Enabling Message Security for RESTful Services

Gabriel Serme*, Anderson Santana de Oliveira*, Julien Massiera*, and Yves Roudier†
*SAP Labs France, France
Email: {gabriel.serme,anderson.santana.de.oliveira,julien.massiera}@sap.com
†Eurecom, France
Email: yves.roudier@eurecom.fr

***Abstract*—The security and dependability of cloud applications require strong confidence in the communication protocol used to access web resources. The mainstream service providers nowadays are shifting to REST-based services in the detriment of SOAP-based ones. REST proposes a lightweight approach to consume resources with no specific encapsulation, thus lacking of meta-data descriptions for security requirements. Currently, the security of RESTful services relies on ad-hoc security mechanisms (whose implementation is error-prone) or on the transport layer security (offering poor flexibility). We introduce the REST security protocol to provide secure service communication, together with its performance analysis when compared to equivalent WS-Security configuration.**

***Keywords*-REST, Performance, Message Security, Protocol**

I. Introduction

With the growing interest of cloud computing, systems are getting inter-connected faster, as applications and cloud API's make intensive usage of RESTful services to expose resources to consumers. There has been a shift from SOAP-based services to more lightweight communication, based on REST which allowed a number of advancements in the way resources are used on the web. As REST web services are self-described, resources can be manipulated through a set of verbs already provided in the communication protocol, accelerating the adoption of the REST philosophy. On the other hand, REST suffers from the absence of meta-descriptions, specially concerning security requirements.

Different solutions have been developed to provide a common way to address service description and communication. For SOAP-based web services, the standard defines envelopes to transmit requests and responses. In contrast, the REST concepts coined by Roy Fielding in his Ph.D. dissertation [1] simplify access to web services by reusing existing and widespread standards instead of adding new layers to the communication stack. The reuse of HTTP protocol contributed to the large industry adoption of RESTful services, supported by the simple CRUD set of operations (Create, Read, Update, Delete).

RESTful services suffer from the lack of a specific security model, unlike SOAP-based services which rely on the message security model defined in WS-Security [2] standard. Especially, the security of existing RESTful API's rely on transport layer security and on some home-made message protection mechanism. The former protects efficiently point-to-point communication channels, but becomes a burden for mobile systems, as the TLS channel need to be frequently reset. The latter can be error-prone, as security protocols are known to be tricky.

In this paper we provide a security protocol to make message security implementation as lightweight and efficient as possible, and yet to respect the REST principles. We show how message signature and encryption can address communication security for RESTful services at a fine-grained level. We present results of the benchmark we conducted on our implementation and compare it to the equivalent realization using SOAP and WS-Security.

The paper is organized as follows: in Section II, We present the REST security protocol and the threat model we aim to mitigate. In Section III we position our protocol with regards to WS-Security via a benchmark. Then we discuss related works in Section IV and conclude in Section V.

II. REST Security Protocol

In the following, we propose a protocol to secure communications for RESTful services. We provide *Encryption*, *Signature* and their combination. We do not aim to provide an equivalent of *Secure Conversation* for RESTful services, as it relates to some transport layer security for HTTP which is already addressed in protocols such as TLS.

A. Message Security Model

We specify an abstract message security model based on confidentiality and digital signatures to protect RESTful messages. The associated threat model is exactly the same as the one described in Web-Service Security standard [2]: "The message could be modified or read by attacker or an antagonist could send messages to a service that, while well-formed, lack appropriate security claims to warrant processing". For instance, a malicious attacker can intercept messages on any intermediary between peers. We want messages to carry tokens for non-repudiation (via digital signatures), to provide data confidentiality by encrypting its content, and to have replay attack protection.

B. PKI-based message exchange

We assume that a PKI landscape is in place and that certificates have been exchanged between clients and servers

prior to the communication. In this way we are able to transmit a certificate identifiers within the messages instead of full certificates, what would bring unnecessary overhead.

In order to distinguish a certificate on both client and server sides, we rely on a unique identifier, called *Certificate ID*, known to all entities. The *Certificate ID* is the aggregation of a serial number and an issuer name. The RFC 5280 [3] specifies that serial numbers "MUST be unique for each certificate issued by a given CA, i.e., the issuer name and serial number identify a unique certificate". The issuer name in our case can be represented by the Distinguished Name of a X509 certificate.

