Message-Level Security In Restful Web Service Compositions

A PROJECT REPORT

Submitted in partial fulfillment of the requirements for the award of the degrees

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

Submitted by:

Bhor Verma, Kunal Gupta
150001005, 150001015

Guided by:
Dr. Abhishek Srivastava

INDIAN INSTITUTE OF TECHNOLOGY INDORE

November 2018

CANDIDATE'S DECLARATION

We hereby declare that the project entitled **"Message-Level Security In Restful Web Service Compositions"** submitted in partial fulfillment for the award of the degree of Bachelor of Technology in Computer Science and Engineering completed under the supervision of **Dr. Abhishek Srivastava**, Associate Professor, Computer Science and Engineering, IIT Indore is an authentic work.

Further, we declare that we have not submitted this work for the award of any other degree elsewhere.

Bhor Verma, Kunal Gupta
Date: 1 December 2018

CERTIFICATE by BTP Guide

It is certified that the above statement made by the students is correct to the best of my/our knowledge.

Dr. Abhishek Srivastava
Associate Professor
Discipline of Computer Science and Engineering
IIT Indore

Preface

This report on "Message-Level Security In Restful Web Service Compositions" is prepared under the guidance of Dr. Abhishek Srivastava, Associate Professor, Discipline of Computer Science and Engineering, IIT Indore.

Through this report, we have tried to give a detailed design on the architecture and implementation technique of a secure web composition. In this project, we have emphasized our focus on message-level security on RESTful architectural designs.

We have put our best efforts to explain the proposed design in a lucid manner. We have also added the figures and screenshots to make the setup, implementation and usage of the design more illustrative.

Bhor Verma, Kunal Gupta
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

ACKNOWLEDGEMENTS

We wish to thank Dr. Abhishek Srivastava for his kind support and valuable guidance. He was always available for the discussion, to answer our doubts and guide us through the different parts of the project. He provided an environment, where we were encouraged to discuss the new ideas and our problems.

Moreover, we would like to thank Mr. Gyan Prakash Tiwari for the help he provided related to research and ideas for the project. We are also thankful to our family members, friends and colleagues who were a constant source of motivation. We offer sincere thanks to everyone who else who knowingly or unknowingly helped me complete this project.

Without their support, this report would not have been possible.

Bhor Verma, Kunal Gupta
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

Abstract

As technology progresses, the use of IOT devices is growing at a breathtaking pace and secure communication of these devices amongst themselves or with other web service is a severe security concern. There are mainly two types of web services technologies, SOAP-based web services and RESTful web services and there exists a large number of researches for web services composition techniques for these web services technologies. In a web services composition the use of standard security practices (SSL/TLS) may fail to achieve the required protection in several cases, and hence instead of encrypting the whole communication, we need security at the message level. Few standards exist covering secure composition of SOAP-based web service but the same does not exist for the composition of RESTful web services. Although few pieces of research exist covering message-level security in RESTful web services, they do not cover applicability of such techniques in web services mashup. Unlike SOAP, RESTful web services do not require following a fixed format of communication and are much lightweight to use because of which the world is rapidly adopting RESTful web services for most of its tasks. The once popular SOAP-based web services are fast losing ground owing to this. In this report, we are proposing a method to create a secure dynamic group of RESTful web services providing various services and the combination of those services. All the members of a group will work towards the same goal. Service providers can join or leave this group dynamically. We are assuming that there can be a finite number of services provided by a group of RESTful web services and a group may contain an infinite number of service providers within it. A client can see the group as a single service provider, which provides a different combination of services and it can request for any combination of services from any service provider. We have implemented message-level security in web service composition so that messages under circulation amongst various service providers in the group is authenticated and encrypted.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Web services are meant for machine-to-machine consumption, mostly without human intervention, and should follow specific formats. A typical web service can return data in various formats such as XML, JSON or plaintext. A user generally interacts with a website, which may communicate with multiple web services to aggregate content retrieved from each of them and displays them back to the user.

Web service can be broadly divided into two categories: SOAP-based web services and RESTful web services. SOAP (Simple Object Access Protocol), designed in 1998 for Microsoft, is a messaging protocol specification, which deals with properly structured data in XML-based formats. In SOAP, a WSDL (Web Service Description Language) file provides the client with the necessary information, which can be used to understand what services the web service can offer [1]. Each SOAP message must contain some specific tags to maintain the defined formal structure. This results in increased message size and thus uses more bandwidth.

REST (REpresentational State Transfer) is an architectural style and not a protocol specification like SOAP. It defines a set of constraints for the creation of web services. It is now being widely used mainly because of its simplicity and uniform interface. It does not require strict formal formats like SOAP, which contributes to its *lightweight* nature, and thus data may be transferred in various formats including JSON and XML [2] [3]. Due to its flexibility, even newer upcoming formats, like ProtoBuf, can be easily utilized in such services. REST

requests consist of the HTTP verbs CRUD (Create, Read, Update and Delete) as operations to be executed on web resources [4].

