address; and (iii) the present security frameworks. As a result of this paper, we expect the inclusion of security elements in SOA enterprise applications to be easier for software architects and developers. Finally, four main conclusions can be drawn from our study: (i) security is an orthogonal aspect for SOA multitier development; (ii) implementation of security patterns relies heavily on security frameworks, with the configuration of security frameworks thus becoming one of the most complex issues when securing J2EE SOA multitier applications; (iii) no J2EE application servers are needed to deploy secure J2EE SOA enterprise applications; and (iv) whether or not applications servers are used, security-related implementations are closely tied to the application container and frameworks used for SOA implementation.

TYPE OF PAPER AND KEYWORDS

Regular Research Paper: *Web Services Security (WS-Security), Security Assertion Markup Language (SAML), Security Token Service (STS), Java Authentication and Authorization Service (JAAS), JavaServer Faces (JSF), Java API for XML Web Services (JAX-WS), Java API for RESTful Web Services (JAX-RS), Simple Object Access Protocol (SOAP), Representational State Transfer (REST)*

1 INTRODUCTION

Enterprise application development [1][26] is a complex issue that requires being an expert in one of its main development platforms (e.g. J2EE or MS .NET). These platforms have frameworks intended for the implementation of views using a controller and actions, the publishing of SOAP and REST web services, transaction management, object-relational mapping, and database connectivity, among others. In addition, these frameworks are built and structured around multitier design patterns [1][26]. They are designed as independent elements, taking into account the software architecture's tiers, but they have to work in an integrated manner according to multitier patterns to build enterprise applications. A good full stack developer must, therefore, be familiar with many frameworks and design patterns.

Although multitier patterns are, to some extent, independent of enterprise platforms, this paper focusses on J2EE, which was the platform chosen in the development of the applications built in our approach.

Security issues are not usually included in the literature on enterprise application patterns and frameworks. Thus, in the context of the J2EE platform, developers can master multitier architecture patterns [1][26], *JavaServer Faces* (JSF) [28], the *Java API for XML Web Services* (JAX-WS) [32], the *Java API for RESTful Web Services* (JAX-RS) [15], the *Java Persistence API* (JPA) [39], the *Java Transaction API* (JTA) [43] and *Java Database Connectivity* (JDBC) [25], but they may not know anything about security because it is not usually covered in the programing references for each framework or pattern catalogue.

Pattern-based design has the advantage of abstracting design problems by isolating them from concrete technologies [27]. Therefore, when including security issues in SOA multitier applications, a pattern-based approach would be desirable [24][71]. The *Core Security Patterns* (CSP) catalogue [71] is a complete and detailed work but is a complex book with more than 1,000 pages.

The CSP catalogue defines twenty-two patterns grouped into five categories: web tier, business tier, web-services, securing identity, and secure service provisioning. This is a major problem because the *Core J2EE Design Patterns* (CJP) catalogue [1], the most closely related catalogue to the CSP, groups its patterns into three tiers: presentation, business and integration. It is thus not obvious how to relate the CSP and CJP patterns because there is a significant mismatch between the classification schemata used in both catalogues.

Another problem with the CSP catalogue is, to some extent, a problem common to software design patterns catalogues: the abstraction and complexity of the patterns, in some cases, obscures the main problem that the patterns solve. This problem is amplified in the CSP catalogue because its vocabulary is not aligned with the security concerns presented in most J2EE security literature and frameworks.

Finally, security software is very difficult to implement because it involves high-level issues such as encryption algorithms and low-level issues such as bit transmission over networks. Therefore, in enterprise applications security software is not built from scratch, but based on security frameworks. Thus, one of the most complex issues in the implementation of security patterns is the configuration of these frameworks.

To address these questions, this paper analyzes the patterns included in the CSP catalogue from a practical point of view, driven by the security problems that must be addressed when securing SOA enterprise applications. This analysis also takes into account the current security-related J2EE frameworks and technologies needed for implementing the CSP patterns as well as the Core J2EE multitier patterns that are more closely linked to CSP patterns.

The work presented in this paper is based on two final degree projects developed in two consecutive academic years, 2017/18-2018/19 [48][50], which were preceded by a more theoretical final master's project [62]. The first final degree project developed a SOA multitier application that only had user authentication/authorization and invocation of web services using a username and password through HTTPS. The second project enhanced the first by including most of the security patterns described in the

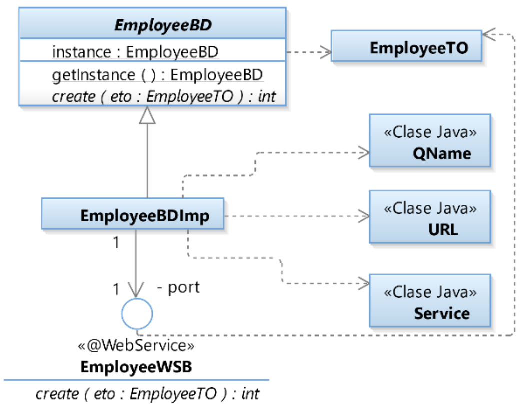

Figure 1: Business delegate (`EmployeeBD` and `EmployeeBDImp`), transfer (`EmployeeTO`) and JAX-WS classes in the WSC for the SOAP invocation of the WSP for employee creation. `EmployeeWSB` is the interface for the JAX-WS proxy for remote access to the web service broker deployed in the WSP

CSP pattern catalogue provided by Steel, Nagappan and Lai [71]. As a consequence of this approach, this study demonstrates that, if multitier architecture patterns are properly applied, security is an orthogonal aspect of SOA multitier development.

This paper has the following sections: Section 2 presents related work; Section 3 describes the architecture of the base SOA application; Section 4 describes the main security concerns that CSP patterns solve and how these patterns have been included in the base SOA application described in Section 3; Section 5 analyses the work and evaluations carried out; finally, Section 6 presents conclusions and future work.

2 RELATED WORK

This paper focusses on the J2EE platform. Therefore, security frameworks outside this platform have not been considered. However, this section also includes an analysis of security patterns that are independent of J2EE.

We have not found specific literature about securing J2EE applications beyond Buege et al. [14], Kumar [41] and Pistoia et al. [65]. Buege et al. focus on the different attacks that J2EE applications can be subject to and how J2EE security-related technologies can prevent them. Kumar and Pistoia et al. follow a more traditional approach, where J2EE security-related technologies are presented, as well as the security threats they prevent. These are good books focusing on J2EE security fundamentals (e.g. Java Security Architecture, class loaders, cryptography, etc.), HTTPS, securing RMI calls, XML security-related standards, securing EJB 2.x objects, and securing JAX-RPC web services. However, these books do not consider a base SOA multitier application and present the Java security technologies as technological elements that are added on. In addition, they do not follow a pattern-based approach to describe the SOA architecture of the underlying application or the security elements included in it. It is thus difficult to relate J2EE security technologies to multitier elements such as application services. They also fail to provide UML diagrams to give an abstract vision of the secure application. Finally, although some technologies presented in these books are still used, the lack of abstraction of the security principles in terms of security patterns makes them seem outdated (e.g. all the SOA security technologies are related to the overridden JAX-RPC web services framework).

In contrast, books that abstract security principles in terms of design patterns are more isolated from specific technologies and their changes over time. In this category of books, we have only found *Core Security Patterns* (CSP) [71] and *Security Patterns in Practice* (SPE) [24]. CSP considers five groups of patterns: web tier, business tier, web services, identity and service provisioning patterns. Although this book provides very interesting patterns, the way they are grouped makes their application difficult for multitier development, which uses three tiers (presentation, business and integration). In addition, the security patterns are not presented in terms of multitier patterns, making their implementation complex in a SOA multitier architecture. SPE is a complete catalogue of security patterns. This catalogue does not pay the same attention to basic security issues and directly defines the pattern catalogue. In addition, this catalogue considers some types of pattern that are not considered in CSP, such as those devoted to secure process management, secure execution and file management, secure operating system architecture and administration, or cloud computing. Therefore, this catalogue has less in common with the SOA multitier architecture than CSP.

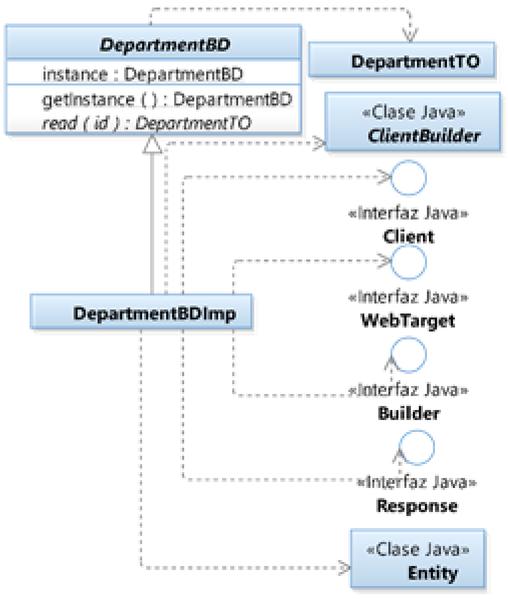

Figure 2: Business delegate (`DepartmentBD` and `DepartmentBDImp`), transfer (`DepartmentTO`) and JAX-RS classes in the WSC for the REST invocation of the WSP for reading a department

Alvi & Zulkernine [2] make a comparative study of software security pattern classifications, but they stress classification schemes instead of the use of patterns in software architectures.

Anand et al. [3] provide a comprehensive classification of security patterns according to the type of vulnerability they address. However, the paper does not focus on the practical application of these patterns to software architectures.

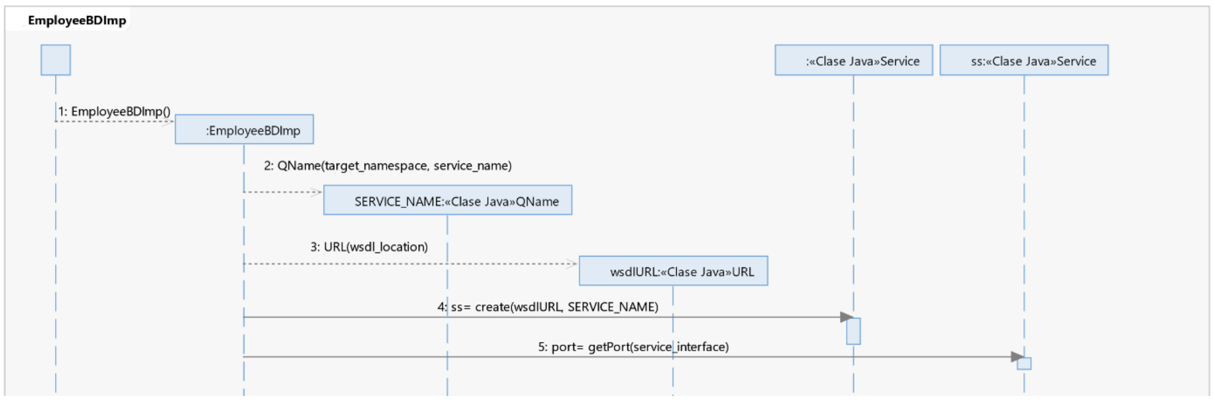
EmployeeBDImp
:«Clase Java»Service
ss:«Clase Java»Service
1: EmployeeBDImp()
:EmployeeBDImp
2: QName(target_namespace, service_name)
SERVICE_NAME:«Clase Java»QName
3: URL(wsdl_location)
wsdlURL:«Clase Java»URL
4: ss= create(wsdlURL, SERVICE_NAME)
5: port= getPort(service_interface)

Figure 3: Configuration in the JAX-WS WSC business delegate for accessing the SOAP WSP. In this example. `target_namespace= http://wsb.employee.business`, `service_name= EmployeeWSBService`, `wsdl_location= http://solaris.fdi.ucm.es:8080/csp/wsdl/employeewsb.wsdl`, and `service_interface= business.employee.imp.EmployeeWSB.class`

Cervantes et al. [19] agree on the importance of security patterns for securing enterprise software, but the paper focuses on analyzing security approaches in industrial and open-source projects and not on security patterns.

Hafiz et al. [29] is an interesting study relating several security patterns belonging to different catalogues. The key element of the paper is a diagram that depicts the relationships established between different patterns. However, these security patterns are not related to architectural patterns, and their inclusion in SOA multitier applications is thus not described.

Halkidis et al. [30] conduct an analysis of security patterns applied to a concrete use case focused on shopping processes. Although well described for this use case, the paper does not provide further examples for other design layers or for SOA architecture.

Mythily et al. [52] provide a method for including security elements in enterprise software that is designed in terms of activity diagrams. Although very useful, the method is restricted to designs made in terms of activity diagrams and it does not include design patterns.

Ryoo et al. [66] provide an architectural analysis method based on three techniques for analyzing security. The paper has an interesting section discussing related research in architectural analysis of security methods, but the whole paper is too abstract because it places its emphasis on analysis rather than on design.

Yoshioka et al. [75] do an extensive survey of security patterns, but the paper focusses on the classification of these patterns and does not provide detailed guidelines for their inclusion in software applications.

There are also catalogues on design patterns that do not focus on security issues such as:

- *Design Patterns: Elements of Reusable Object-Oriented Software* written by *Gang of Four* [27]. This is the seminal work on design patterns. With a focus on common patterns found in object-oriented frameworks, it does not provide either architectural or security patterns.

- *Core J2EE Patterns: Best Practices and Design Strategies* [1] and *Patterns of Enterprise Application Architecture* [26] present the key patterns for multitier architecture. The Alur et al. catalogue [1] is more orthodox, but Fowler [26] provides key information about concurrency management. In any case, neither emphasizes security issues. *J2EE Design Patterns* [21] presents similar patterns to those identified by Alur et al. [1].

- *SOA Design Patterns* [23] provides descriptive patterns for SOA software but also avoids security concerns.

- Wiley's series on *Pattern-Oriented Software Architecture* [16][17][18][40][68] includes five volumes that consider patterns similar to those defined in the above-mentioned pattern catalogues and two volumes specifically focusing on pattern languages. However, no special emphasis is put on security in this series.

Considering the above, additional work is needed to relate CSP security patterns, current J2EE security technologies and SOA multitier architectural patterns to make it easier to secure J2EE SOA applications.