C. The REST Security principle

The principle of our protocol is to propose secure communication at the message level with the minimum overhead: we try to respect the philosophy of RESTful services and to reuse HTTP protocol to its full advantage. For example, we take into account the specificity of HTTP verbs in the design of the protocol. The REST security protocol is closely related to the WS-Security standard: it proposes a fine-grained approach to provide authenticity, non repudiation, and confidentiality to messages. But the approach targets another type of service. We claim that our approach is complementary to provide consistent application of security policies, disregarding the type of service being addressed. When comparing both approaches, we can highlight the reduced development effort and also less computation at runtime. This is a consequence of the optimization in the message size while we have performed, yet respecting the compatibility with service's definition and implementation.

We propose a set of HTTP-headers for transmitting meta-data, unlike WS-Security which modifies messages to add its own container describing the security meta data. The headers are described in Table I. They start with a prefix "X-JAG" to distinguish them from other application headers. The main difference with the WS-Security approach, is that we are agnostic about the information format. WS-* services use a strict approach to determine the transformations of XML-based messages to ensure the correct handling by interpreters at both sides. In our approach, we consider the information as a set of multiparts, and protocol headers. It allows us to gain flexibility in terms of fine-grained signature and encryption of attached documents, and/or to restrict visibility of a number of headers.

In the following, we present the REST security protocol process. For illustration purposes, we present the interaction trace produced by the request of a RESTful service in the Listing 1. A client requests customer information to the service and expects a JSON-encoded result. One can notice the expected result can be in any format accepted by the server (*e.g.,* XML, YAML, plain text, audio file, binary content, *etc.*). The response produced by the application server starts at Line 6 .

Header keys	**Value**
X-JAG-CertificateID	Unique identifier for a certificate
X-JAG-DigestAlg	Algorithm used to obtain digest
X-JAG-DigestValue	Value of the digest(s)
X-JAG-SigAlg	Algorithm used to obtain the signature
X-JAG-SigValue	Value of the signature(s)
X-JAG-EncAlg	Algorithm used to encrypt headers and messages' part
X-JAG-EncKeyAlg	Algorithm used to encrypt the symmetric key
X-JAG-EncKeyValue	Encrypted value of the symmetric key
X-JAG-MultiParts	Designation of headers and messages' part

Table I
REST SECURITY PROTOCOL HEADERS

```
1  GET /customer/123 HTTP/1.1
2  Accept: application/json
3  Host: 127.0.0.1:8080
4  Connection: keep-alive
5
6  HTTP/1.1 200 OK
7  Server: Apache-Coyote/1.1
8  Content-Type: application/json
9  Content-Length: 77
10
11 {"Customer":{"firstname":"Gabriel","id":123,"
       lastname":"Serme","title":"Mr"}}
```

Listing 1. RESTful request and response

D. Message Signature

Providing digital signature along with requests gives confidence on the data being transmitted. A server might need information on the authenticity of a message to launch internal orders and to render the service correctly. A digital signature brings non-repudiation: a requester cannot deny the request. Also, the service cannot later repudiate the response if it includes signed token linked to the initial request. Additionally, digital signature protects from unintentional or malicious modifications during the transmission.

Algorithm 1 presents the steps to attach signature information to the message after a "digest then encrypt" processing. It starts with a message m or part of it, with: the digest algorithm, the signature algorithm, the *Certificate Id* of the sender, and the private key of the sender. The algorithms can be decided by the sender itself, or imposed by the server policy. In our implementation, we allow the client to decide about the algorithm to be used, but the server can deny access if its policy considers the protection to be insufficient. We have defined a "digest then encrypt" function over the message payload, security parameters, and header information. The algorithm vary slightly depending on the concrete signature algorithm. The values are then attached to the message along with algorithm information.

In Algorithm 2, we present the signature verification function. It starts from a message m, or part of it, and with the public key of the sender. The steps are the reverse of the previous "digest then encrypt" algorithm. We first calculate the digest value of a set of headers and the payload. Then,

Algorithm 1 Signature of REST messages

Require: m is a message, sig is a signature algorithm name, dig is a digest algorithm name, cid is a *Certificate Id*, pk is the sender private key, $urlpath$ the requested path, hds are headers element to protect

$dv \leftarrow$ digest($m.payload$, dig)
$url \leftarrow$ ‘’
if m is a request **then**
  $url \leftarrow$ urlpath
end if
$bytes \leftarrow$ concat($dv, url, sig, dig, cid, hds$)
$digValue \leftarrow$ digest($bytes, dig$)
$m.sigValue \leftarrow$ encrypt($digValue, sig, pk$)
$m.\{url, sig, dig, cid, hds\} \leftarrow url, sig, dig, cid, hds$

we retrieve the digest value calculated by the sender. The encrypted value is transmitted along with the message, on a specific header. When we decrypt the value, we are then able to detect any corruption in the payload and headers but also to guarantee message safety and authenticity, as it has been digitally proved by the sender.