Security is an important requirement in every web service composition and is crucial for the following purposes: confidentiality, integrity and authenticity. TLS (Transport Level Security) already has mechanisms for the encryption of the total data in a request, but not inside the request itself. It is good if data in a given request needs to reach a single destination, but not if it needs to be retrieved by intermediate nodes [5]. Now, message-level security is required to maintain security inside the request such that the data intended for a given node can be decrypted and read by that node only. There are existing methodologies in place for message-level security in SOAP but the same not defined for RESTful architectures.

1.2 Objective

To propose and implement a mechanism for message-level security in RESTful web service composition.

Chapter 2

Background

2.1 Example Scenario

Let us take a real-life example to show the necessity of a secure web service composition (Figure 2.1).

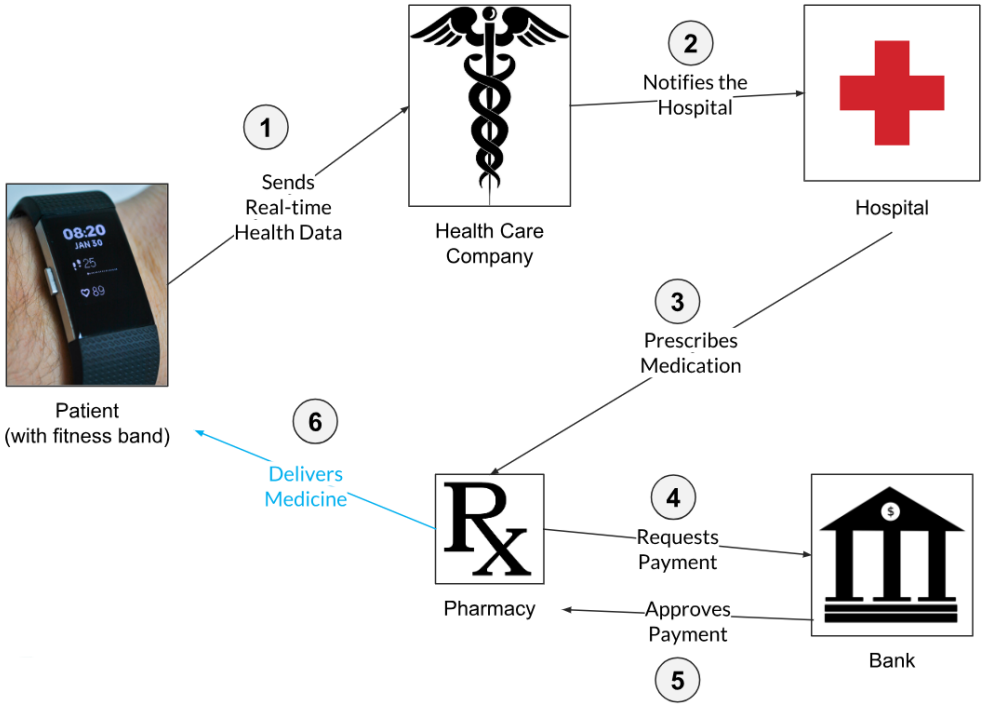

Figure 2.1: Real Life Example of Web Service Composition

There is a patient with a wristband (resembling an IOT client), and it is paired with a healthcare company. Under, the healthcare company, there are further hospitals connected to it, which have further connections to pharmacies. The patient's bank is also a part of this composition. The network works in the following way:

1) The wristband is constantly sending updates about the patient's health to the healthcare company.
2) The healthcare company then forwards the health data obtained from the wristband to the appropriate hospital upon detecting a health issue.
3) The hospital checks the health report submitted to it and after diagnosing the issue, it prescribes the required medicines. This prescription is then forwarded to one of the connected pharmacies with a request to deliver the same to the patient.
4) The pharmacy then contacts the patient's bank for the payment request of the prescription on behalf of the patient.
5) The bank then checks the patients credentials (which come from the wristband and after proper verification, it authorizes the payment request.
6) After the payment authorization, the pharmacy delivers the prescribed medication to the patient.

In this scenario, we wish to reduce the URL redirection as much as possible as it is not feasible for the low-computation-power wristband. A similar problem will arise on other IoT models as well, since most of the IoT device are have low computation-power, just like the wristband. In the security perspective, we cannot provide the bank account details of the patient to the healthcare company or the hospital or the pharmacy as it can lead to its misuse. Therefore, the wristband instead sends the bank account details along with the other health details. Now the challenge is to hide the personal details (bank account details) from the other nodes (healthcare company, hospital, and pharmacy) such that it is only readable by the intended node (bank).