It is well worth mentioning that this work cannot be understood without mastering multitier patterns and, in particular, *business delegate* and *web service broker patterns* [1], which are two of the most advanced patterns in this architecture. It is beyond the scope of this

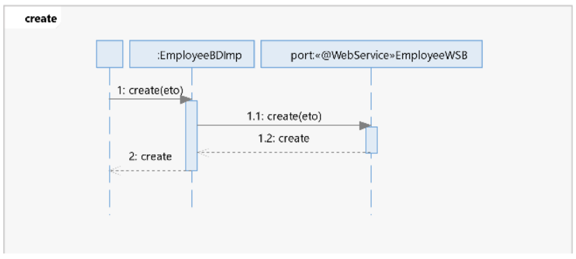

Figure 4: Access to the SOAP WSP in the JAX-WS business delegate using the JAX-WS proxy that implements the interface `EmployeeWSB`

paper to explain multitier and SOA architectures, but Navarro et al. [54] and Huertas & Navarro [33] are interesting references for both architectures that share the vocabulary and style used in this paper. Additionally, a command of JAX-WS and JAX-RS, the J2EE frameworks for managing SOA and REST web services, is also necessary to understand this paper.

Finally, securing J2EE applications involves some technologies that should be mentioned before continuing:

- The *X.509* standard for public key certificates [35] and the Java `keytool` command used to manage the *keystore*, a database of keys and certificates [60].
- *Security Assertion Markup Language* (SAML) [57] is an XML-based language that allows encoding authentication and authorization information.
- *eXtensible Access Control Markup Language* (XACML) [58] is an XML-based language that allows encoding information about access control policies in a declarative manner.
- *Web Services Security* (WS-Security) [56] is an extension to SOAP that allows programmers to sign, encrypt and attach security tokens to SOAP messages.
- Regarding web services, [59] is a very good reference for understanding web service security concepts.

3. SOA Base Application

Our work starts from a simple base SOA multitier application focused on the management of departments, employees and projects. Although this application includes almost all the patterns of the Alur et al. catalogue [1], in this paper we only focus on those necessary for applying security patterns. This application considers a *web service client* (WSC) with a JSF presentation tier connected to a business tier that consumes JAX-WS SOAP and JAX-RS REST web services. The *web service provider* (WSP) provides JAX-WS services for the management of employees and projects and JAX-RS services for the management of departments. Persistence, including transaction management, relies on JPA. Web services expose information using transfers instead of JPA entities. There are two reasons for this: (i) JPA entities contain cycles that cannot be marshalled unless MOXy [22] is used; and (ii) MOXy provides incomplete information with regard to the domain (i.e. a department can be obtained mutually linked with its employees, but if an employee is obtained, it is mutually linked to its department, which ignores its other employees).

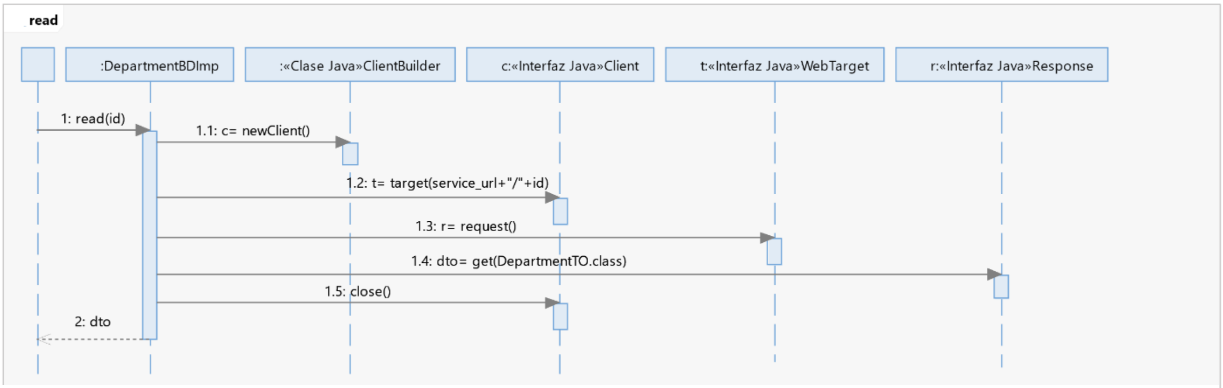
read
:DepartmentBDImp
:«Clase Java»ClientBuilder
c:«Interfaz Java»Client
t:«Interfaz Java»WebTarget
r:«Interfaz Java»Response
1: read(id)
1.1: c= newClient()
1.2: t= target(service_url+"/"+id)
1.3: r= request()
1.4: dto= get(DepartmentTO.class)
1.5: close()
2: dto

Figure 5: Access to the REST WSP in the JAX-RS business delegate using the JAX-RS classes for remote REST access. In this example, `service_url = http://solaris.fdi.ucm.es:8080/csp/services/department/wsb`

For the sake of conciseness, in this paper, we shall consider only two use cases of this application: the creation of an employee (which involves a JAX-WS web service) and the reading of a department (which involves a JAX-RS web service)[1]. Several operations, including create/read/update/delete operations for the management of all entities are implemented in the application.

Figures 1 and 2 depict the business tier of the WSC for the employee and department management. Business delegates and transfers are used according to a multitier architecture, as Huertas & Navarro describe [33]: business delegates represent remote services in client applications while hiding the connection details, and transfers provide object-oriented representation of data for their transfer between layers. The `EmployeeBD` and `EmployeeBDImp` classes depicted in Figure 1 are the SOAP business delegate for employee management. The `DepartmentBD` and `DepartmentBDImp` classes depicted in Figure 2 are the REST business delegate for department management.

Figure 3 describes the creation of the business delegate `EmployeeBDImp` class. This class implements (extends) the abstract class of the singleton (`EmployeeBD`) and configures the JAX-WS proxy Service for accessing the SOAP WSP as Hansen details [32].

Figure 4 describes the invocation of the WSP via the JAX-WS proxy made by the business delegate.

Figure 5 describes how the business delegate `DepartmentBDImp` (class that implements/extends `DepartmentBD`) uses the JAX-RS classes for invoking the REST WSP as Burke details [15].

Figures 6 and 7 describe the business tier of the WSP for the employee and department management. Web service brokers (WSBs), application services and transfers are used according to multitier architecture, as Huertas & Navarro [33] describe. WSBs publishes the services using SOAP or REST strategies and application services implement the business rules specified in the requirements. The `EmployeeWSB` class depicted in Figure 6 is the WSB for employee management. The interface `EmployeeWSB` stereotyped with `@WebService` in Figure 1 is the JAX-WS interface used in the WSC for characterizing the JAX-WS proxy for remote access to the `EmployeeWSB` class deployed in the WSP [32]. For the sake of conciseness, these JAX-WS interfaces used in SOAP WSC business delegates are omitted from the deployment diagrams, such as in Figure 8.

Finally, Figure 8 depicts the deployment diagram for both, WSC and WSP. Windows 10 Pro 64, JRE 1.8.11, and Apache CXF 3.2.0 [4] are the main elements depicted. Other execution environments and components, such those related to the database management, have not been included for the sake of conciseness.

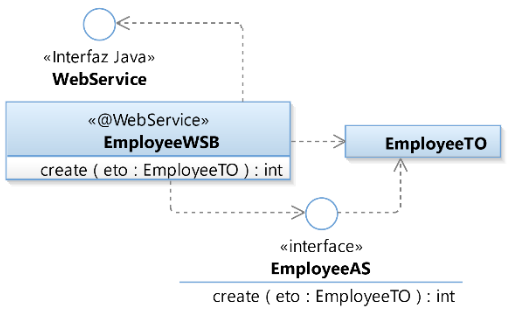

Figure 6: Web service broker (`EmployeeWSB`), application service interface (`EmployeeAS`), transfer (`EmployeeTO`) and JAX-WS classes in the WSP for the SOAP implementation of the web service for reading employee creation

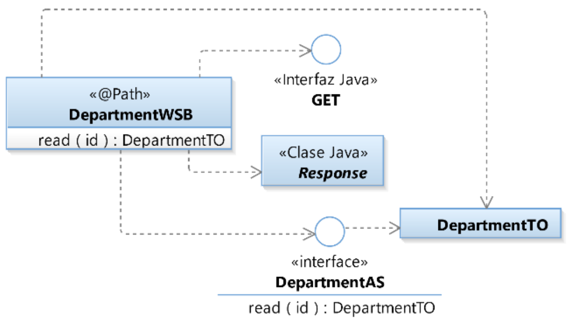

Figure 7: Web service broker (`DepartmentWSB`), application service (`DepartmentAS`), transfer (`DepartmentTO`) and JAX-RS classes in the WSP for the REST implementation of the web service for reading a department

[1] WSC code from the base application can be found here `https://github.com/hunzaGit/TFG_cliente` and WSP code from the base application can be found here `https://github.com/hunzaGit/TFG_server`

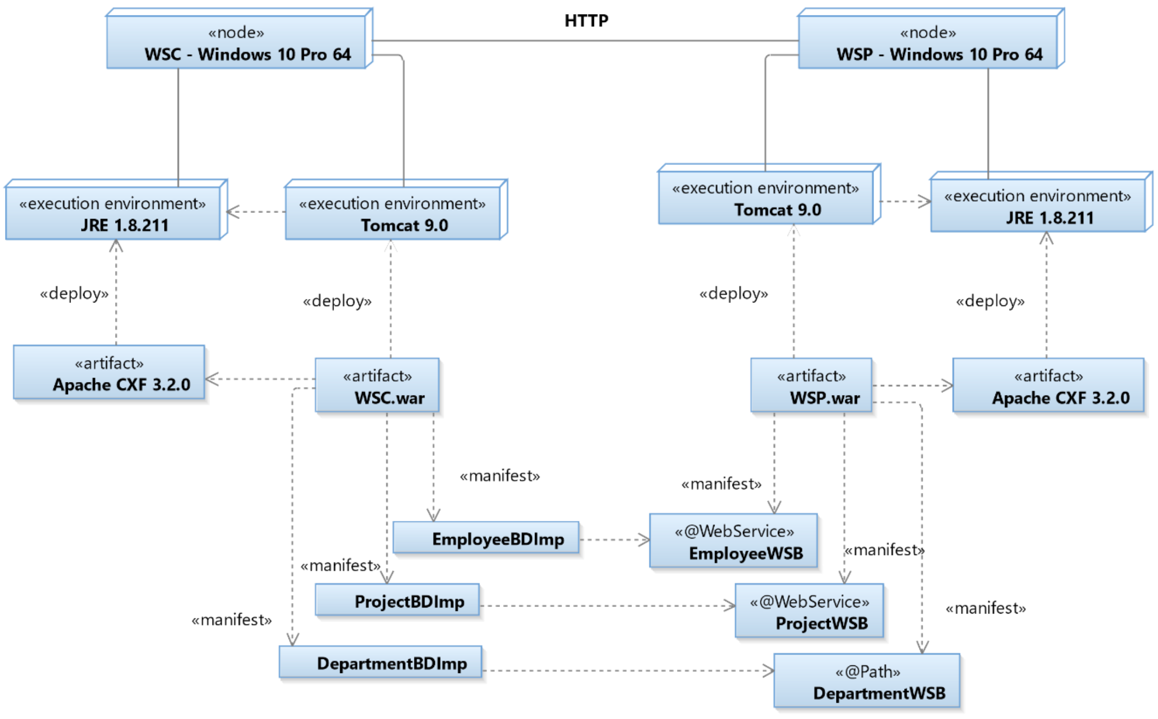
HTTP
«node» WSC - Windows 10 Pro 64
«node» WSP - Windows 10 Pro 64
«execution environment» JRE 1.8.211
«execution environment» Tomcat 9.0
«execution environment» Tomcat 9.0
«execution environment» JRE 1.8.211
«deploy»
«deploy»
«deploy»
«deploy»
«artifact» Apache CXF 3.2.0
«artifact» WSC.war
«artifact» WSP.war
«artifact» Apache CXF 3.2.0
«manifest»
«manifest»
EmployeeBDImp
«@WebService» EmployeeWSB
«manifest»
«manifest»
ProjectBDImp
«@WebService» ProjectWSB
«manifest»
«manifest»
DepartmentBDImp
«@Path» DepartmentWSB

Figure 8: Deployment for WSC and WSP. Services for employee and project management are published as JAX-WS SOAP web services (`EmployeeWSB` and `ProjectWSB`) and services for department management are published as JAX-RS REST web services (`DepartmentWSB`)

4. Rationale for Use of CSP Security Patterns in the Context of a SOA J2EE Application

This section aims to explain the use of CSP patterns in combination with CJP patterns, taking into account the security requirements that need to be addressed, and the current J2EE security frameworks. To this extent, we enhance the architecture of the base SOA application described in Section 3 with CSP security patterns that implement different security requirements[2].

As previously mentioned, the configuration of security frameworks is one of the most complex issues when implementing security patterns. Therefore, in this section many deployment diagrams describing the classes and artefacts deployed are provided, including only the key elements for each pattern in each case. We have provided deployment diagrams because, in our opinion, when frameworks configurable by text files are used, deployment diagrams are the best choice for defining these configuration files as well as the libraries used. Certainly, UML class and interaction diagrams are the best way to characterize the use of these frameworks, but this is beyond the scope of this paper and can be found in the references provided throughout. Thus, class and interaction diagrams are only included in this paper when they are strictly necessary to understand the implementation of CSP patterns.

4.1 Authentication and Authorization in the WSC Application: CSP Authentication and CSP Authorization Enforcer

According to the CSP catalogue [71]:

- *Authentication enforcer* creates centralized authentication enforcement that performs authentication of users and encapsulates the details of the authentication mechanisms.