Algorithm 2 Verification of REST Signature

Require: m is a message, Pk is the sender public key

$dv \leftarrow$ digest($m.payload, m.dig$)
$bytes \leftarrow$ concat($dv, m.url, m.sig, m.dig, m.cid, m.hds$)
$calculatedDigest \leftarrow$ digest($bytes, m.dig$)
$retrievedDigest \leftarrow$ decrypt($m.sigValue, m.sig, Pk$)
if $retrievedDigest \equiv calculatedDigest$ **then**
  return true
end if
return false

The Listing 2 presents a HTTP trace with concrete headers and payload value. The request starts at Line 1 and the response starts at Line 10. We can observe for example that message request is issued by a sender identified as the 4102th certificate issued by the CESSA Authority. This sender protects the request of the customer 123. The response is given by another peer, identified as the 4th certificate issued by the CESSA Authority, on Line 12. The request and response are here signed, which allows the party consuming the message to verify the identity of the producer and the validity of the security token, to detect if the message has been tampered with. A replay attack can be avoided by binding the messages to elements with unique characteristics: MAC, timestamp , session related nonce, *etc.*.

```
1  GET /sign/customer/123 HTTP/1.1
2  Accept: application/json
3  X-JAG-CertificateID: CN=CA CESSA, <...>O=SAP
       Labs France, C=FR;4102
4  X-JAG-DigestAlg: w3.org/2000/09/xmldsig#sha1
5  X-JAG-DigestValue: 2jmj7l5rSw0yVb/vlWAYkK/YBwk=
6  X-JAG-SigAlg: w3.org/2000/09/xmldsig#rsa-sha1
7  X-JAG-SigValue: CwgrRTaC0oGBMpLPF6m<...>+
       gjtCMnuC+2svEdI5zJvITbM=
8  Host: 127.0.0.1:8080
9
10 HTTP/1.1 200 OK
11 Server: Apache-Coyote/1.1
12 X-JAG-CertificateID: CN=CA CESSA, <...>O=SAP
       Labs France, C=FR;4
13 X-JAG-DigestAlg: w3.org/2000/09/xmldsig#sha1
14 X-JAG-DigestValue: RUAYhPTuXqwChvIGrclAyRtA22Y=
15 X-JAG-SigAlg: w3.org/2000/09/xmldsig#rsa-sha1
16 X-JAG-SigValue: pmpc347XG/8a9QIFWYaHHsbt79hCwF
       <...>G/buHnjsHQvZhaggilRuM=
17 Content-Type: application/json
18 Content-Length: 77
19
20 {"Customer":{"firstname":"Gabriel","id":123,"
       lastname":"Serme","title":"Mr"}}
```

Listing 2. Signed request and response

E. Message Encryption

Message encryption provides confidentiality to sensitive assets so that no eavesdropping and data modification happen during messages transmission. In requests, several assets are transmitted, such as payload, session headers in cookies, etc. In our approach, we focus on payload and header protection mainly. We envisage extensions to address parameter encryption in GET requests in future versions of the protocol. The encryption has the property to modify the payload and headers, unlike signature which needs read-only access to the message. The encryption mechanism is also process-intensive.

The Algorithm 3 processes the payload of a message, or part of it for encryption. The PKI environment gives us mechanisms to share information between actors: the public and private keys. However, asymmetric algorithms are too heavy in order to perform an encryption on large amounts of data. Instead, we generate a symmetric key for encryption. This key is small enough to be encrypted with an asymmetric algorithm and sent with the message. Thus, the message contains an encrypted symmetric key for the receiver, the encrypted payload, and several headers expressing the algorithm used for encryption.