2.2 Problems Faced

Let us consider the following web composition (Figure 2.2):

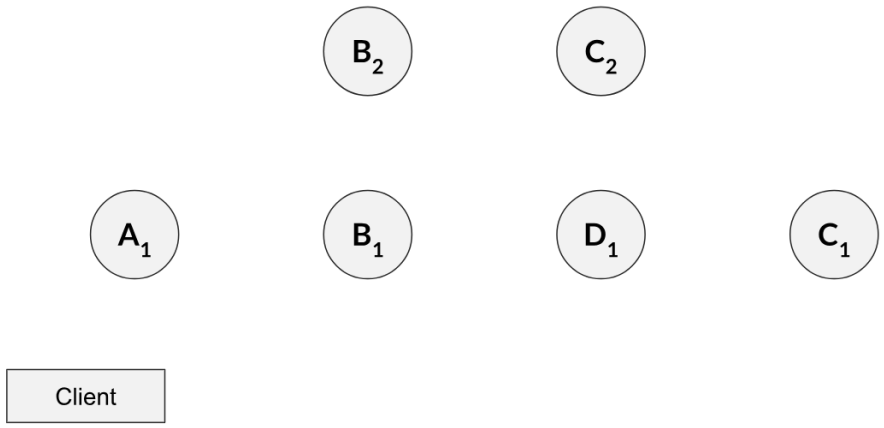

Figure 2.2: A Typical Web Service Composition

Here, the nodes (referred to as Service Provider Nodes or SPNs) and the provided services are defined as:

- Node A_1 provides service A
- Node B_1, B_2 provide service B
- Node C_1, C_2 provide service C
- Node D_1 provides service D

We will use a similar nomenclature for the further sections.

In the next subsections, we analyze the problems that can occur in creating such a composition.

2.2.1 Bouncing and Loops

Consider the previous composition (Figure 2.3):

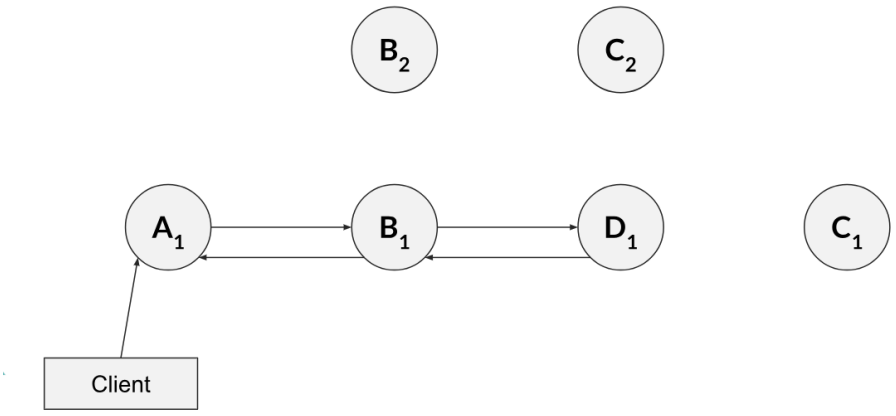

Figure 2.3: The Looping Problem

Let the client request the service AD from A_1. A_1 can provide service A, and it forwards this request is to B_1 to look for SPN of D which forwards it to D_1. Now D_1 oblivious to the fact that A_1has provided service A, and may forward the request back to $B_1$to look for SPN of A. Now if B_1send it back to A_1, it causes a loop and infinite bouncing of the request.

2.2.2 Selection of Service Provider

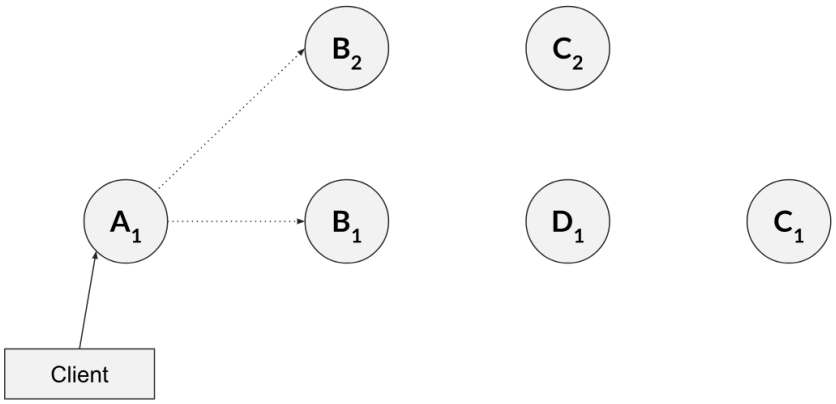

Figure 2.4: SPN Selection problem

If a client requests a service AB (Figure 2.4), the question comes up whether B_1 or B_2 should be selected to satisfy B out of the AB requested. Also when one of

them is initially chosen, can the other SPN be used for future requests? The answer to this indifference depends on the actual application of the composition.

2.2.3 Selection of Data Path

Client requests for service AC from A_1.