[2] WSC code from the secured application can be found here `https://github.com/sergiomgm/TFG_cliente` and WSP code from the secured application can be found here `https://github.com/sergiomgm/TFG_servidor`

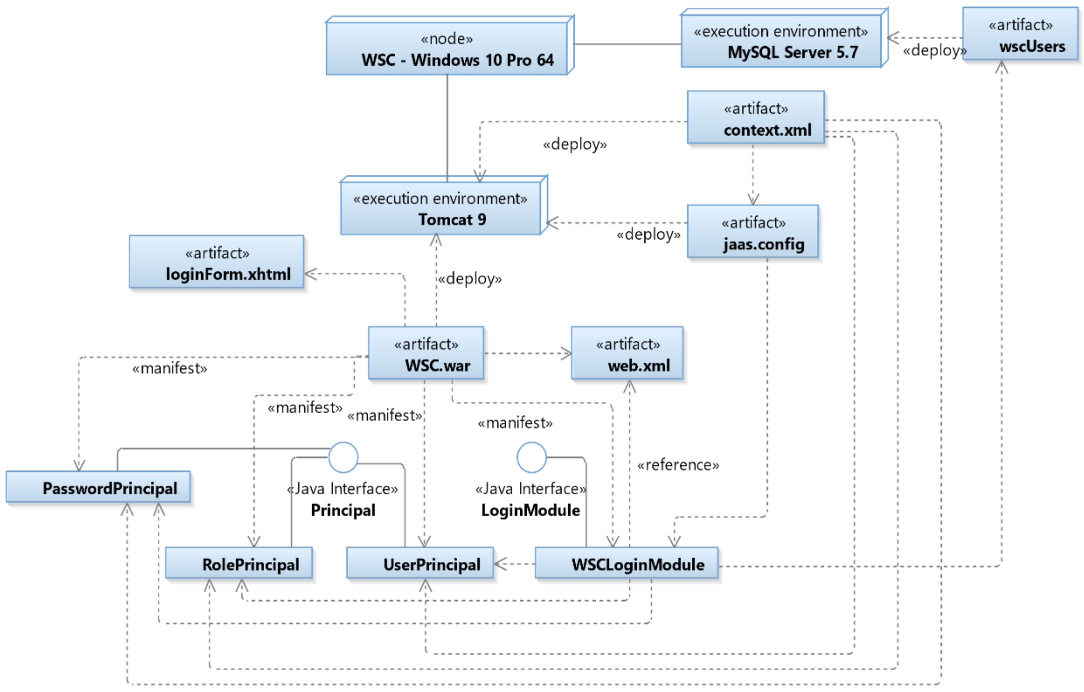

Figure 9: JAAS deployment in the WSC

- *Authorization enforcer* creates an access controller that performs authorization checks using standard Java security API classes.

This section explains how to use them to perform authentication and authorization in the WSC.

WSCs are usually web applications with some type of web graphical user interface in the presentation tier. Thus, users interact with the WSC using a browser. It is very common to have some type of access control to parts of the WSC application by providing a username and password (e.g. customer accounts). Two aspects must be checked to grant user access: (i) to check the authenticity of the users, i.e., *authentication*; and (ii) to check that the authenticated user has access to the requested resource, i.e. *authorization*. The components that deal with authentication and authorization in the CSP catalogue are named authentication and authorization enforcer respectively.

Java Authentication and Authorization Service (JAAS), is the J2EE framework that deals with user authentication and authorization in J2EE applications [61]. The use of JAAS with Tomcat is explained by Marques [45][46][47].

To use JAAS, programmers have to provide user, role and password Principals and a `LoginModule` that creates and validates them as Figure 9 depicts. A *principal* is an entity that is granted security rights [14]. Tomcat automatically calls a predefined login form (`loginForm.xhtml` in Figure 9) to enforce user authentication and validation when a protected page is requested and no user and role principals are present. After user and password are provided in this form, Tomcat calls the implementation of the `LoginModule`, which validates users, for example, using a relational database (table `wscUsers` in Figure 9).

Deployment of JAAS enforcers are a bit cumbersome and, in addition to the code provided (user and role principals and login module) several files have to be modified and/or provided:

- The application's roles and access rights to web pages have to be defined in the application's `web.xml` file.
- A file (e.g. `jaas.config`) has to be provided to tell Tomcat the available login modules.
- The bound of specific applications to concrete user, role and password principals is made in Tomcat's `server.xml` file, or in a Tomcat's context file (e.g. `context.xml`).

Figure 9 gathers all the elements needed to make JAAS work. It is well worth noting the web of dependences between the components that JAAS needs

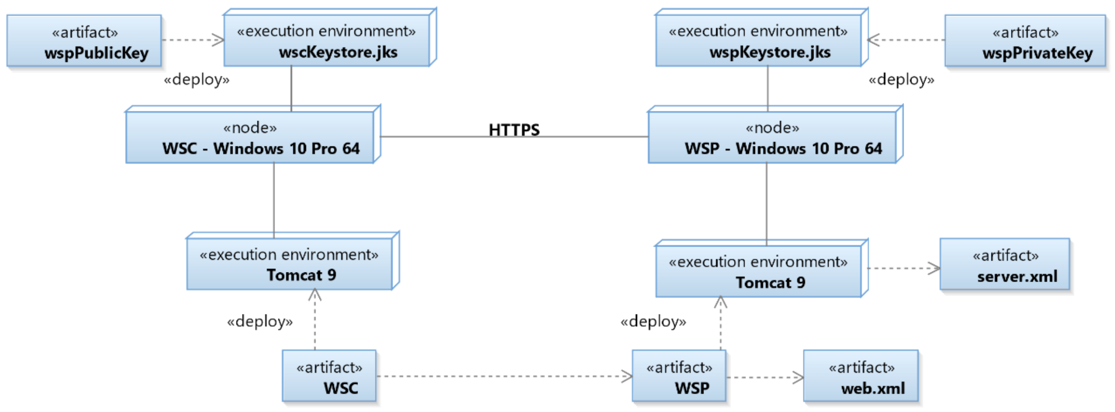
«artifact» wspPublicKey
«deploy»
«execution environment» wscKeystore.jks
«execution environment» wspKeystore.jks
«deploy»
«artifact» wspPrivateKey
«node» WSC - Windows 10 Pro 64
HTTPS
«node» WSP - Windows 10 Pro 64
«execution environment» Tomcat 9
«execution environment» Tomcat 9
«artifact» server.xml
«deploy»
«deploy»
«artifact» WSC
«artifact» WSP
«artifact» web.xml

Figure 10: Deployment for secure pipe between WSC and WSP

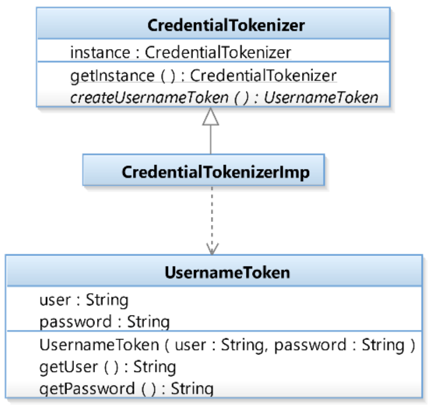
CredentialTokenizer
instance : CredentialTokenizer
getInstance ( ) : CredentialTokenizer
createUsernameToken ( ) : UsernameToken
CredentialTokenizerImp
UsernameToken
user : String
password : String
UsernameToken (user : String, password : String)
getUser ( ) : String
getPassword ( ) : String

Figure 11. Simplified credential tokenizer for username tokens

for it to be used, which can be a bit annoying when dealing with them for the first time.

The application developed uses traditional authentication based on username/password, but JAAS also supports biometric authentication as [53] depicts.

4.2 HTTPS Connections: CSP Secure Pipe

According to the CSP catalogue, *secure pipe* guarantees the integrity and privacy of data sent over the wire [71].

This section explains how to use it to establish HTTPS connections.

A common security problem in enterprise applications is unauthorized access to information sent between two points, the so-called *man-in-the-middle* (MITM) attack [71]. The most common way to avoid this problem is to encrypt the information sent between two points. In the context of HTTP connections, this solution takes the form of *HTTP Secure* or *HTTPS* [71]. The CSP catalogue characterizes HTTPS connections as the secure pipe pattern.

A detailed description of HTTPS is outside the scope of this paper, but in order to make it work two X.509 certificates are needed: the server's public key, used to encrypt the information, and the server's private key, for decryption [13].

The mechanism for describing the creation of X.509 certificates, their deployment in keystores, and the Tomcat configuration to make it work is explained in [38][3].

The WSP stores its private key in its keystore and the WSC stores the WSP public key in its own keystore in a correct configuration. For the sake of simplicity, no differences between *keystore* and *truststore* have been made in the figure. Using a WSC truststore to store a WSP public key would have been more accurate [36].

Once certificates have been created and developed, the `web.xml` file web of applications needs to be configured according to [34] to establish SSL connections. Tomcat also has to be configured to support these connections via the `server.xml` file [34].

Figure 10 depicts a typical deployment for a WSC accessing a WSP using an HTTPS connection.

[3] It is explained better in the first edition than in the second edition.

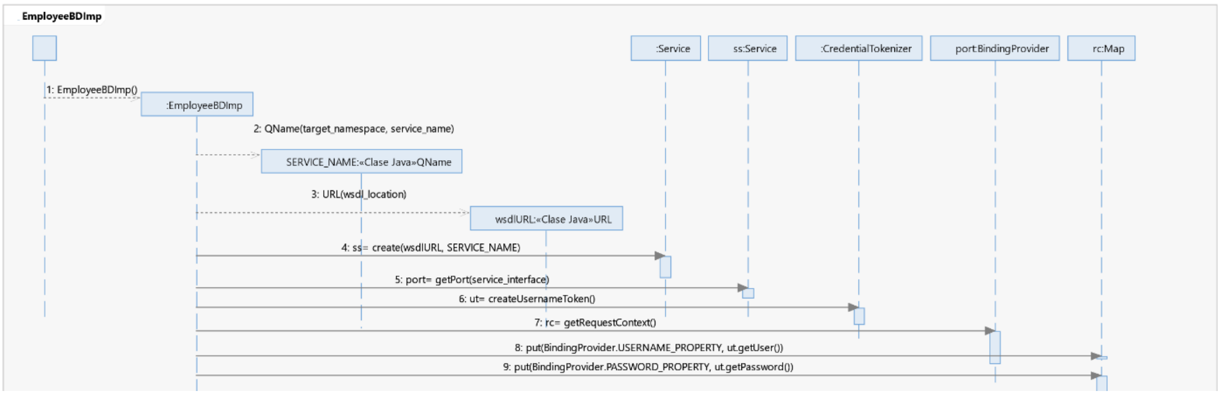
EmployeeBDImp
:Service
ss:Service
:CredentialTokenizer
port:BindingProvider
rc:Map
1: EmployeeBDImp()
:EmployeeBDImp
2: QName(target_namespace, service_name)
SERVICE_NAME:«Clase Java»QName
3: URL(wsdl_location)
wsdlURL:«Clase Java»URL
4: ss= create(wsdlURL, SERVICE_NAME)
5: port= getPort(service_interface)
6: ut= createUsernameToken()
7: rc= getRequestContext()
8: put(BindingProvider.USERNAME_PROPERTY, ut.getUser())
9: put(BindingProvider.PASSWORD_PROPERTY, ut.getPassword())

Figure 12. Configuration in the JAX-WS WSC business delegate for accessing the SOAP WSP using username and password obtained from a credential tokenizer. In this example, the values of `target_namespace`, `service_name`, `wsdl_location`, and `service_interface` coincide with those of Fig. 3

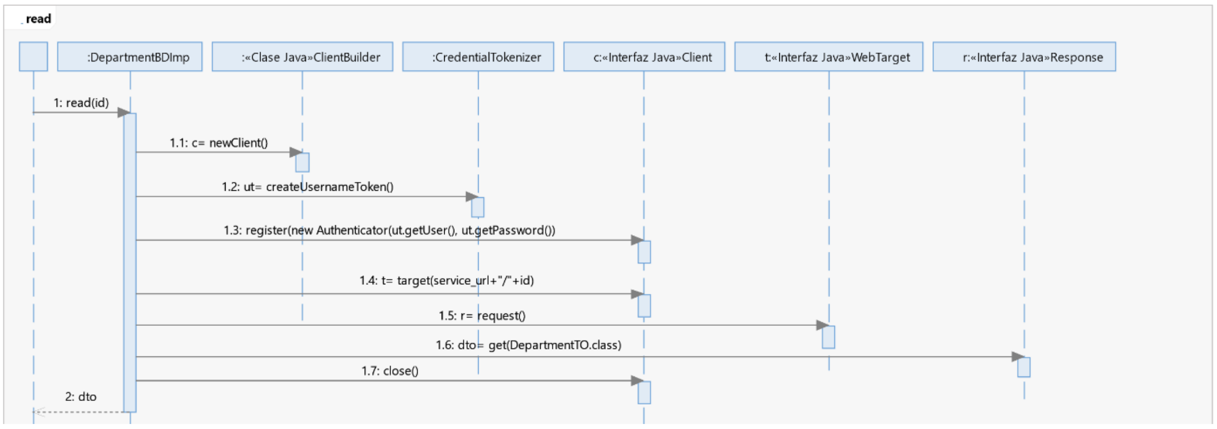
read
:DepartmentBDImp
:«Clase Java»ClientBuilder
:CredentialTokenizer
c:«Interfaz Java»Client
t:«Interfaz Java»WebTarget
r:«Interfaz Java»Response
1: read(id)
1.1: c= newClient()
1.2: ut= createUsernameToken()
1.3: register(new Authenticator(ut.getUser(), ut.getPassword())
1.4: t= target(service_url+"/"+id)
1.5: r= request()
1.6: dto= get(DepartmentTO.class)
1.7: close()
2: dto

Figure 13: Access to the REST WSP in the JAX-RS business delegate using username and password obtained from a credential tokenizer. In this example, the value of `service_url` coincides with that of Fig. 5

4.3 Point-to-Point Secure Invocation of WSP: CSP Secure Pipe, CSP Secure Session Object, CSP Container-Managed Security and CSP Credential Tokenizer

According to the CSP catalogue [71]:

- *Secure session object* abstracts encapsulation of authentication and authorization credentials that can be passed across boundaries.
- *Container-managed security* defines application-level roles at development time and perform user-role mappings at deployment time or thereafter.
- *Credential tokenizer* encapsulates different types of user credentials as a security token that can be reused across different security providers.

This section explains how to use them to make point-to-point secure invocation of WSPs.