Algorithm 3 Encryption of a REST message

Require: m is a message, P_k is the receiver public key, enc is a symmetric algorithm name, $aenc$ is an asymmetric algorithm name, hds are headers element to protect

$skey \leftarrow$ generateSymmetricKey(enc)
$m.payload \leftarrow$ encrypt($m.payload, skey$)
for all $name, value \leftarrow hds$ **do**
  $hds[name] \leftarrow$ encrypt($value, skey$)
end for
$m.keyValue \leftarrow$ encrypt($skey, aenc, P_k$)
$m.enc, aenc, hds \leftarrow enc, aenc, hds$

```
1 GET /encrypt/customer/123 HTTP/1.1
2 Accept: application/json
3 X-JAG-CertificateID: CN=CA CESSA, <...>O=SAP
      Labs France, C=FR;4102
4 Host: 127.0.0.1:8080
5
6 HTTP/1.1 200 OK
7 Server: Apache-Coyote/1.1
8 X-JAG-CertificateID: CN=CA CESSA, <...>O=SAP
      Labs France, C=FR;4
9 X-JAG-EncKeyValue: RHvEjpmkt2QF3ZPCtqFbflDzA48
      <...>/UYNCYPbB265W2ZjYhL5VQSyv1Xs3Skm0=
10 X-JAG-EncAlg: w3.org/2001/04/xmlenc#aes128-cbc
11 X-JAG-EncKeyAlg: w3.org/2000/09/xmldsig#rsa-
      sha1
12 Content-Type: application/json
13 Content-Length: 101
14
15 eIdV39/XV/IHgPNWB2Hpo2jWglsI9p<...>k5c4+
      vVs9d53o6OEoh7M0bybmtGwdZE=
```

Listing 3. Encrypted payload during a request

The Algorithm 4 presents the reverse operation with respect to the above algorithm, to be executed on the receiver side. The procedure is performed on an encrypted message m or part of it. The message usually contains meta-information about encrypted parts and algorithms used for key encryption and data encryption. Otherwise, these information should result of a previous agreement between the sender and the receiver. To decrypt the data, the receiver retrieves the symmetric key and uses it to replace the headers and the payload.

Algorithm 4 Decryption of a REST message

Require: m is a message, p_k is the receiver private key
$skey \leftarrow$ decrypt($m.keyValue, m.aenc, p_k$)
for all $name, value \leftarrow m.hds$ **do**
$m.hds[name] \leftarrow$ decrypt($value, m.enc, skey$)
end for
$m.payload \leftarrow$ decrypt($m.payload, m.enc, skey$)

The Listing 3 presents a HTTP trace where the request does not contain custom information apart from the *Certificate Id*. The service has been configured to send back all messages encrypted. The service then processes and encrypts the message content for the requester. In the Listing, the payload is protected and no eavesdropping can be performed during the transmission. The protection mechanisms described in the previous section for replay attacks are also apply here.

F. Signature and Encryption

Signature combined with encryption is an important feature. Signature alone brings non-repudiation to the system, but an attacker can still read the content of messages and remain unnoticed. Providing encryption-only brings data confidentiality, but do not prevent against data tampering: any intruder can replace the payload and security tokens with

```
1 PUT /sign/customer/111/file HTTP/1.1
2 Content-Type: multipart/form-data; boundary="
      uuid:7d156074-35"; start="<root>";
3 X-JAG-CertificateID: CN=CA CESSA, <...>O=SAP
      Labs France, C=FR;4102
4 X-JAG-DigestAlg: w3.org/2000/09/xmldsig#sha1
5 X-JAG-DigestValue: 0;8X3Ci4M+bhWKMg+
      f83CXoXXjjns=
6 X-JAG-SigAlg: w3.org/2000/09/xmldsig#rsa-sha1
7 X-JAG-SigValue: 0;lcj7v4UAMxFOkhBoX+8<...>
      NKo393OQ=
8 X-JAG-Multiparts: 0;<root>
9 Host: 127.0.0.1:8080
10 Transfer-Encoding: chunked
11
12 --uuid:7d156074-35
13 Content-Type: application/octet-stream
14 Content-Transfer-Encoding: binary
15 Content-ID: <root>
16 Content-Disposition: attachment;filename=data.
      dat
17 <..binary content..>
18
19 <.. HTTP Response ..>
```

Listing 4. Multipart signature example

its own, as there is no binding with the proof of identity. For this purpose, the combination of encryption and signature at the message level provides confidence that data is kept confidential from intruders, and that no modification have been made to it. The signature testifies authenticity of the encrypted content, and only the receiver can retrieve the original data. In the current version of our work, we do not address ordering between the two mechanisms, therefore it is not yet possible to encrypt a signature.