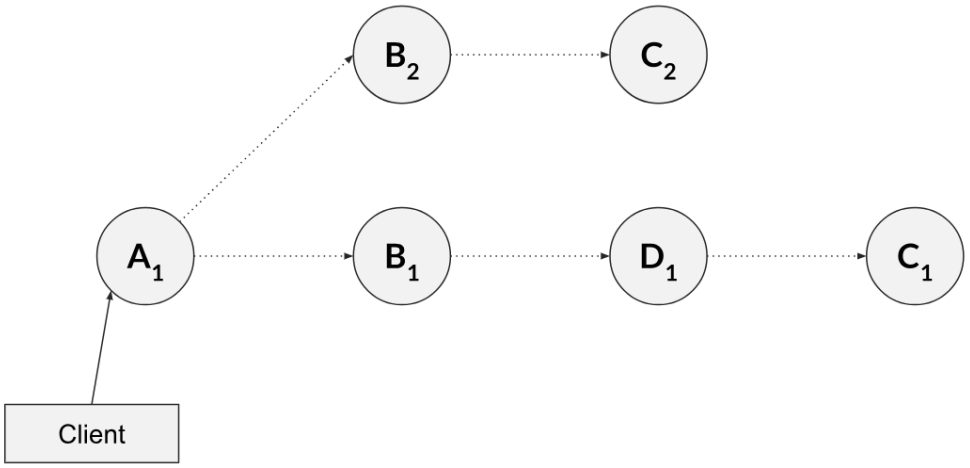

Figure 2.5: Selection of Data Path

There is a need to select pathing to provide C out of AC: $A_1B_2C_2$ or $A_1B_1D_1C_1$?

The answer again depends on the actual application and implementation of the composition to be used and may depend on multiple criteria such as:

- Load balancing
- Latency
- Congestion
- Registration of new service providers

Chapter 3

Previous Work

As previously discussed in Chapter 1, web services may be divided into SOAP and RESTful. This chapter discusses the existing methodologies and works pertaining to message-level security in SOAP.

3.1 XML-Enc

XML-Enc (XML Encryption) defines the specification to encrypt the contents of XML elements [6]. It is governed by a W3C recommendation. It uses the CBC (Cipher Block Chaining) mode of encryption. CBC works in the following way:

1. An initialization vector iv is chosen at random.
2. Let m_1 be the first message, then the first ciphertext $c_1$is obtained as:

$$c_1 = m_1 \oplus iv$$

3. The subsequent cipher texts, $c_i$is obtained for message m_i using:

$$c_i = c_{i-1} \oplus m_i$$

Thus, the current message is encrypted using the previous cipher text.

XML-Enc is known to have severe security concerns [7] due to the way CBC works, as CBC has already been shown to be malleable and breakable.

3.2 WS-Security

WS-Security (Web Services Security) [8] was published by OASIS (Organization for the Advancement of Structured Information Standards) and is an extension to SOAP. It describes three main mechanisms:

- Signing of messages to assure integrity

- Encryption of messages to assure confidentiality
- Attachment of tokens to confirm sender identity

While discussing WS-Security, technical details are kept out of scope and it is only used as a specification. It is up to how one implements the framework so that the result is not vulnerable.

WS-Security uses the XML SIG and XML ENC for signing and encryption of the XML messages. This causes a significant overhead in data transfer if message exchange is frequent. In addition, since this specification has no technical details involved, CBC mode may be used and it may lead to issues described in XML-Enc above.

Chapter 4

Problem Definition and Solution

4.1 Problem Definition

We need to define a secure dynamic web service composition, so that:

- Any other node within the network cannot read the information intended for a given recipient.
- Separate data intended for separate nodes can be sent in the same request with readability available to the intended node.

A secure web service composition should comprise of:

- Correct Message Structure
- Mechanisms to ensure the authenticity of the source of messages
- Decryption possible only at receiving (intended) end
- Fairly good encryption algorithms

4.2 Solution

In our work, we are creating a group of RESTful web services with a dynamic number of service providers working towards the same goal with the following goals:

- The number of services provided by this group is limited.
- A client can be connected to any service provider of the group as if it is connected to the whole group and using all its services.
- A client will see the group as a single service provider providing all the services.

The group formation starts with only one service provider. This first member of the group will be the most privileged and trusted member of the group and is

termed as Privileged Service Provider (PSP) and can be seen as the Certification Authority (CA) of the group.

Administrators are specially privileged SPN of the group and are added by the PSP. Administrators can accept new administrators with trust level greater than a threshold. An administrator can also accept other non-administrator SPN requests (i.e., not having administrative capabilities), but there is a maximum limit to the SPNs under any administrator.

Thus, the whole group is like a tree where nodes represent SPNs, parent nodes representing administrator SPNs, their children representing the SPNs respectively accepted by them and the PSP as the root node.

4.2.1 Routing of Requests

Each SPN's registry contains information about all the SPNs directly or indirectly registered under it.