Section 4.2 presented CSP secure pipe as the way to establish HTTPS encrypted channels for web communication. In the context of web applications, HTTPS connections are established between browsers and servers. In the context of SOA applications, HTTPS have to be established between the WSC and the WSP. According to the CSP catalogue, four patterns are needed to establish HTTPS between them:

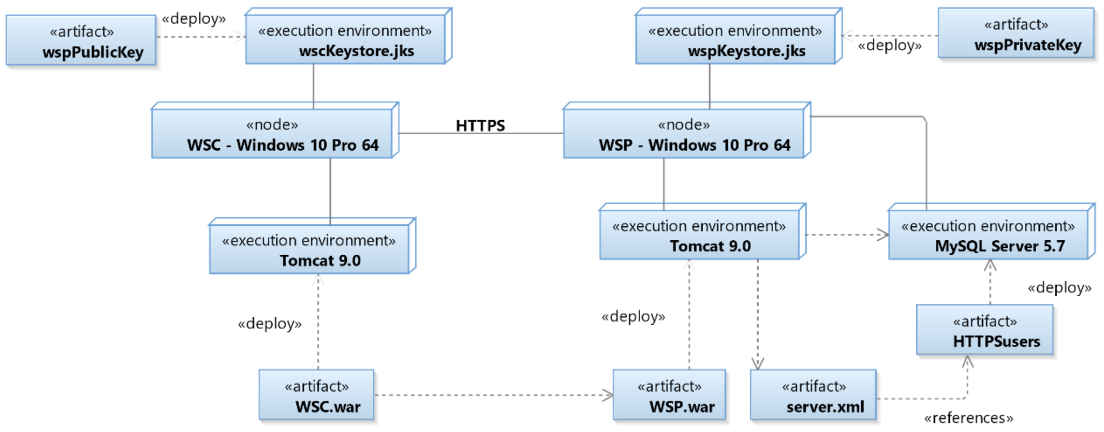
«artifact» wspPublicKey
«deploy»
«execution environment» wscKeystore.jks
«execution environment» wspKeystore.jks
«deploy»
«artifact» wspPrivateKey
«node» WSC - Windows 10 Pro 64
HTTPS
«node» WSP - Windows 10 Pro 64
«execution environment» Tomcat 9.0
«execution environment» Tomcat 9.0
«execution environment» MySQL Server 5.7
«deploy»
«deploy»
«deploy»
«artifact» HTTPSusers
«artifact» WSC.war
«artifact» WSP.war
«artifact» server.xml
«references»

Figure 14: Deployment for point-to-point secure invocation of WSP. The configuration is valid for both SOAP and REST services

- Secure pipe: to physically establish the HTTPS connection.
- Container-managed security: to enforce containers (e.g. Apache Tomcat) to check that a request includes username and password. This prevents users from programming specific code for user authentication and authorization.
- Credential tokenizer: to encapsulate a security token in different formats, such as username/password or X.509.
- Secure session object: to gather username and password information.

It is worth noting that:

- Secure pipes can be established without usernames and password. This avoids the problem of MITM attacks, but anyone with access to the WSP could invoke it.
- Secure session objects and container-managed security enforce the authentication/authorization of the WSC before invoking the WSP, but do not prevent MITM attacks.

Therefore, the presence of secure pipe and container-managed security patterns is needed to make point-to-point secure invocations of WSP. Credential tokenizer is optional, but in this use case we introduce it as credential holder. Figure 11 shows the simplified credential tokenizer used in this project for the management of username tokens, which contains username and password.

The inclusion of the username and password in the JAX-WS WSC is as simple as including them in a map obtained from the JAX-WS class `javax.xml.ws.BindingProvider`. This is the only change needed in the code, since WSP does not check them because container-managed security is used. Thus, the secure session object is implemented as the map that contains username and password and is internally managed by Tomcat and JAX-WS (and JAX-RS also). Figure 12 modifies the interaction of Figure 3 to include username and password (obtained from a credential tokenizer) in the invocation to the SOAP WSP.

The inclusion of username and password in the JAX-RS WSC is a bit more complex, because the JAX-RS interface `javax.ws.rs.client.ClientRequestFilter` has to be implemented with the class `Authenticator` to codify this information in the request as exemplified by Bien [12]. Figure 13 modifies the interaction of Figure 5 to include username and password (obtained from a credential tokenizer) in the invocation of the REST WSP.

Tomcat greatly facilitates the implementation of container-managed security and the secure session object. Only the `server.xml` file has to be modified to define a `Realm` that enforces HTTPS connections to provide username and password. This information can be checked against different information containers, such as tables in relational database management systems as [5] describes (table `HTTPSusers` in Figure 14).

Figure 14 depicts the components needed for the implementation of container-managed security, as well as for a secure pipe.

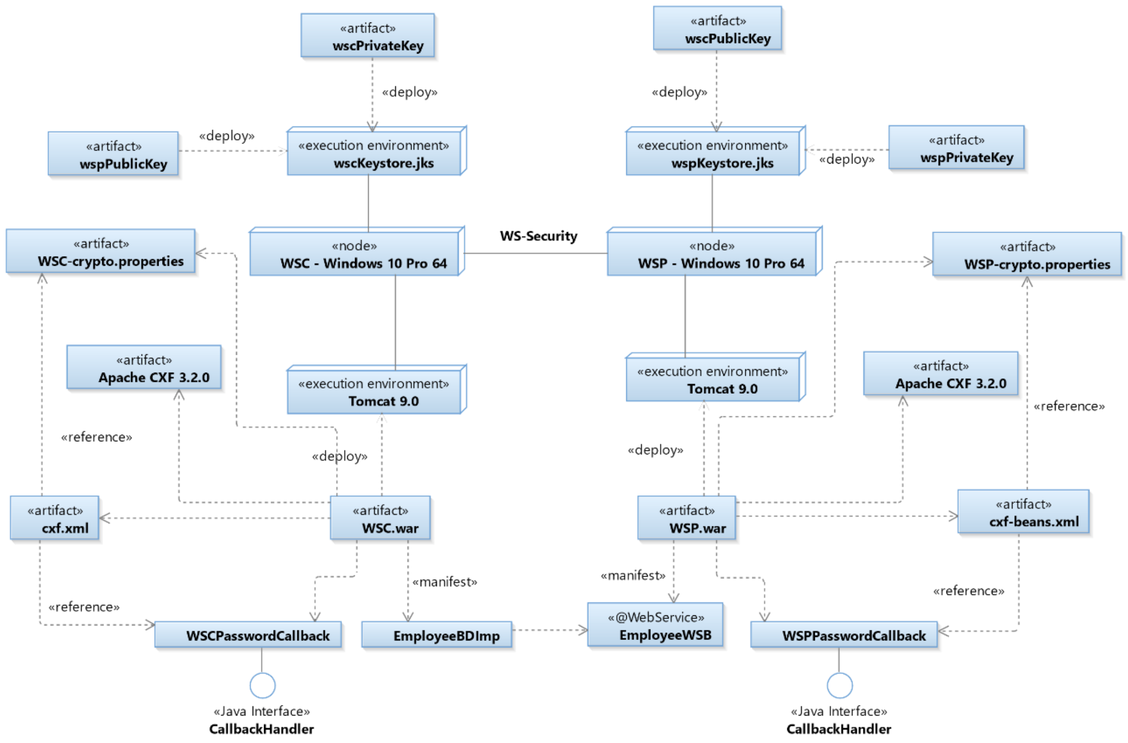
«artifact» wscPrivateKey
«artifact» wscPublicKey
«deploy»
«artifact» wspPublicKey
«execution environment» wscKeystore.jks
«execution environment» wspKeystore.jks
«artifact» wspPrivateKey
«artifact» WSC-crypto.properties
«node» WSC - Windows 10 Pro 64
WS-Security
«node» WSP - Windows 10 Pro 64
«artifact» WSP-crypto.properties
«artifact» Apache CXF 3.2.0
«execution environment» Tomcat 9.0
«reference»
«artifact» cxf.xml
«artifact» WSC.war
«artifact» WSP.war
«artifact» cxf-beans.xml
«manifest»
WSCPasswordCallback
EmployeeBDImp
«@WebService» EmployeeWSB
WSPPasswordCallback
«Java Interface» CallbackHandler

Figure 15: Deployment for WS-Security

4.4 End-to-End Secure Invocation of WSP with WS-Security: Not Considered in the CSP Catalogue

Point-to-point security has two major drawbacks: (i) if intermediate gateways are used, these gateways need access to the WSP private key; (ii) and, because encryption is reliant on the transport layer, instead of the application layer, the application's security is extremely dependent on the server's infrastructure, which the developers of the application may not control (i.e., the application can be hosted in external servers).

To overcome both drawbacks, end-to-end security can be achieved by encrypting the information in the application layer instead of in the transport layer, as HTTPS does. To this extent, WS-Security is a SOAP extension that provides integrity, confidentiality and identity credentials to SOAP messages, allowing the encryption and signing of parts of these messages in the application layer [56].

Although very powerful, WS-Security has two major drawbacks: (i) it is not available for REST web services; and (ii) it provokes a significant server overload. Therefore, the use of point-to-point or end-to-end security is an issue that has to be carefully analyzed before taking a decision.

It is worth mentioning that, although the CSP catalogue presents WS-Security, it does not define a pattern for it.

WS-Security can be used with or without secure pipe because SOAP messages (or their confidential elements) are encrypted by the WSC before sending them to the WSP (in the example in Figure 15, HTTPS is also used). The configuration for using WS-Security, regarding certificates and keystores, is very similar to the configuration for using secure pipes. However, in the context of SOA enterprise applications, where information could be encoded in both directions, we need to include the WSP public key and the WSC private key in the WSC keystore. Reciprocally, we need to include the WSC public key store and the WSP private key in the WSP keystore.

However, the key question remains unanswered: how do we encode our SOAP messages using public keys and decrypt them using private keys? JAX-WS does not provide any specifics regarding this. Thus, the encryption of the SOAP messages has to be encoded using the underlying implementation of JAX-WS. In the case of Apache CXF, *WSS4J interceptors* must be used [6][7][44]. *WSS4J Project* is the framework that provides a Java implementation of the primary security standards in the WS-Security specifications [8].

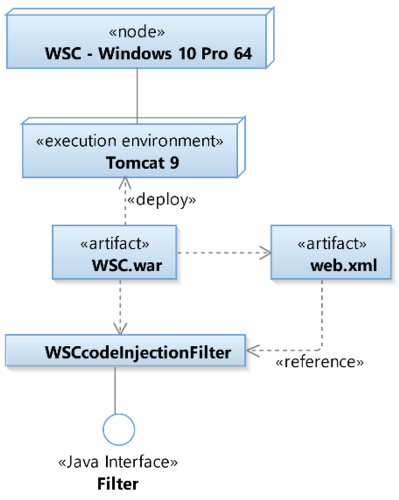

Figure 16: Deployment for intercepting validators in the WSC

Programmers only have to write code for WSC and WSP password callback classes implementing the `javax.security.auth.callback.CallbackHandler` interface. These callback handlers provide access to the WSC and WSP private keys used for signing and decryption. In addition, configuration files (`WSC-crypto.properties` and `WSP-crypto.properties`) have to be provided, both in the WSC and in the WSP, for defining keystore location, password and default alias. Private keys defined in callback handlers could be included in these configuration files, but, because text files can be more easily hacked than Java classes, this is considered a less secure practice [44].

In addition, CXF has to configure output/input interceptors in the WSC and input/output interceptors in the WSP. The WSC uses a configuration file (e.g. `cxf.xml`) that defines a `conduit` (i.e. URL for accessing WSP) that configures output/input interceptors for SOAP encryption/decryption [6][44]. Thus, the WSC, the `EmployeeBDImp` has to obtain its Java access proxy for accessing the WSP using the conduit named `https://solaris.fdi.ucm.es:8443/csp/services/EmployeeWSBPort`, which is configured for providing WS-Security in the WSC using the `cxf.xml` configuration file.

The WSP uses a CXF configuration file (`cxf-beans.xml`), loaded when Tomcat starts the WSP, which configures the service `endpoint` (i.e. the implementation of the service), with input/output interceptors for SOAP decryption/encryption [7][44]. Thus, the WSP endpoint, the `EmployeeWSB`, which implements the `https://solaris.fdi.ucm.es:8443/csp/services/EmployeeWSBPort`, is configured for providing WS-Security in the WSP using the `cxf-beans.xml` configuration file.

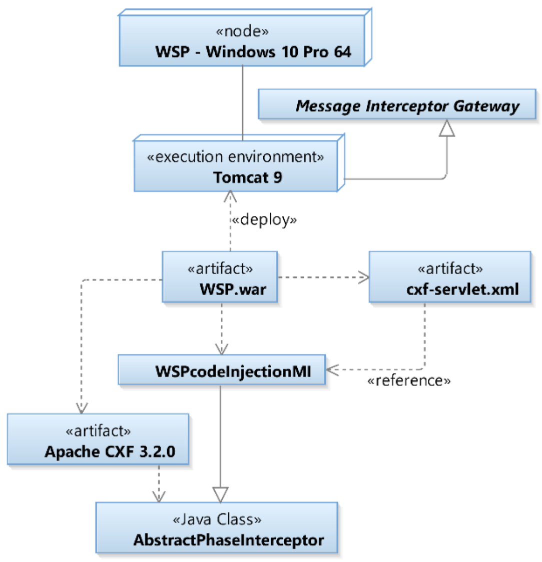

Figure 17: Deployment for MIG and MI in the WSP. In the figure, Tomcat plays the role of message interceptor gateway, i.e., `Message Interceptor Gateway` is not a class belonging to any framework

Figure 15 depicts the deployment of WSC and WSP to support WS-Security.

4.5 Avoiding Code Injection Attacks in the WSC: CSP Intercepting Validator

According to the CSP catalogue, *intercepting validator* cleanses and validates data prior to its use within the application, using dynamically loadable validation logic [71].

This section explains how to use it to prevent code injection attacks in the WSC.

As with any web application, WSCs can be subject to different types of attacks, such as code injection or denial of service (DoS) [71]. The simplest way to avoid this is to filter all the incoming requests and try to find a pattern that could identify an attack. CSP proposes using intercepting validators for detecting this type of attacks. Thus, CSP intercepting validators are multitier *intercepting filters* [1] used for security purposes.

In practice, CSP intercepting validators are servlet filter classes [31] that implement `javax.servlet.Filter` interface. These filters are referenced from `web.xml` file. Figure 16 depicts a deployment diagram for intercepting validators.