G. Multiparts

We consider the case where one request or response message contains several parts. It is the case for example when forms are submitted with several fields containing user data, or when several files are attached along the same request. In such case, we might have general-purpose information and sensitive-information. To encrypt sensitive information, we need a mechanism that specifies the format of the different parts. We have several choices: we can apply the security requirements on the entire request/response of the RESTful service, or just on some parts/elements. HTTP makes usage of the Multipurpose Internet Mail Extensions (MIME) standard[1] to separate the content in several parts. We can take advantage of this usage to distinguish parts of the data along requests. Therefore, if a request contains multiple parts, we can choose to sign and encrypt some of them without affecting the others.

The approach differs from what is implemented in WS-Security standards and S/MIME standard. In our approach, we are independent from the actual content-type, and proposes to gather in one place all security meta-data. WS-* standards deal with XML-based content, so they propose a

[1] http://www.ietf.org/rfc/rfc2045

Processor	Intel Core i7-2600 @ 3.40GHz
Installed RAM	16 GB
Hard Drive	Seagate ST3500413AS Barracuda 7200 500 GB
Application Server	Tomcat 7.0.21
Server JVM Memory	-Xmx 8000m
WS framework	CXF 2.4.2
Server certificate	RSA 1024
Client's certificates	RSA 4096

Table II
BENCHMARK ENVIRONMENT

fine-grained approach at the XML-data level. Our approach is more general, and provides resource-grained encryption and signature. The Listing 4 highlights this principle. It represents the signature for the first multipart element identified by *<root>*. In a multipart environment, the meta-information vary depending on the part subject to encryption or signature. The header *X-JAG-Multiparts* contains a set of multipart elements and some headers referenced by identifiers. These identifiers are used to reference digest and signature values in the other security headers.

III. COMPARISON TO WS-SECURITY

The REST security protocol is close to the WS-Security standard. WS-Security [2] describes enhancements to SOAP messaging to provide protection through message integrity, confidentiality, and single message authentication. More precisely, it is an open format for signing and encrypting message parts leveraging XML Digital Signature and XML Encryption protocols, for supplying credentials in the form of security tokens, and for securely passing those tokens in a message. As explained in previous sections, the REST security protocol has been designed to be an equivalent alternative to WS-Security for RESTful services, with some differences in the way messages are secured.

A. Environment & Methodology

In order to position the protocol performance with respect to the state of the art, we have run several performance tests to compare WS-* and RESTful based services. In order to have a clear methodology and to reproduce performance tests, the evaluation has been made on the same environment to eliminate network side-effects. We limited resource starvation on the server to obtain accurate data. The Table II lists server characteristics. In order to compare the different services, we evaluate them in a single framework proposing coverage of both JAX-RS and JAX-WS specifications. The CXF service framework[2] allows us to compare the complexity of the two kinds of web services under the same conditions.

We have defined and implemented three scenarios, corresponding to real-use cases. In this way, we simulate several scenarios in order to evaluate and compare performance, message size, *etc.* The three scenarios correspond to:

[2]http://cxf.apache.org/index.html

Simple Get: In the following, we identify this scenario with the acronym *Get*. The scenario retrieves information without further processing. It is materialized by the invocation of a method in WS-* to retrieve customer information, from customer identifier. In RESTful services, the client requests a customer through a *GET* action, and the service renders the customer in the requested format.

Modify Post: In the following, we identify this scenario with the acronym *Post*. In this scenario, the data is transmitted in the request phase, and the response phase is just an indicator of the success or failure. Some additional processing is made on background to modify objects on the server. The modification of a remote resources is materialized by a method invocation with WS-* services, whereas it is a *POST* request in REST.

Large payload: In the following, we identify this scenario with the acronym *Large* or *Big*. It corresponds to the transmission of large amount of data between client and server. The size of messages brings out the real impact of the protocol. Each operation gives rise to accurate observation of the cost in terms of size and performance. It is materialized by a method invocation for a customer document in the input for WS-* services, and by a *PUT* request in the RESTful version. The reply contains indication of success or failure.