Whenever a client needs a service *S*, it contacts one of the SPN, say SPN_i. SPN_i now looks for the SPN_s required for *S*:

- SPN_i, if a non-administrator and cannot provide *S*, forwards the request to the parent SPN_p. If SPN_i is an administrator, it looks for SPN_s as explained in the next point (like SPN_p).
- The parent SPN_p looks for SPN_s in its registry, if not found the request is forwarded to parent. If found, the request is forwarded to SPN_p's child having SPN_s in its registry.
- This process continues until SPN_s is found.

The complete routing process (Figure 4.1) can be summarized as finding the Lowest Common Ancestor (LCA) of the SPN_i and SPN_s and then going up from SPN_i to LCA and down to SPN_s as the whole network is like a tree.

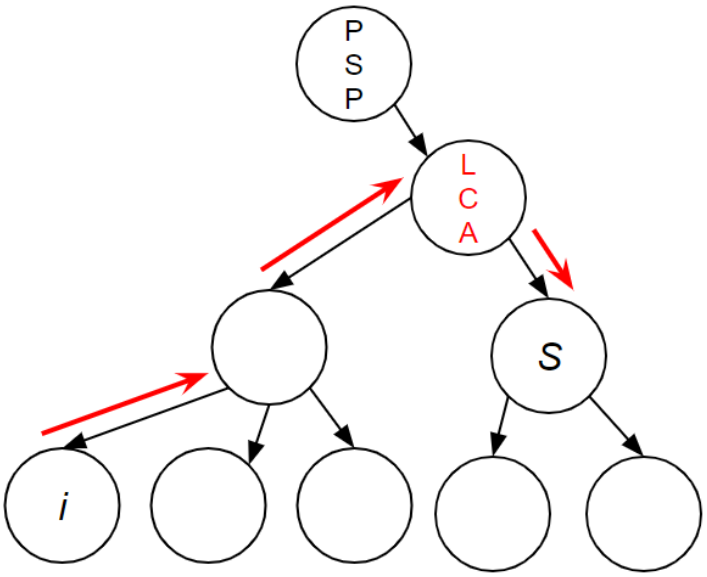

Figure 4.1: Routing via LCA

This actually solves the problems previously discussed in Chapter 2.

4.2.2 Addition of SPNs and Key-Exchange

The addition of an SPN (administrator or non-administrator) goes through the following process:

- When a new SPN tries to register in the network it sends a CSR (certificate signing request) to either one of the administrators or the PSP.
- The PSP either directly or indirectly (via the requested administrator) then validates the SPN using the trust level and then issues a certificate for the new SPN.
- Likes wise every SPN in the network will have a certificate issued by the PSP.

When a client requests a particular service *S*, suppose from SPN_i following steps takes place:

- The client sends a request to SPN_i for service *S* along with its public key
- SPN_i forwards the request to the appropriate service provider SPN_s by routing the request in the group (explained in the routing section). SPN_s

generates a new symmetric key (K_s) and encrypts it using client's public key [9] and sends back its PSP validated certificate along with the encrypted K_s.

- The client checks whether the received certificate is valid and upon successful validation, considers the symmetric key K_s sent by SPN_s via SPN_i.
- If the certificate is valid the symmetric key is successfully exchanged and now they both can start communication securely.

4.2.3 Message-Level Security

The client and SPN_s now have a common symmetric key K_s for service S.

Now, for each message requesting a set of services$(S_1, S_2, \ldots S_i)$, the data corresponding to each of the service will be encrypted by their respective keys $K_1, K_2, \ldots K_i$. This way each the message can only be read by the intended SPN only, thus providing an end-to-end encryption between each pair of communicating nodes.

4.3 Implementation

We implemented the project using Django [10] for REST API and Python's cryptography module [11].

The workflow is explained in the following sub-sections.

4.3.1 SPN Registration

4.3.1.1 CSR and Private Key Generation

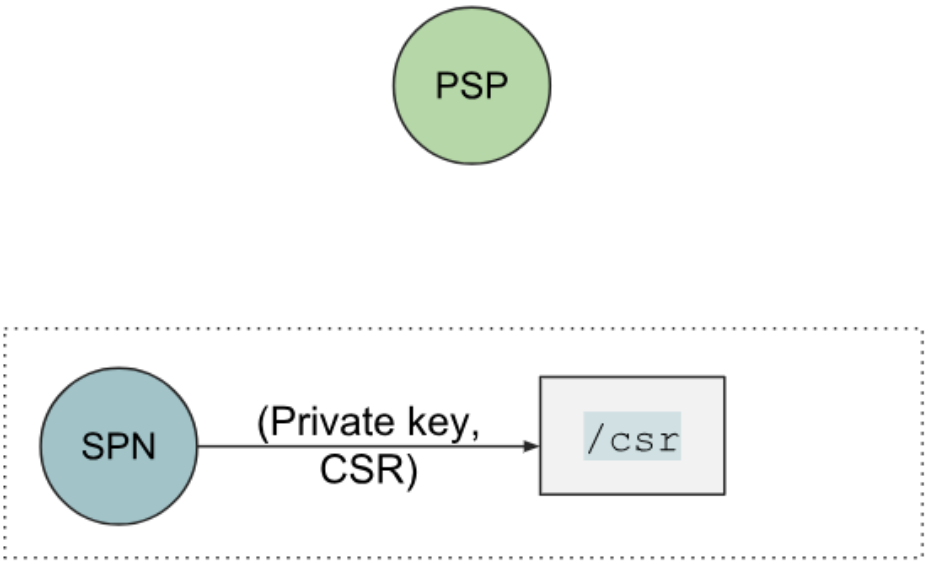

Figure 4.2: CSR and Private Key Generation.