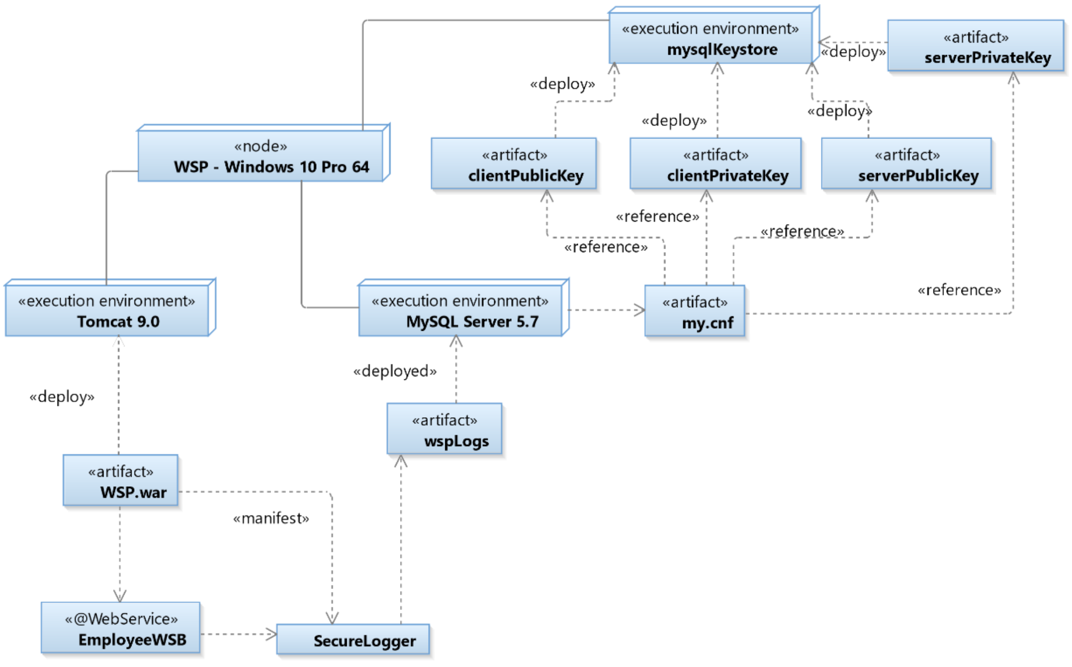
«execution environment» mysqlKeystore
«artifact» serverPrivateKey
«deploy»
«node» WSP - Windows 10 Pro 64
«artifact» clientPublicKey
«artifact» clientPrivateKey
«artifact» serverPublicKey
«reference»
«execution environment» Tomcat 9.0
«execution environment» MySQL Server 5.7
«artifact» my.cnf
«deployed»
«artifact» wspLogs
«artifact» WSP.war
«manifest»
«@WebService» EmployeeWSB
SecureLogger

Figure 18: Deployment for secure logger in the context of a secure service proxy. In this case, the WSP plays the role of client and MySQL Server plays the role of server. This is important in order to understand the keys deployed in the keystore used by MySQL Server (`mysqlKeystore`)

It is worth noting that intercepting validators can be used for other purposes than avoiding code injection attacks. They could perform other security-related tasks before invoking a service.

4.6 Avoiding Code Injection Attacks in the WSP: CSP Message Interceptor Gateway and CSP Message Inspector

According to the CSP catalogue [71]:

- *Message interceptor gateway* (MIG) is a proxy infrastructure providing a centralized entry point that encapsulates access to all target service endpoints of a web services provider.
- *Message inspector* (MI) is a modular or pluggable component that can be integrated with infrastructure service components that handle pre-processing and post-processing of incoming and outgoing SOAP or XML messages.

This section explains how to use them to prevent code injection attacks in the WSP.

WSPs are web applications and, like WSCs, they can be subject to code injection or DoS attacks. To prevent this type of attacks, CSP provides two patterns: MIG and MI. The MIG is any software or hardware component responsible for filtering the incoming requests to the WSP. Once filtered, each connection is passed to one or several MIs that work as intercepting filters in the WSP.

In this case, servlet filters have not been used because they did not give access to elements of the request sent to the WSP [69]. Thus, MIG has been implemented using Apache Tomcat, and MI has been implemented as *CXF interceptors* [9]. Thus, MIs extend the abstract class `org.apache.cxf.phase.AbstractPhaseInterceptor` and are referenced from the `cxf-servlet.xml` file belonging to the WSP [6]. This approach is valid for both JAX-WS and JAX-RS web services.

Figure 17 depicts the deployment for MIG and MI.

It is worth noting that, as in the case of intercepting validators, MIG and MI can be used for purposes other than avoiding code injection attacks. They could perform other security-related tasks before accessing a service.

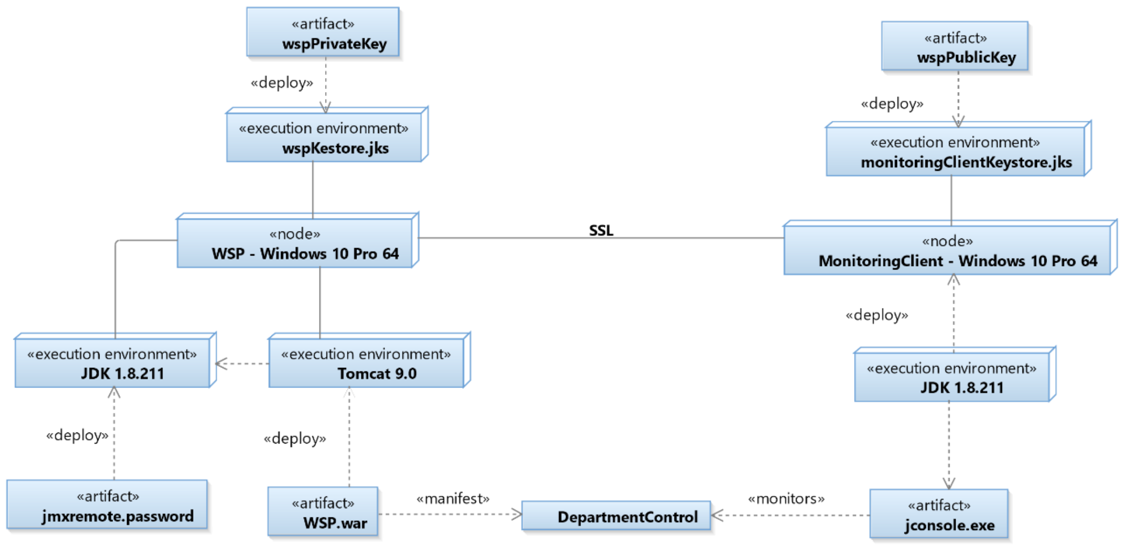
«artifact» wspPrivateKey
«deploy»
«execution environment» wspKestore.jks
«node» WSP - Windows 10 Pro 64
SSL
«artifact» wspPublicKey
«deploy»
«execution environment» monitoringClientKeystore.jks
«node» MonitoringClient - Windows 10 Pro 64
«deploy»
«execution environment» JDK 1.8.211
«execution environment» Tomcat 9.0
«deploy»
«deploy»
«execution environment» JDK 1.8.211
«artifact» jmxremote.password
«artifact» WSP.war
«manifest»
DepartmentControl
«monitors»
«artifact» jconsole.exe

Figure 19: Deployment for dynamic server management of the WSP. In the figure, the class `DepartmentControl` is the control object used for taking account of the creation and deletion of departments

4.7 Logging Every Tier: CSP Secure Logger, CSP Secure Base Action, CSP Policy Delegate, CSP Secure Service Proxy, CSP Audit Interceptor

According to the CSP catalogue [71]:

- *Secure logger* logs messages in a secure manner so that they cannot be easily altered or deleted and so that events cannot be lost.
- *Secure base action* coordinates security components and provides web tier components with a central access point for administering security-related functionality.
- *Policy delegate* mediates requests between clients and security services, reducing the dependency of client code on the implementation specifics of the service framework.
- *Secure service proxy* provides authentication and authorization externally by intercepting requests for security checks and then delegating the request to the appropriate service.
- *Audit interceptor* centralizes auditing functionality and defines audit events declaratively, independently of the business tier services.

This section explains how to use them for logging every tier in SOA applications.

Knowing who has done what in an enterprise application is considered important nowadays. The Enron case showed how important financial information can be erased, making it impossible to know both who did it and what the information was. To avoid these problems, for companies listed on the stock exchange, the *Sarbanes-Oxley Act* mandates that every user action in the system be logged for accounting applications [51].

To that extent, CSP catalogue provides a secure logger that uses a secure pipe to log the application's behavior against some register, such as a table in a relational database. The process for establishing SSL connections between MySQL Server and its clients is very similar to the one used for making WS-Security work and is defined in [20]. The MySQL configuration file `my.cnf` references public and private keys belonging to the client and server, which are deployed in a keystore.

It is important to note that, in a SOA application, a user request goes through several tiers and components. Therefore, the CSP catalogue defines several patterns for logging (or doing other security processing) in each tier:

- For the presentation tier of the WSC a secure base action is defined.
- For the business tier of the WSC a policy delegate is defined.
- For the business tier of the WSP a secure service proxy is defined for logging the proxy that gives

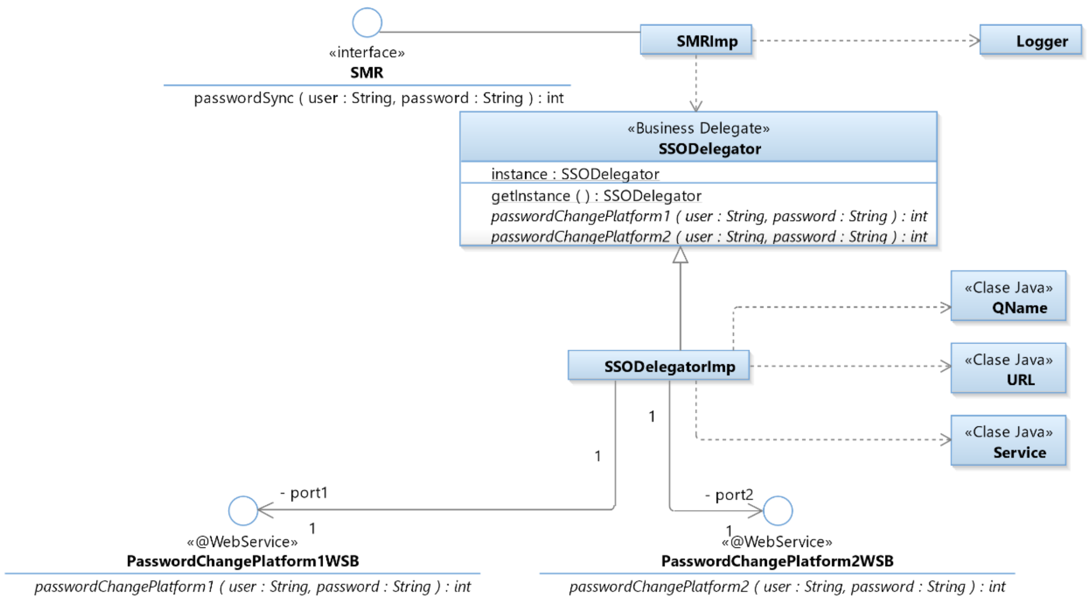
«interface»
SMR
passwordSync (user : String, password : String) : int
SMRImp
Logger
«Business Delegate»
SSODelegator
instance : SSODelegator
getInstance ( ) : SSODelegator
passwordChangePlatform1 (user : String, password : String) : int
passwordChangePlatform2 (user : String, password : String) : int
«Clase Java»
QName
SSODelegatorImp
«Clase Java»
URL
«Clase Java»
Service
1
1
- port1
1
- port2
1
«@WebService»
PasswordChangePlatform1WSB
passwordChangePlatform1 (user : String, password : String) : int
«@WebService»
PasswordChangePlatform2WSB
passwordChangePlatform2 (user : String, password : String) : int

Figure 20: The SMR (`SMR` and `SMRImp`) uses a SSO delegator (`SSODelegator` and `SSODelegatorImp`) for password synchronization. The SSO delegator has two JAX-WS interfaces (`PasswordChangePlatform1WSB` and `PasswordChangePlatform2WSB`) whose implementations give remote access to the WSPs. A `Logger` is used for manual transaction compensation

access to the application service, and an audit interceptor is defined for logging the application service. Audit interceptor is intended for those applications where no web services are present, because, otherwise, conducting loggings in the secure service proxy and the audit interceptor could be redundant.

Regarding implementation technologies, if JSF is used in the WSC's presentation tier, *managed beans* [28] are the closest to actions, and should thus be responsible for implementing secure base actions. Policy delegates are simple business delegates entrusted with additional security functions such as logging. Secure service proxies are web service brokers that implement security functions. Finally, audit interceptors are simple *proxies* [27] that create logs before accessing an application service.

Figure 18 depicts the use of a secure logger in the context of a secure service proxy (the `EmployeeWSB`) that makes logs using a secure pipe against the MySQL table `wspLogs`. The MySQL server is deployed in the same node but could be deployed in a different one. A simple logger class can be programmed, or more advanced loggers such those provided in *Java Logging API* [72] or *Apache Log4j 2* [10] can be used. Deployment of loggers for secure base actions, policy delegates and audit interceptors are equivalent to the one depicted in Figure 18.

It is worth noting that, similar to the intercepting validators, secure base action, policy delegate, secure service proxy, and audit interceptor can be used for different purposes other than logging, but taking into account the coordinated use of different patterns in the application developed, logging is the most reasonable use for them.

4.8 Controlling Objects Deployed in the Server: CSP Secure Pipe and CSP Dynamic Service Management

According to the CSP catalogue, *dynamic service management* enables fine-grained instrumentation of business objects at runtime on an as-needed basis using JMX [71].

This section explains how to use it for controlling objects deployed in the server.