The different scenarios provide heterogeneous tests to verify several properties of the REST security protocol, in different conditions. They cover the most problematic situation one can face in a real production environment. They are a good basis for protocol comparison. For each of the scenarios, we have configured and run several tests with different security capabilities: signature, encryption, signature & encryption and no-security acting as the baseline. The experiments were performed couple of times to ensure consistent and valid results for comparison. The REST security implementation uses the same cryptographic algorithms as in the WS-Security configuration. For instance, both SOAP and REST services are set to use the “Basic128Rsa15” security algorithms suite: it determines the algorithms for digest, symmetric encryption, asymmetric encryption, as well as key derivation algorithms and key-wrap algorithms.

B. Size comparison

The Table III indicates the measurement in size to compare REST and WS-* services in the different scenarios. It lists the incoming and outgoing message sizes with distinction between headers and payload size. The results correspond to the different scenarios, with an equivalence between the *Get* and *Post* scenarios in terms of total size. The *Large* scenario sends a resource of around 3311kB. In the *Get* scenario, a client sends a request to the server in order to retrieve a *customer* object. In SOAP messages, the request is embedded in a SOAP envelope. The envelope

grows with the type of security used. For each type, the SOAP headers comprise secure data to indicate the type of algorithm, the encrypted or signed parts, and sometimes full certificates. In REST messages, the request is directly represented by the HTTP verb used to query the server. Therefore, no additional payload is necessary than the actual data plus some meta-data headers.

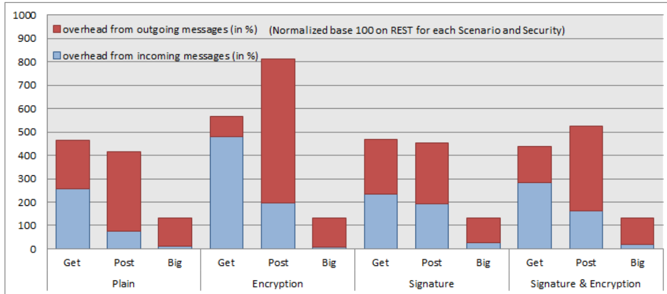

Figure 1. Overhead of SOAP messages compared to REST. For each scenario and security, the REST size represents the base 100

The Figure 1 highlights the global overhead using SOAP with any security mechanisms for the different scenarios. The REST size represents 100 for each scenario and security. We compare then the message overhead of different security mechanisms with its REST equivalent. For example, a SOAP-signed message size with the *Get* scenario represents around 460 when its counterpart in REST is 100. In the figure, we distinguish a second dimension: the origin of the overhead - from incoming message or outgoing message. The message increase for the previous scenario is half due to the incoming message, and second half by the outgoing message. In all tests, the usage of SOAP services instead of REST services is less efficient in terms of message size. The minimal overhead impact in all scenarios is 33%, which is the case where message payload is really large. We can explain it by the minimal impact of SOAP overhead compared to the actual data to transmit. This number is the result of our measurements, where the size of messages (including incoming and outgoing payload and headers) is larger when WS-* services are used compared to REST services, with all security mechanisms. The experimental cases where REST security protocol is the most efficient compared to WS-Security is on encryption of small set of data. The *Get* and *Post* scenarios present high SOAP overhead when data to transmit is small. For such cases, SOAP adds to much meta-data compared to the actual information, which multiply up to eight times the message size for a request and response in our measurements.

C. Processing performance comparison

In this paragraph, we present the processing performance comparison. The server has a certificate with RSA 1024 bits key, and the different clients have RSA 4096 bits. The difference of key size for the clients and the server impacts the time of processing depending on actions performed by the different actors. This behavior is directly linked to the performance of asymmetric algorithm that differs from encryption and decryption [4]. For instance, the encryption algorithm is straightforward has it uses a small value for the exponentiation (typically $0x10001$). The decryption algorithm requires more computation as the exponent is of the size of the private key (1024 or 4096 bits in our benchmarks). Thus, the server can decrypt faster than clients at the cost of less security. The calculated factor shows server decryption is around 20 times faster than client decryption. In our benchmarks, it impacts the performance comparison between the different scenarios we have defined. For instance, the server processes messages from the *Get* scenario with one encryption (fast operation) when messages from the *Post* scenario needs to be decrypted (slow operation) which lowers the processing time and throughput.