Each new service provider looking to join the service composition visits its endpoint `/csr` where it generates its Certificate Signing Request by generating its private key (Figure 4.3).

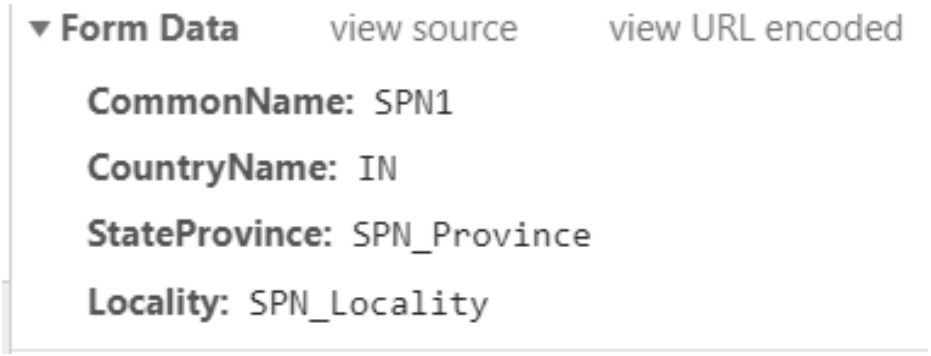

Figure 4.3: CSR POST Payload

The response is the CSR request and the generated private key (Figure 4.4).

{CSR: "-----BEGIN CERTIFICATE REQUEST-----↵MIICrjCCAZYC
AQ…ho+meebEoXQ0o2↵-----END CERTIFICATE REQUEST-----↵",
key: "-----BEGIN RSA PRIVATE KEY-----↵MIIEpAIBAAKCAQEAs
B…Pjf3b8t6EXBcI7MQ==↵-----END RSA PRIVATE KEY-----↵"}

Figure 4.4: CSR Response

4.3.1.2 Certificate Request

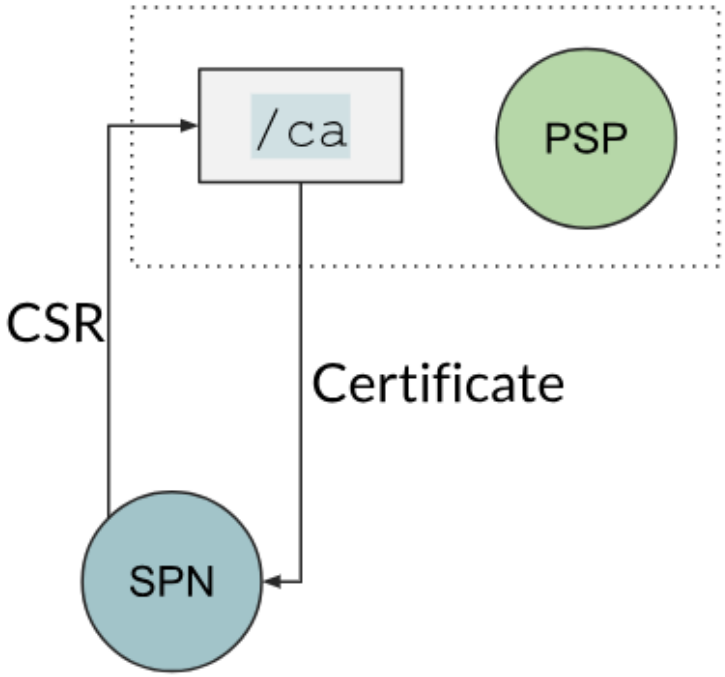

Figure 4.5: PSP responding with the certificate after receiving CSR

The CSR generated in the previous section is now used for Certificate Request to the CA via the `/ca` endpoint, which validates the CSR and issues a valid certificate (Figure 4.6) as well as the unique identification of the certificate.