J2EE provides *Java Management Extension* (JMX) [64] a mechanism for the remote control from the classes deployed in a JVM. This can be useful for accounting and reporting the classes deployed in the WSP. JMX is

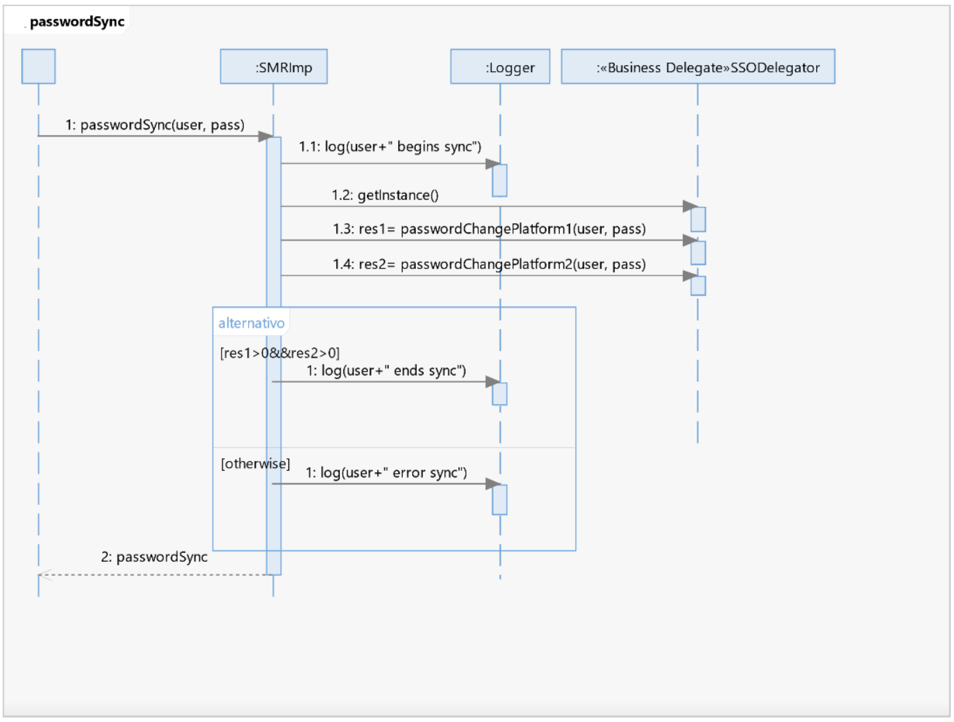
passwordSync
:SMRImp
:Logger
:«Business Delegate»SSODelegator
1: passwordSync(user, pass)
1.1: log(user+" begins sync")
1.2: getInstance()
1.3: res1= passwordChangePlatform1(user, pass)
1.4: res2= passwordChangePlatform2(user, pass)
alternativo
[res1>0&&res2>0]
1: log(user+" ends sync")
[otherwise]
1: log(user+" error sync")
2: passwordSync

Figure 21: Interaction between `SMRImp` and `SSODelegator` for password synchronization

characterized in CSP as the dynamic service management pattern.

The use of JMX is quite straightforward. Control objects that audit the desired behavior (e.g. the creation or deletion of a department) are created and then referenced with a specific name using `javax.management.MBeanServer` class. These control classes can then be remotely accessed using the `jconsole.exe` application included in the standard JDK.

Certainly, SSL connections and, at least, username/password access must be granted. This forces the presence of the WSP private key in the WSP key store and the WSP public key in the monitoring remote application keystore, as well as the presence of a file `jmxremote.password` in the WSP defining the roles and passwords [62].

Figure 19 depicts the deployment for dynamic service management.

4.9 Secure Invocation of WSP Using SSO + Password Synchronization: CSP Secure Pipe, CSP Secure Message Router, CSP SSO Delegator, CSP Assertion Builder, CSP Password Synchronizer

According to the CSP catalogue [71]:

- *Secure Message Router* (SMR) establishes a security intermediary infrastructure that aggregates access to multiple application endpoints in a workflow or among partners participating in a web-services transaction.
- *Single Sign-On delegator* (SSO delegator) encapsulates access to identity management and single sign-on functionalities, allowing the independent evolution of loosely coupled identity management services while providing system availability.

- *Assertion builder* abstracts similar processing control logic to create SAML assertion statements.
- *Password synchronizer* centralizes management of synchronizing user credentials across different application systems via programmatic interfaces.

This section explains how to use them to make secure invocations of WSPs using *Single Sign-On* (SSO). We have used the case of password synchronization as an example of WSP.

Secure invocation of WSPs using SSO is the most complex issue addressed by the CSP catalogue, because it involves three patterns. The two most important are SMR and SSO delegator. SMR is a class responsible for secure communications with different WSPs in a federated environment. SSO delegator is a class responsible for hiding the process of invoking heterogeneous WSPs (e.g. SOAP and REST) as well as the identity management process. Assertion builder is used for building the SAML assertions used during the single sign-on.

Thus, SMR is responsible for the choreography of several WSPs, which are invoked in a federated environment using an SSO delegator. This SSO delegator uses SAML assertions for credential management during WSP invocation.

The implementation of SMR made the creation of a new use case necessary in the application: the password synchronization in different applications. Thus, the SMR was used to implement the password synchronizer pattern. For the sake of simplicity, *Service Provisioning Markup Language* [55] was not used in our implementation. Of course, SMR does not involve the presence of a password synchronizer, but we thought that this was a suitable use case that could mix both the SMR and the password synchronizer.

The main problem with the SMR is that the information contained in the CSP catalogue about *Liberty-enabled identity providers* [71] is very misleading because identity providers are suitable for web applications, where humans interact with web pages, but are not well tailored for web services. The *translation* of all of this to the web services world is called *Security Token Service* (STS) [67][70], and in the context of Apache CXF, Talend's STS is the key element [49][73]. STSs validate username/password sent by WSCs (the SMR in our case) and include SAML tokens in WSC requests sent to the WSPs. WSPs then use these SAML tokens to accept or deny the request. WSPs can follow three different strategies for the validation of the SAML tokens, i.e., for the subject confirmation: *holder of key*, *bearer* and *sender vouches* [42][74].

Holder of key is used in our application. According to this strategy [37][49]: (i) WSC signs (or includes username/password) and sends a request for validating itself against the STS; (ii) STS validates the WSC sign using the WSC public key (or username/password), generates an SAML token with the WSC public key and signs it; (iii) the WSC gets the SAML token, generates a SOAP message, and signs it; (iv) WSP gets the SAML token, validates the STS signature of the SAML token using the STS public key, gets the WSC public key from the SAML token and validates the WSC signature of the SOAP message using the WSC public key included in the SAML token. Thus, WSP trusts the SOAP message, because it is signed by the WSC, and trusts the WSC because it is included in the SAML assertion signed by the STS. Finally, the WSP trusts the STS precisely because it is the component entrusted with user authentication. Although not explicitly mentioned, signatures are made using the private keywords of each signer [71].

This approach makes it unnecessary for heterogeneous WSPs (e.g. a WSP for hotel booking and a WSP for flight purchase) working in a federated environment (e.g. a WSC for travel planning) to handle the validation of the WSCs. WSPs trust the STS, and, therefore, the STS is responsible for deploying the WSC public keys in its keystore (or other validation approaches) for the validation of WSCs.

Note that STS is about user credentials and is orthogonal to the use of secure pipes or WS-Security for avoiding MITM attacks.

In our application, the implemented SMR invokes two WSPs, which does password synchronizations in two different platforms. Both WSPs are SOAP services that expect a SAML ticket with the WSC credentials signed by the STS. HTTPS connections are established between the WSC and the STS and WS-Security is used between the WSC and the WSPs.

Figure 20 depicts the classes involved in the SMR. Figure 21 depicts how the `SMRImp` does the password synchronization between platform 1 and platform 2 using an `SSODelegator`. Note that no reference to STS or SAML is made because the entire configuration is handled using Apache CXF configuration files. No classes are necessary for assertion builder, because frameworks build and manage the SAML tokens without user intervention. [11] shows examples of how the WSS4J classes manage the SAML assertions, making the explicit implementation of an assertion builder unnecessary.

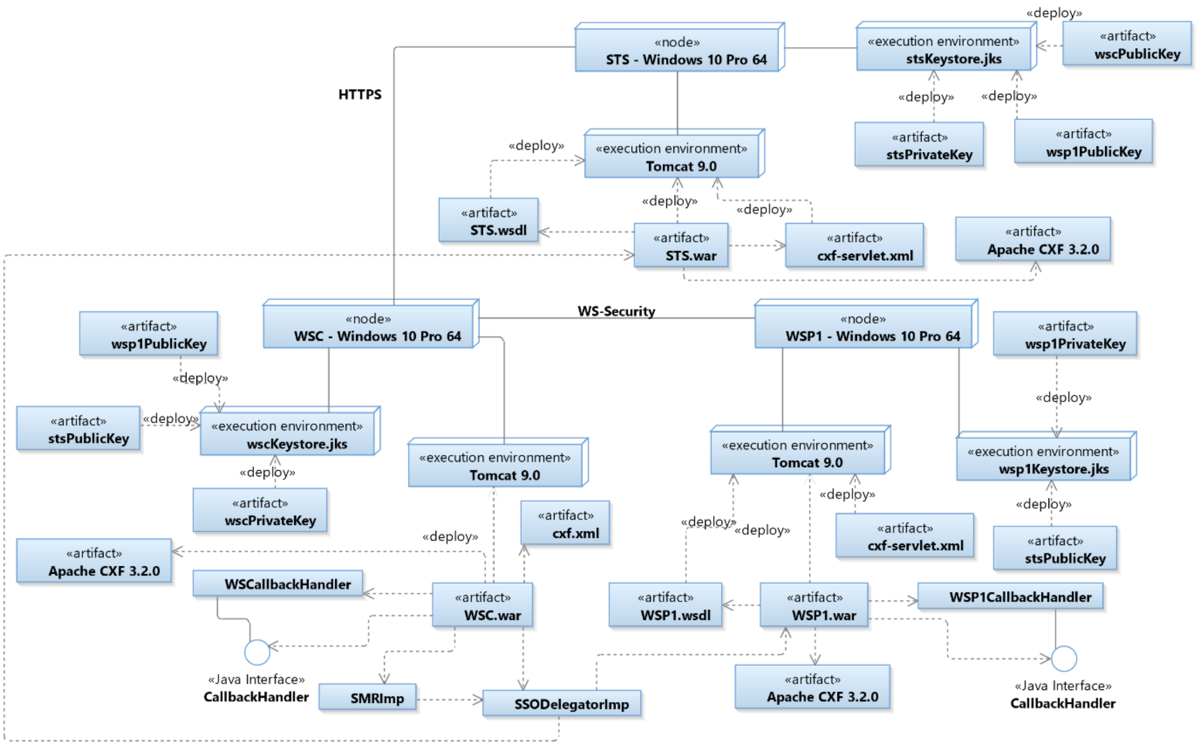
«node» STS - Windows 10 Pro 64
«execution environment» stsKeystore.jks
«artifact» wscPublicKey
«artifact» stsPrivateKey
«artifact» wsp1PublicKey
HTTPS
«execution environment» Tomcat 9.0
«artifact» STS.wsdl
«artifact» STS.war
«artifact» cxf-servlet.xml
«artifact» Apache CXF 3.2.0
«deploy»
WS-Security
«node» WSC - Windows 10 Pro 64
«node» WSP1 - Windows 10 Pro 64
«artifact» wsp1PublicKey
«artifact» stsPublicKey
«execution environment» wscKeystore.jks
«artifact» wscPrivateKey
«execution environment» Tomcat 9.0
«artifact» cxf.xml
«artifact» Apache CXF 3.2.0
WSCallbackHandler
«artifact» WSC.war
«Java Interface» CallbackHandler
SMRImp
SSODelegatorImp
«artifact» wsp1PrivateKey
«execution environment» Tomcat 9.0
«execution environment» wsp1Keystore.jks
«artifact» cxf-servlet.xml
«artifact» stsPublicKey
«artifact» WSP1.wsdl
«artifact» WSP1.war
WSP1CallbackHandler
«artifact» Apache CXF 3.2.0
«Java Interface» CallbackHandler

Figure 22: Deployment for SMR. For the sake of conciseness, only one platform for password change is depicted

It is worth noting that this SMR operation would need a distributed transaction [43]. However, for the sake of simplicity, we have omitted it, making a log of the process to make transaction compensation easier, if necessary. Thus, as Figure 21 depicts, if any process fails after a WSP invocation via the SSO delegator, at least a log describing an open transaction that was not closed is recorded in the logger.

To make our SMR work, the WSC must include a directive in the `cxf.xml` to obtain the SAML ticket from the STS and provide an implementation of `javax.security.auth.callback.CallbackHandler` interface that is used to obtain the WSC private key password. The WSC keystore must contain its private key, the STS public key, and the WSP public key [49].

The STS must include a WSDL file defining the authentication method for the WSC (note that the STS is itself a WSP) to include modifications in the STS `cxf-servlet.xml` file, and of course, to deploy the Talend's STS. The STS keystore must contain its private key and WSP public key [49]. In addition, if the WSC signs the SOAP messages sent to the STS (as our WSC does) instead of including username/password, the STS keystore must also contain the WSC public key.

The WSP must include in its WSDL file the policy annotations that refer to the STS and define input/output policies. It must also provide an implementation of the `javax.security.auth.callback.CallbackHandler` to return the WSP private key password when needed by the WSP. The `cxf-servlet.xml` file must also be modified. the WSP keystore must contain its private key and STS public key [49].

Figure 22 depicts the deployment diagram for the SMR. For the sake of conciseness, only one platform for password change is depicted, but the second one would be equivalent to the one depicted.

Although STS has been bound to the SMR in the previous description, in practice, the SMR delegates the SSO and WSP invocation in the SSO delegator. Therefore, the SSO delegator is the class that (in turn, delegating in Apache CXF) generates the SOAP requests to both the STS and the WSPs.

4.10 The Rest of the CSP Patterns

Three patterns from the CSP catalogue have not been implemented in our project, because they were not necessary, taking into account our security and SOA multitier requisites [71]:

- *Intercepting web agent*, used for retrofitting authentication and authorization in an existing web application. Our application needs no retrofit authentication/authorization; therefore, the pattern was not implemented.
- *Obfuscated transfer*, used for protecting critical data within applications and between tiers. The presence of HTTPS and/or WS-Security made its use unnecessary.
- *Secure service façade*, a session façade that deals with security concerns. Session façades are only used in the context of EJBs [1], but our application exposed the logic using web services instead of EJBs.

5. Evaluation and Analysis

The paper's main goal was to analyze the patterns included in the CSP from a practical point of view, driven by the security problems that must be addressed when securing SOA enterprise applications. This analysis also sought to take into account the J2EE technologies needed for implementing the CSP patterns as well as the multitier patterns more closely related to CSP patterns. Table 1 summarizes this analysis carried out in Section 4.