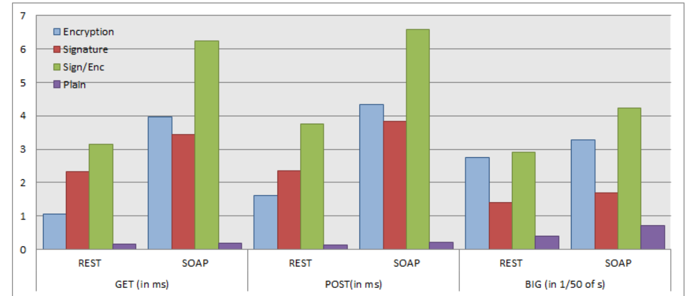

Figure 2. Average processing time comparison for the different scenarios

The Table III lists the average processing time calculated under the same conditions. Each scenario has been launched for 60 seconds, with a single client emitting requests. The client sends messages sequentially not to overload the server and to extract the optimal processing time. The Figure 2 depicts the differences between the different scenarios. The difference between REST and SOAP average processing time differs depending on the algorithm scheme and scenario used. In the *Get* and *Post* scenarios, REST is twice more efficient than SOAP when cryptography is used. It can be explained by the ratio of data related to XML format and SOAP meta-information that impact size of messages. For thin SOAP messages, the ratio doubles the size compared to REST messages. The time spent to process message is directly impacted by this size. For large messages, the encryption scheme is shown to be slower than signature.

We can notice differences in term of performance with regards to encryption and signature, depending on the size of data to be processed. Although SOAP encryption is always more costly than SOAP signature, REST shows better performances with encryption when amount of data remains low like in the *Get* and *Post* scenarios. If the data size growths, signature is faster than encryption.

		SOAP				REST			
		Enc	Sign	Sign&Enc	Plain	Enc	Sign	Sign&Enc	Plain
Get	Payload In	4991	5407	6849	765	0	0	0	0
	Headers In	228	228	228	221	330	1200	1200	192
	Payload Out	3685	3190	5149	827	236	161	236	161
	Headers Out	2	2	2	2	1001	531	1359	38
	Processing Time	3.964	3.445	6.244	0.187	1.060	2.332	3.156	0.145
Post	Payload In	5052	5458	6912	820	236	167	236	163
	Headers In	228	228	228	221	662	1216	1532	208
	Payload Out	3581	3107	5040	746	0	0	0	0
	Headers Out	2	2	2	2	193	551	551	58
	Processing Time	4.343	3.821	6.574	0.218	1.610	2.352	3.766	0.128
Large	Payload In (in kB)	5891 kB	4420 kB	5893 kB	4415 kB	4415 kB	3311 kB	4415 kB	3311 kB
	Headers In	228	228	228	221	815	1369	2052	361
	Payload Out	3565	3089	5028	734	0	0	0	0
	Headers Out	2	2	2	2	193	551	551	58
	Processing Time	164.412	85.063	211.377	35.510	137.950	69.468	146.043	19.776

Message size in Bytes & Average processing time in ms

Table III
COMPARISON OF PAYLOAD AND HEADER SIZE, AND AVERAGE PROCESSING TIME FOR SOAP AND REST MESSAGES

IV. RELATED WORK

In this section, we present some security models adopted by existing web services to expose their REST API's. Then, we provide alternative approaches to address REST security and performance issues.

The security model adopted by Amazon S3 [5] supports authentication and custom data encryption over HTTP requests. The requests are issued with a token to prevent unauthorized users from accessing, modifying or deleting the data. The token conveys a signature value calculated per request which transmits a proof of identity, ensuring the authenticity of the request, similar to our protocol. The data encryption can be performed by the client itself, or by the server prior storage. The communication is supposed secured through SSL endpoints. Our approach brings more flexibility as actors decide of resources and headers to protect and transform. The server benefits of the PKI environment to render services to its clients without the need to generate and maintain a set of secret keys. The clients can also enable the REST security protocol with different service providers by simply uploading their public key.

The other models adopt a slightly different approach, making intensive usage of the OAuth 2.0 protocol. Yahoo [6] uses OAuth Authorization protocol (OAuth Core 1.0 [7]) which is a simple, secure protocol to publish and share protected data when several actors require access to the resource. Yahoo demands the usage of an API Key to sign requests and provide end-user authentication. Twitter [8] leverages the transport layer security by exposing REST APIs over SSL. Facebook [9] requires the OAuth 2.0 protocol [10] for authentication and authorization. They distribute SSL Certificates to consumers so that they can create signed requests and force users to use HTTPS. The Dropbox model [11] allows third-party applications to use their services on behalf of users. Their model forces the requests through SSL and requires additional authenticity checks on messages. Like the previous approaches, they are combining transport layer security and application security. In our approach, we simplify the access of resources by unifying security at the message level. For instance, performing a request to retrieve a file with Dropbox transmits content metadata in an header. This content can be visible when the packet reaches the endpoint of a SSL tunnel, whereas our approach protects the header until its consumption.