{crt: "-----BEGIN CERTIFICATE-----↵MIIDKDCCAhCgAwIBAgIQ
dj…7oXNbiR9eUfqSRAwDUB7s=↵-----END CERTIFICATE-----↵",
serial: "763da898-926e-4152-bca3-8322940e5160"}

Figure 4.6: CA-issued Certificate

4.3.2 Symmetric Key Exchange

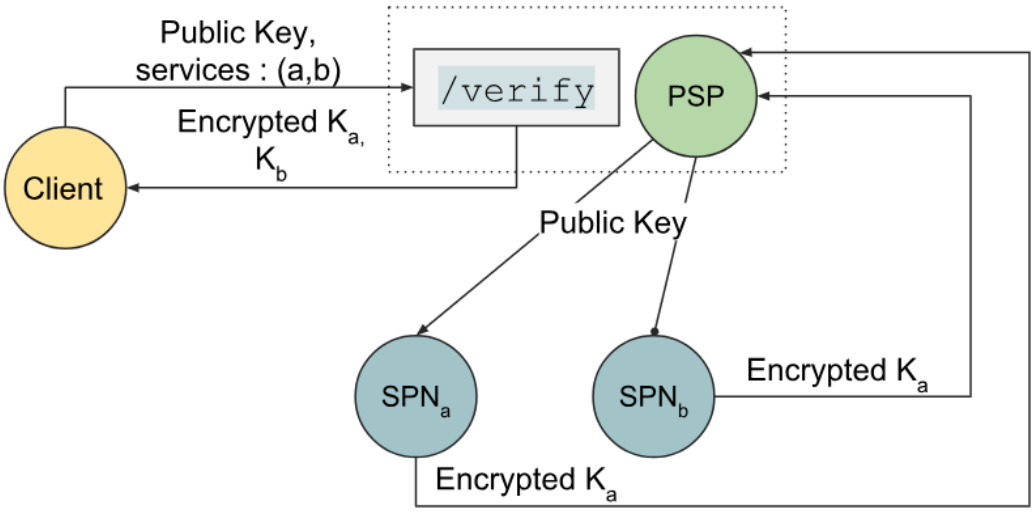

Figure 4.7: Exchange of Encrypted Symmetric Keys

The client sends its public key and the required services via the `/verify` endpoint using POST data (Figure 4.7 and Figure 4.8).

Form Data view source view URL encoded

services[]: a

services[]: b

pubkey:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAp5/wToLVCIfaCFr9MdfV
HAoj7ds1MtzT1ge1RHVzkXhbBtGXIfmlklHka6CQo4s/aOEJn33FtiGvLPDGqqc+
3h13rLN5A1WlFgR3LQMcHzYlQc5ebG6/GdEmcIM8xRJp2/8EalVOWjzOuW0OcYr9
90aVP89k4A48QJrCpoyiPjXb6o49Y7rsNnrp3+y0NFhUmm8e5i48/pwXuw2KcNrF
l4mq4mLyK/+ogkh/AoYNhROok8z3QUnzv7Y6bLztWK4aQFve83khLclBtySy2cvu
/ihGpQ9rGRRfbpni6eQUwr9vcUEF+wrItairkfXFtkooiRaXOwOa3Cf1hLSu8Q3B
ZwIDAQAB
-----END PUBLIC KEY-----
```

Figure 4.8: Client Request for Symmetric Key Exchange

The response (Figure 4.9) contains encrypted symmetric keys of SPN_a and SPN_b and PSP provides certificates $Cert_a$ and $Cert_b$ after their certificate verification (at the backend of the `/verify` endpoint).

{b: "WEqsrnKcojCEpR+NDU3gu5RahiZgrb8a/3EbdFBTAOOQx8bTUK…PY5KZz0LMd9XPmpIGf25nL3yfL4pSDnjlj1Age4D0xK/0wg==", a: "mshDC3HBn/IAJDq9NyYyRapC21kKUMJXPaGdJMb0XiUDOblrAS…TWtuxzohUNFQ+exnNEG5Z1Qd43BeVjLBJ4di+CXagZXzQgg==", verified: true, a_cert: "-----BEGIN CERTIFICATE-----↵MIIDHTCCAgWgAwIBAgIRAJ…Hbz8NIdLSMOfsuKAsK+PgW↵-----END CERTIFICATE-----↵", b_cert: "-----BEGIN CERTIFICATE-----↵MIIDGjCCAgKgAwIBAgIRAP…bzL4KYTw/E0HwmzDiV/L2P↵-----END CERTIFICATE-----↵"}

Figure 4.9: Response with Encrypted Symmetric Keys

The client verifies the authenticity of the SPN_a and SPN_b by validating it and can then decrypt the symmetric keys K_a and K_b using its private key.

4.3.3 Data Exchange

Now using the provided symmetric keys, the client can easily encrypt the data for SPN_i using the Key K_i, which can only be decrypted by SPN_i.

This data exchange can be done using a standard structure for data exchange so that parts of data meant for different SPNs can be easily distinguished (but not read) by the intermediate nodes.

The standard structure and its rules can vary greatly based on the application of the composition and hence we have avoided the specification of the same.

Chapter 5

Conclusion and Future Scope

Message level security is an important aspect of the currently evolving web services compositions as they become more and more complex. We analyzed the problems that can occur in RESTful web service compositions and studied previously existing solutions for SOAP and their tradeoffs. We also attempted to design and implement a real-life solution that caters to those problems as well as provides a secure environment in the composition such that message-level-security is insured.

This solution can be easily implemented in most use case scenarios with the flexibility of having application-focused data-exchange structure.

A more standardised approach is needed for looking into the message-level security across different types of web service composition to have set norms and implementations in place for the better security of the same.

Bibliography

[1] O. Zimmermann, M. Tomlinson and S. Peuser, "Perspectives on web services: applying SOAP, WSDL and UDDI to real-world projects," *Springer Science & Business Media,* 2012.

[2] C. Pautasso, O. Zimmermann and F. Leymann, "Restful web services vs. "big"' web services: making the right architectural decision," in *17th international conference on World Wide Web*, Beijing, China, 2008.

[3] M. Lanthaler and C. G.¨utl, "Towards a restful service ecosystem," in *4th IEEE International Conference on Digital Ecosystems and Technologies*, 2010.

[4] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," *University of California, Irvine Doctoral dissertation,* vol. 7, 2000.

[5] M. P. Singh, "XML and web services security standards," in *IEEE Communications Surveys & Tutorials 11*.

[6] T. Imamura, B. Dillaway and E. Simon, "XML Encryption Syntax and Processing," in *W3C Recommendation,* 11 April 2013.

[7] Association for Computing Machinery, "How To Break XML Encryption," 19 October 2011. [Online]. Available: https://www.nds.ruhr-uni-bochum.de/media/nds/veroeffentlichungen/2011/10/22/HowToBreakXMLenc.pdf. [Accessed 7 October 2018].

[8] B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama, A. Nadalin, N. Nagaratnam, H. Prafullchandra and J. Shewchuk, *Web Services Security (WS-Security).*

[9] W. Polk, R. Housley and L. Bassham, "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," April 2002. [Online].

[10] "Django," [Online]. Available: https://www.djangoproject.com/.

[11] "Python Cryptography," [Online]. Available: https://cryptography.io/.

Appendix A

Implementation of SPN Registration

In this section, we explain the python code used in the implementation in Section 4.3.1.

A.1 CSR Generation

The code in this section is executed on the requesting SPN's end.

Firstly, we generate the private key:

```
# Generate Private key
spn_private_key = rsa.generate_spn_private_key(
    public_exponent=65537, key_size=2048, backend=default_backend())
```

Next, we set up a builder object to generate the CSR request:

```
# Build a CertificateSigningRequest
builder = x509.CertificateSigningRequestBuilder()
builder = builder.subject_name(x509.Name([
    x509.NameAttribute(NameOID.COMMON_NAME, CommonName),
    x509.NameAttribute(NameOID.COUNTRY_NAME, CountryName),
    x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME,
StateProvince),
    x509.NameAttribute(NameOID.LOCALITY_NAME, Locality)
]))
builder = builder.add_extension(x509.BasicConstraints(
    ca=False, path_length=None), critical=True,)
```

Finally, we generate the CSR Request using the builder object:

```
# Sign CSR with SPN's_private_key using the builder object
CSRrequest = builder.sign(spn_private_key, hashes.SHA256(),
default_backend())
```

```
spn_csr = CSRrequest.public_bytes(Encoding.PEM).decode('utf-8')
```

A.2 Certificate issuing

The code in this section is executed on the CA's end.

The CA firstly loads the CSR from the request on the Django server:

```
# open the CertificateSigningRequest
pem_csr = str.encode(request.POST.get('csr'))
csr = x509.load_pem_x509_csr(pem_csr, default_backend())
```

CA's certificate and private key is loaded:

```
# load CA's certificate and Private key
cert_file = open(CA_CERT_PATH, 'rb').read()
ca_cert = x509.load_pem_x509_certificate(cert_file,
default_backend())
key_file = open(CA_PRIVATE_KEY_PATH, 'rb').read()
ca_key = serialization.load_pem_private_key(
    key_file, password=None, backend=default_backend())
```

We then initiate the builder object and generate a new certificate:

```
# Building Certificate
builder = x509.CertificateBuilder()
builder = builder.subject_name(csr.subject)
builder = builder.issuer_name(ca_cert.subject)
builder = builder.not_valid_before(datetime.datetime.now())
builder = builder.not_valid_after(
    datetime.datetime.now()+datetime.timedelta(100))
builder = builder.public_key(csr.public_key())
serial = uuid.uuid4()  # Random unique serial for each certificate
builder = builder.serial_number(int(serial))
for ext in csr.extensions:
    builder = builder.add_extension(ext.value, ext.critical)
```

Finally, the certificate for the SPN is signed by the CA:

```
# sign the certificate
certificate = builder.sign(
    private_key=ca_key, algorithm=hashes.SHA256(),
backend=default_backend())
spn_certificate = certificate.public_bytes(
    serialization.Encoding.PEM).decode('utf-8')
```