Regarding the integration of CSP and CJP patterns, multitier architecture is a modular solution for the development of enterprise applications. Multitier architecture is mainly service-oriented because the application service pattern gathers the services provided by the application. Because the multitier architecture expects these services to be accessed by remote clients, it provides three patterns to expose application services to them [1]: session façade for EJB clients, web service broker for SOA clients (REST or SOAP), and service activator for message-driven (or event-driven) clients. The invocation of these remote-exposed services can be accomplished with the business delegate pattern. As Table 1 suggests, the business delegate and web service broker are the key elements for including security activities in SOA applications. Of course, other CJP patterns are needed to secure enterprise applications, but business delegate and web service broker are the most commonly used

As we have previously mentioned, the configuration of security frameworks is one of the most complex issues during the implementation of CSP security patterns. Table 1 identifies the security frameworks and technologies used for securing our SOA application, and Section 4 provides UML-deploying diagrams that help to understand the configuration of these frameworks.

Finally, regarding the suitability of CSP patterns to handle security threats, several tests and validations were made to the application developed. However, because security elements are implemented in terms of security frameworks and technologies, the security frameworks themselves are primarily responsible for dealing with security threats. For example, are secure pipes (HTTPS connections) properly established in our SOA application? We can guarantee that we have configured Apache Tomcat to use only HTTPS connections using specific X.509 certificates. Are these HTTPS connections safe? Yes, as long as Apache Tomcat is able to manage HTTPS connections. Have we filtered HTTP connections to verify that Apache Tomcat's HTTPS connections are encrypted? No, because we trust Apache Tomcat's HTTPS connections. It is possible to incorrectly configure security frameworks, but these frameworks are extremely cautious. Programmers must pay careful attention during configuration; otherwise, the frameworks can generate a myriad of exceptions that will saturate the application. Special attention has thus been paid to their configuration. In addition, we have conducted several security-related tests in our application. JUnit and manual tests were carried out to validate the application developed. These tests checked the correctness of the application and the security elements included in it. The configuration was the one used by the implemented application. In detail:

- *Authentication/*authorization *in WSC*s. CSP authentication and authentication enforcers were implemented using JAAS and Apache Tomcat. We run tests to make sure that it was impossible to access web pages or servlet classes without JAAS-defined credentials. Therefore, authentication and authorization were enforced in our application using properly configured JAAS, as the testing confirmed.

- *HTTPS connections*. CSP secure pipe was implemented using the Apache Tomcat option for using only HTTPS connections for SOAP and REST WSBs, as well as all of the configuration steps described in Section 4.2. We can be sure that these connections were established because Tomcat refused connections without the server's public key, used to encrypt the information, and the server's private key, used to decrypt the information. These keys are not needed for regular HTTP connections. Therefore, HTTPS connections were established in our application using properly configured Apache Tomcat and X.509 certificates, as we confirmed with testing.

Table 1: Security concerns addressed, CSP security patterns, implementing frameworks/technologies and related CJP multitier patterns

Security concern	**CSP security patterns**	**Frameworks/Technologies**	**CJP multitier patterns**
Authentication/authorization in WSC application	Authentication and authorization enforcer	JAAS, Tomcat	-
HTTPS connections	Secure pipe	X.509 certificates, Apache Tomcat	-
Point-to-point secure service invocation of WSP	Secure pipe, secure session object, container-managed security, credential tokenizer	X.509 certificates, Apache Tomcat, Apache CXF, JAX-WS, JAX-RS	Business delegate, web service broker
End-to-end secure service invocation of WSP	-	WS-Security, X.509 certificates, Apache Tomcat, Apache CXF, JAX-WS	Business delegate, web service broker
Code injection in WSC	Intercepting validator	Apache Tomcat	Intercepting filter
Code injection in WSP	Message interceptor gateway, message inspector	Apache Tomcat, Apache CXF	-
Logging every tier:			
Presentation - WSC	Secure base action, secure logger	JSF, MySQL, X.509 certificates	Front/application controller
Business - WSC	Policy delegate, secure logger	Apache CXF, MySQL, X.509 certificates	Business delegate
Business - WSP - web service	Secure service proxy, secure logger	Apache CXF, MySQL, X.509 certificates	Web service broker
Business - WSP - application service	Audit interceptor, secure logger	MySQL, X.509 certificates	Application service
Controlling objects deployed in the server	Secure pipe, dynamic service management	X.509 certificates, Apache Tomcat, JMX	-
Secure invocation of WSP using SSO	Secure pipe, secure message router, SSO delegator, assertion builder	WS-Security, X.509 certificates, Apache Tomcat, Apache CXF, Talend's STS, JAX-WS, JAX-RS	Business delegate, web service broker
Password synchronizing in several platforms	Password synchronizer, secure pipe	X.509 certificates, Apache Tomcat, Apache CXF, JAX-WS, JAX-RS	Business delegate, web service broker, distributed transaction management
Retrofitting authentication and authorization in an existing web application	Intercepting web agent	-	Front controller
Protecting critical data within application and between tiers	Obfuscated transfer	-	Transfer object
Security concerns in session façades	Secure service façade	Session EJB	Session façade

- *Point-to-point secure invocation of WSPs*. CSP secure pipe, CSP secure session object, CSP container-managed security and CSP credential tokenizer have to be implemented to address this security threat. Secure session object is an inner class used to manage security information and has no direct influence on the implementation of the point-to-point secure invocation; it only helps to manage information in the context of the application. Therefore, the correct application relies on the implementation of CSP secure pipes, CSP container-managed security, and CSP credential tokenizers. CSP container-managed security was achieved using Apache Tomcat capabilities. It was not possible to invoke WSP services without the WSC credentials that Tomcat requires to allow WSP services. The CSP credential tokenizer was implemented using the Apache CXF framework, which included WSC credentials in a SOAP/REST service invocation. Information was encrypted using CSP secure pipes, which were tested, as we previously mentioned. Therefore, point-to-point secure invocation of WSPs was achieved in our application using properly configured Apache Tomcat, X.509 certificates and Apache CXF, as we confirmed with testing.

- *End-to-end secure service invocation of WSP*. No CSP pattern is defined for this issue, so we used WS-Security to implement end-to-end secure invocation. The implementation of WS-Security is a complex issue, as Section 4.4 and Figure 15 describe. Any failure in the configuration described in Figure 15 generated a deluge of exceptions in the application. The implementation of WS-Security relies heavily on Apache WSS4J interceptors. In this case, it was possible to see how outgoing and incoming SOAP messages were encrypted by these interceptors. We did not check the encryption made by the WSS4J interceptors because we trust Apache's implementations. Therefore, end-to-end secure invocation of WSPs was achieved in our application using properly configured Apache Tomcat, X.509 certificates and Apache WSS4J interceptors, as we confirmed with testing.

- *Code injection in WSCs*. The CSP intercepting validator was implemented using servlet filters. We checked that incoming requests to the WSC application were intercepted and processed by the servlet filters. Therefore, the detection of code injection in the WSC was achieved in our application using properly configured Apache Tomcat, as we confirmed with testing.

- *Code injection in WSPs*. Apache Tomcat played the role of CSP MIG, and CSP MIs were implemented using Apache CXF interceptors. The validations made are similar to those made to avoid code injection attacks in the WSC. Therefore, the detection of code injection in the WSP was achieved in our application using properly configured Apache Tomcat and Apache CXF, as we confirmed with testing.

- *Logging every tier*. CSP secure logger, CSP secure base action, CSP policy delegate, CSP secure service proxy, and CSP audit interceptor had to be implemented to make logs in every audited component of the SOA application. The CSP secure logger was implemented using MySQL accessed via HTTPS connections. Thus, the validations made were similar to those made with CSP secure pipes. With regard to the logging of the audited components, they were easily validated as logs were created as the requests passed through the different components. Therefore, logging on every tier was achieved in our application using properly configured MySQL, X.509 certificates, JSF, and Apache CXF, as we confirmed with testing.

- *Controlling objects deployed in the server*. CSP dynamic service management was implemented using JMX. Validations checked that monitored elements were properly controlled using JMX. CSP secure pipe (SSL connection) was tested as in the rest of the cases. Therefore, the control of objects deployed in the server was achieved in our application using properly configured Apache Tomcat, X.509 certificates, and JMX, as we confirmed with testing.

- *Secure invocation of WSP using SSO*. The CSP secure message router is a WSC coordinator responsible for the invocation of different WSPs, which delegates the SSO capabilities in the CSP SSO delegator. The implementation of the CSP SSO delegator was made using Apache CXF and Talend's STS. The configuration of Talend's STS is a very complex process, as Figure 22 shows. We checked that SAML assertions were requested by the WSC, included by the STS in the SOAP messages, and validated in the WSP. The CSP assertion builder was not built by us, because SAML assertions are automatically included by the STS and managed by Apache CXF, both in the WSC and the WSP. The CSP secure pipe was tested as in the rest of the cases. Therefore, the secure invocation of WSP using SSO was achieved in our application using properly configured Apache Tomcat, X.509 certificates, Apache WSS4 interceptors, and Talend's STS, as we confirmed with testing.

- *Password synchronizing in several platforms*. The CSP password synchronizer is a simple class that can be easily tested by checking whether or not passwords are changed in the respective databases. The complexity here is to implement it as a CSP secure message router with a CSP single sign-on delegator, which we did and tested in our project. The CSP secure pipe was tested as in the rest of the cases. Therefore, password synchronization was achieved in our application using properly configured Apache Tomcat and Apache CXF, as we confirmed with testing. This password synchronization was also secure and SSO, as we tested (see *Secure invocation of WSP using SSO* above).

- Finally, CSP intercepting web agent, obfuscated transfer, and secure service façade were not implemented, because they were not useful in our application. However, the underlying CJP patterns needed for their implementation (front controller, transfer object, and session façade) do not involve any specific security framework and can be easily implemented and tested.

6. CONCLUSIONS AND FUTURE WORK

The first conclusion of this work is that including security in a J2EE SOA application is a complex issue and is beyond what the average programmer is able to take on. Not only must JAX-WS and JAX-RS be mastered by programmers, but also all the complex J2EE security frameworks and standards. Thus, to make J2EE SOA application architectures secure it is necessary to master a large number of frameworks that are only accessible to elite J2EE architects. This can explain why there are only two enterprise application development platforms (i.e. J2EE and Microsoft .NET). Their implementation requires an enormous effort by the respective software vendors (i.e. Oracle/Eclipse Foundation and Microsoft) and their use demands a significant effort by designers and programmers.

In our opinion, the analysis carried out in Section 4, and Section 5 makes it easier for architects and developers to secure J2EE SOA multitier applications. Although specific technologies are essential to tackle the security concerns, the CSP patterns provide an abstraction layer that, to some extent, isolates security concerns from specific technologies and makes this paper more abstract than J2EE security books. In addition, the relationships established between CSP patterns and multitier patterns make it easier for multitier architects and developers with little experience with CSP patterns to include them in SOA multitier applications. The presence of UML diagrams included in this paper also helps in this respect.

A significant conclusion of this work is that the CSP catalogue does not consider a pattern for end-to-end secure invocation of WSPs. Although in practice, this issue is closely tied up with WS-Security, an abstract security pattern could be defined.

In addition, this study demonstrates that, if multitier architecture patterns are properly applied, security is an orthogonal aspect for SOA multitier development because the presence of business delegates in WSCs and web service brokers in WSPs make it very easy to include security issues in pre-existing code that must address security concerns. This is an advantage of multitier patterns themselves. Of course, the lack of business delegates and web service brokers would make it difficult to include security patterns in SOA applications and to maintain them.

Furthermore, the presence of security frameworks, configurable with external files, makes it easier to include security features in existing code. However, based on our experience, configuring these frameworks is a complex task, and can therefore make maintenance difficult if future security-related changes are needed. Moreover, because the security implementation relies on these frameworks, the practical implementation of CSP patterns becomes an issue of framework configuration, which can be very complex and, in contrast to CSP abstract patterns, can involve a myriad of low-level details.

This paper also demonstrates that no J2EE application servers are needed to deploy secure J2EE SOA enterprise applications. However, our implementation is very dependent on inner frameworks and containers, in particular, on the frameworks used for web services publication (i.e. Apache CXF) and on the web container (i.e. Apache Tomcat). From this point of view, it would be very difficult to change Apache CXF for another implementation of JAX-WS and JAX-RS, due to both SOA and security issues. Of course, because Apache CXF is highly configured by external files, this change would have low impact on the code, but a high impact on application deployment. The change from Apache Tomcat to another web container would have similar difficulties. Therefore, if being bound to a specific J2EE application server can be a concern for secure SOA multitier applications, similar concerns appear for the underlying technologies for deploying them without application servers. Consequently, companies must carefully choose the software providers on which they will build their SOA and security infrastructure, since security development is very dependent on and bound to them. However, the companies must assume this risk, since the implementation of these services from scratch would be extremely expensive and unfeasible.

If we analyze the categories provided by the CSP catalogue, we can see that logging is a common issue present in web and business tiers. The web tier patterns are mainly focused on authentication/authorization issues and on checking the data arriving at WSCs. The business tier patterns focus on secure invocation of WSPs, and message validation. The web service tier focusses on analyzing messages arriving at WSPs and on augmenting the authorization/authentication of WSPs beyond user and password via HTTPS. In particular, the SMR pattern has been very complex, involving an SSO delegator. Identity management patterns are focused on credential management and in single sign-on (this issue is also closely related to SMR). Finally, the service provisioning category only has one pattern focused on password synchronization between different platforms.

Regarding the tolerance of our application to security attacks, its strength is directly proportional to the effectiveness of the CSP patterns and the correctness of the security frameworks used. Thus, if an attack is not considered in the pattern catalogue, our application would be completely vulnerable (excluding those attacks solved by WS-Security). In the same way, if Apache's security frameworks are not properly implemented, the application's security can be compromised. However, the CSP catalogue provides extensive coverage of all types of attacks to SOA multitier applications, the industry trusts Apache frameworks, and our applications have been properly tested.

Future work should include extending the CSP catalogue to deal with the issues not yet covered, such as WS-Security (as this paper considers) or multi-factor authentication (not considered here).

Testing the performance of WS-Security vs. simple HTTPS connections would be very interesting.

Migrating Apache CXF and/or Apache Tomcat to other frameworks and/or containers, or even the whole application to one or several J2EE application servers, would be a very interesting exercise, but it would be extremely costly. In any case, the CSP patterns would still be valid, although the underlying deployment-support files and classes would change significantly.

Finally, comparing the effort for developing .NET secure SOA applications, as well as the applicability of CSP patterns to .NET, would be a very interesting area of research, but it would have an even greater cost than migrating Apache CXF and/or Apache Tomcat to another J2EE technology.

REFERENCES

[1] D. Alur, D. Malks, and J. Crupi. *Core J2EE Patterns: Best Practices and Design Strategies. Second Edition*. Upper Saddle River, New Jersey: Prentice Hall, 2003.

[2] A.K. Alvi, M. Zulkernine, "A Comparative Study of Software Security Pattern Classifications", in Proc. *ARES 2012*, pp. 582-589, 2012.

[3] P. Anand, J. Ryoo and R. Kazman, "Vulnerability-Based Security Pattern Categorization in Search of Missing Patterns", in Proc. *ARES 2014*, pp. 476-483, 2014.

[4] Apache, "Apache CXF. An Open-Source Services Framework", https://cxf.apache.org/index.html, accessed 17th July 2020.

[5] Apache, "Apache Tomcat 9. Realm Configuration How-To", https://tomcat.apache.org/tomcat-9.0-doc/realm-howto.html#JDBCRealm, accessed 17th July 2020.

[6] Apache, "Apache CXF configuration", http://cxf.apache.org/docs/configuration.html, accessed 17th July 2020.

[7] Apache, "Apache CXF. Configuring an endpoint", http://cxf.apache.org/docs/jax-ws-configuration.html, accessed 17th July 2020.

[8] Apache, "Using Apache WSS4J", http://ws.apache.org/wss4j/using.html, accessed 17th July 2020.

[9] Apache, "Apache CXF Interceptors and phases", https://cxf.apache.org/docs/interceptors.html, accessed 17th July 2020.

[10] Apache, "Apache Log4j 2", https://logging.apache.org/log4j/2.x/, accessed 17th July 2020.

[11] Apache, "Apache code examples of WSS4J classes for SAML tokens management", http://svn.apache.org/viewvc/webservices/wss4j/trunk/ws-security-dom/src/test/java/org/apache/wss4j/dom/saml/SamlTokenTest.java?view=markup, accessed 17th July 2020.

[12] A. Bien, "Client-Side HTTP Basic Access Authentication with JAX-RS 2.0", http://www.adam-bien.com/roller/abien/entry/client_side_http_basic_access, accessed 17th July 2020.

[13] D. Bisson, "What is an SSL/TLS X.509 Certificate?", https://www.venafi.com/blog/what-ssltls-x509-certificate, accessed 17th July 2020.

[14] B. Buege, R. Layman and A. Taylor, *Hacking Exposed J2EE & Java*. Berkeley, California: McGraw-Hill/Osborne, 2002.

[15] B. Burke. *RESTful Java with JAX-RS 2.0: Designing and Developing Distributed Web Services. Second Edition*. Sebastopol, California: O'Reilly, 2013.

[16] F. Buschmann, R. Meunier, H. Rohnert and P. Sommerlad, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Hoboken, New Jersey: Wiley, 1996.

[17] F. Buschmann, K. Henney and D.C. Schmidt, *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Hoboken, New Jersey: Wiley, 2007.

[18] F. Buschmann, K. Henney and D.C. Schmidt, *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. Hoboken, New Jersey: Wiley 2007.

[19] H. Cervantes, R. Kazman, J. Ryoo and D. Choi and D. Jang, "Architectural Approaches to Security: Four Case Studies", *IEEE Computer*, vol. 49, no. 11, 2016.

[20] cPanel, "How to Configure MySQL SSL Connections", https://documentation.cpanel.net/display/CKB/How+to+Configure+MySQL+SSL+Connections, accessed 17th July 2020.

[21] W. Crawford and J. Kaplan, *J2EE Design Patterns*. Sebastopol, California: O'Reilly, 2003.

[22] B. Doughan, "MOXy's @XmlInverseReference is now Truly Bidirectional", http://blog.bdoughan.com/2013/03/moxys-xmlinversereference-is-now-truly.html, accessed 17th July 2020.

[23] T. Erl, *SOA Design Patterns*. Upper Saddle River, New Jersey: Prentice Hall, 2009.

[24] E. Fernandez-Buglioni, *Security Patterns in Practice: Designing Secure Architectures Using*

Software Patterns. Hoboken, New Jersey: Wiley, 2013.

[25] M. Fisher, J. Ellis and J. Bruce, *JDBC API Tutorial and Reference. Third Edition*. Reading, Massachusetts: Addison-Wesley Professional, 2003.

[26] M. Fowler, *Patterns of Enterprise Application Architecture*. Reading, Massachusetts: Addison-Wesley Professional, 2002.

[27] E. Gamma, R. Helm, R., Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley Professional, 1994.

[28] D. Geary and C.S. Horstmann, *Core JavaServer Faces. Third Edition*. Upper Saddle River, New Jersey: Prentice Hall, 2010.

[29] M. Hafiz, P. Adamczyk, and R. Johnson, "Growing a Pattern Language (for Security)", in Proc. *Onward! 2012*, pp 139-158, 2012.

[30] S.T. Halkidis, A. Chatzigeorgiou and G. Stephanides, "A qualitative analysis of software security patterns", *Computers and Security*, vol. 25, 2006.

[31] M. Hall, L. Brown and Y. Chaikin, *Core Servlets and JavaServer Pages: Advanced Technologies. Vol. 2. Second Edition*. Upper Saddle River, New Jersey: Prentice Hall, 2007.

[32] M.D. Hansen, *SOA Using Java Web Services*. Upper Saddle River, New Jersey: Prentice Hall, 2007.

[33] F. Huertas and A. Navarro, "SOA support to virtual campus advanced architectures: The VCAA canonical interfaces", *Computer Standards & Interfaces*, vol. 40, 2015.

[34] Infoworld, "How to configure Tomcat to always require HTTPS", https://www.infoworld.com/article/3304289/how-to-configure-tomcat-to-always-require-https.html, accessed 17th July 2020.

[35] ITU-T, "X.509 (10/2016)", https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.509-201610-I!!PDF-E&type=items, accessed 17th July 2020.

[36] Java 67, "Difference between trustStore vs keyStore in Java SSL", https://www.java67.com/2012/12/difference-between-truststore-vs.html, accessed 17th July 2020.

[37] JBoss, "SAML Holder-Of-Key Assertion Scenario", https://docs.jboss.org/author/display/JBWS/SAML+Holder-Of-Key+Assertion+Scenario, accessed 17th July 2020.

[38] M. Kalin, *Java Web Services: Up and Running*. Sebastopol, California: O'Reilly, 2009.

[39] M. Keith and M. Schincariol, *Pro JPA 2*. New York City, New York: Apress, 2013.

[40] M. Kircher and P. Jain, *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. Hoboken, New Jersey: Wiley, 2004.

[41] P.J. Kumar, *J2EE Security for Servlets, EJBs, and Web Services*. Upper Saddle River, New Jersey: Prentice-Hall, 2004.

[42] D. Liyanage, "SAML Subject confirmation methods: Bearer vs. Holder of Key vs. Sender Vouches", http://dulanja.blogspot.com/2013/01/saml-subject-confirmation-methods.html, accessed 17th July 2020.

[43] M. Little, J. Maron, and G. Pavlik, *Java Transaction Processing: Design and Implementation*. Upper Saddle River, New Jersey: Prentice Hall, 2004.

[44] D, "Mandal, D. WS Security Implementation Using Apache CXF and WSS4J", https://sites.google.com/site/ddmwsst/ws-security-impl, accessed 17th July 2020.

[45] G. Marques, "JAAS authentication in Tomcat example", https://www.byteslounge.com/tutorials/jaas-authentication-in-tomcat-example, accessed 17th July 2020.

[46] G. Marques, "JAAS form based authentication in Tomcat example", https://www.byteslounge.com/tutorials/jaas-form-based-authentication-in-tomcat-example, accessed 17th July 2020.

[47] G. Marques, "JAAS logout example", https://www.byteslounge.com/tutorials/jaas-logout-example, accessed 17th July 2020.

[48] S. Martín, and E. Romero, "Patrones de seguridad software en el contexto de la arquitectura multicapa para la plataforma J2EE", https://eprints.ucm.es/56616/, accessed 17th July 2020.

[49] G. Mazza, "Deploying and Using a CXF Security Token Service (STS)", https://glenmazza.net/blog/entry/cxf-sts-tutorial, accessed 17th July 2020.

[50] R. de Miguel, "Atravesando las capas de una aplicación empresarial: demostrador tecnológico J2EE", https://eprints.ucm.es/50215/, accessed 17th July 2020.

[51] E.F. Monk and B. Wagner, *Concepts in Enterprise Resource Planning. Fourth edition*. Boston, Massachusetts: Course Technology, 2012.

[52] M. Mythliy, M.L. Valarmathi, C. Anand Deva Durai and J.A.M. Rexie, "An automation framework design for secure development", *Journal of Software: Evolution and Process*, vol. 31, no. 10, 2019.

[53] R. Nagappan and R. Williams, "Biometric Authentication for J2EE Applications", in Proc. *JavaOne Conference, 2005*.

[54] A. Navarro, J. Cristobal, C. Fernández-Chamizo and A. Fernández-Valmayor, "Architecture of a multiplatform virtual campus", *Software: Practice and Experience*, vol. 42, 2012.

[55] OASIS, "Service Provisioning Markup 2 Language (SPML) Version 1.0", https://www.oasis-open.org/committees/download.php/4137/os-pstc-spml-core-1.0.pdf, accessed 17th July 2020.

[56] OASIS, "Web Services Security: SOAP Message Security 1.1 4 (WS-Security 2004)", https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf, accessed 17th July 2020.

[57] OASIS, "Security Assertion Markup Language (SAML) V2.0 Technical Overview", http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html, accessed 17th July 2020.

[58] OASIS, "eXtensible Access Control Markup Language (XACML) Version 3.0", http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html, accessed 17th July 2020.

[59] Oracle, "Understanding Web Service Security Concepts", https://docs.oracle.com/middleware/1212/owsm/OWSMC/owsm-security-concepts.htm#OWSMC116, accessed 17th July 2020.

[60] Oracle, "keytool command", https://docs.oracle.com/javase/10/tools/keytool.htm#JSWOR-GUID-5990A2E4-78E3-47B7-AE75-6D1826259549, accessed 17th July 2020.

[61] Oracle, "JAAS Tutorials", https://docs.oracle.com/javase/10/security/jaas-tutorials.htm#JSSEC-GUID-272DB20A-B590-4B2E-BD60-7EF9EB54AB5A, accessed 17th July 2020.

[62] Oracle, "Enabling remote JMX with password authentication and SSL", https://docs.oracle.com/javadb/10.10.1.2/adminguide/radminjmxenablepwdssl.html, accessed 17th July 2020.

[63] W. E. Parra, "Integración de patrones de seguridad y patrones de diseño J2EE", https://eprints.ucm.es/26475/, accessed 17th July 2020.

[64] S.Perry, *Java Management Extensions*. Sebastopol, California: O'Reilly, 2002.

[65] M. Pistoia, N. Nagaratnam, L. Koved and A. Nadalin, *A. Enterprise Java Security. Building Secure J2EE Applications*. Reading, Massachusetts: Addison-Wesley, 2004.

[66] J. Ryoo, R. Kazman and P. Anand, "Architectural Analysis for Security", *IEEE Security and Privacy* vol. 13, no. 6, 2015.

[67] D. Senarath, "WS-Trust and Security Token Service (STS)", https://medium.com/@dinika.15/ws-trust-and-security-token-service-sts-fd1c92a5f53c, accessed 17th July 2020.

[68] D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Hoboken, New Jersey: Wiley, 2000.

[69] stackoverflow, "HTTP GET with request body", https://stackoverflow.com/questions/978061/http-get-with-request-body, accessed 17th July 2020.

[70] Stackoverflow, "Do we absolutely need a STS for SAML?", https://stackoverflow.com/questions/485084/do-we-absolutely-need-a-sts-for-saml, accessed 17th July 2020.

[71] C. Steel, R. Nagappan and R. Lai, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Upper Saddle River: Prentice Hall, 2005.

[72] L. Vogel, "Java Logging API – Tutorial", https://www.vogella.com/tutorials/Logging/article.html, accessed 17th July 2020.

[73] O. Wulff, "Configure and deploy CXF 2.5 STS - Part I Open Source and SOA, ESB and Security", http://owulff.blogspot.com/2011/10/configure-and-deploy-cxf-25-sts-part-i.html, accessed 17th July 2020.

[74] O. Wulff, "SAML tokens and WS-Trust Security Token Service (STS)", http://owulff.blogspot.com/2012/02/saml-tokens-and-ws-trust-security-token.html, accessed 17th July 2020.

[75] N. Yoshioka, Washizaki and K.A. Maruyama, "A survey on security patterns", *Progress in Informatics*, vol. 5, 2008.

AUTHOR BIOGRAPHIES

Antonio Navarro obtained his PhD in Mathematics-Computer Science from Universidad Complutense de Madrid, Spain, in 2002. He is associate professor in the Software Engineering and Artificial Intelligence Department of Universidad Complutense de Madrid. His research interests include software engineering, software architectures, software design patterns, software modelling, and model-driven architecture. He is the author and coauthor of several papers related to these research topics.

Wilmer Eduardo Parra obtained his university degree in Computer Science from Universidad Industrial de Santander, Colombia, in 2007, were he worked with the Calumet group. He obtained his MSc in Computer Science from Universidad Complutense de Madrid, Spain, in 2014, and his MSc in Big Data and Data Science from Universidad Autónoma de Madrid, Spain, in 2017. He has worked in different projects in public and private companies such as Renfe, ADIF and El Corte Inglés. Currently he is working at the Spanish bank BBVA. His research interests include software engineering, big data, data science, and IT and digital transformation.

Eduardo Romero obtained his Certificate of Higher Education in Network System Administrator from IES Valdehierro, Spain, and his university degree in Software Engineering from Universidad Complutense de Madrid, Spain, in 2019. Currently he is working at GMV as service developer for the Galileo Project from European Space Agency. His research interests include UI/UX, web services, system administration, and web applications security.

Sergio Martin obtained his university degree in Electronic Engineering and Industrial Automation from Universidad de Alcalá, Spain, and his university degree in Software Engineering from Universidad Complutense de Madrid, Spain, in 2019. His research interests include software modelling, artificial intelligence for stock trading, big data and web applications.

Rodrigo de Miguel obtained his university degree in Software Engineering from Universidad Complutense de Madrid, Spain, in 2018. We is co-founder of the web pet shelter *Adopta un Animal* https://www.adopta-un-animal.es/. His research interests include big data and real-time processing.