The idea of having RESTful security as an equivalent of WS-Security has been expressed in a blog entry [12], using a similar approach but with no implementation and concrete specification. An approach to sign and encrypt multiparts have been drafted in [13]. They do not refer to REST services, but rather propose a model integrated to the multipart separation content to describe meta-information. Our approach benefits from multipart to split the payload in several resources, but we prefer centralizing security meta-data in headers to avoid service disruption, and to incorporate other field protection: headers, parameters, *etc.*. Our lightweight approach modifies content only when necessary.

Pautosso *et al.* [14] describe the differences between REST services and “big” services with a number of architectural decisions about which type of service is more appropriate. We have used this work to compare security of both approaches and to provide an extension to REST services for more security. The work in [15] addresses attacks targeting SOAP-based services. Although attacks are based on the XML message format, we advocate that the approach presented can be easily introduced in our implementation using particular header fields to inform about the document structure.

Optimizing service consumption in terms of performance has been addressed for a long time. The problem is rather

to balance usability and composability while allowing cross-cutting concerns such as security to protect the messages with a variable level of granularity. We can mention work on Fast Web Services [16] which defines binary-based messages to lower bandwidth and memory consumption. The price is the loss of self-description so that intermediaries cannot process the messages. In [17], Suzumura *et al.* propose a different approach, which is based on SOAP messages. They boost performance by considering partial regions of messages that differ from previously processed ones. Albeit the approach gives interesting results, they can not help with encrypted SOAP messages in the current state of the protocol.

V. Conclusion

In this paper, we have presented a novel approach to provide security for RESTful services equivalent to WS-Security. Our solution respects the REST philosophy by minimizing the processing overhead to service consumers, without interfering in the service composition already in place. We are able to keep messages confidential and to sign them with a fine granularity. The custom and ad-hoc processing on a per-message basis is a valid alternative to the existing approaches, which consider mainly transport layer security for securing all REST services. The advantage of our approach is to hide the complexity for the consumers, with no pollution on request parameters, while still carrying security tokens processable and verifiable by recipients.

We also conducted a performance evaluation considering several use-cases to analyze the impact of message protection to the performance of the web services. The analysis comprises heterogeneous scenarios to compare different security mechanisms among them, but also the behavior of the application server when dealing with RESTful services versus SOAP-based web services. The results show that RESTful services are processed more efficiently from any point of view, which is inherent to the service's purpose. RESTful services are oriented to handle resources, when SOAP-based services forge requests for operation invocation. The protocol is self-descriptive, so all information about the message verifications and transformations are specified to let the recipient informed about the message state.

As future works, we intend to provide several extensions. The protocol will handle other security constraints. For instance, we can think of carrying encrypted basic authentication tokens, signed P3P claims, or even convey authorization token decisions. We would also like to investigate an automated way to configure services to enable security transformations when necessary, i.e. when the resource is sensitive or contains restricted information.

Acknowledgment

This work was partially supported by the ANR project CESSA (Compositional Evolution of Secure SOA's with Aspects), grant number 09-SEGI-002-01.

References

[1] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[2] OASIS, "Web Services Security : SOAP Message Security 1.1," http://www.oasis-open.org/committees/wss, February 2006.

[3] IETF, "Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile," http://tools.ietf.org/html/rfc5280\#section-4.1.2.2, 2008.

[4] W. Dai, "Crypto++ 5.6.0 benchmarks," http://www.cryptopp.com/benchmarks.html, 2009.

[5] Amazon, "Amazon Simple Storage Service REST Security Model," http://docs.amazonwebservices.com/AmazonS3/latest/dev/RESTAPI.html, 2006.

[6] Yahoo, "OAuth Authorization Model," http://developer.yahoo.com/oauth/.

[7] IETF, "The OAuth 1.0 Protocol," http://tools.ietf.org/html/rfc5849, 2010.

[8] Twitter, "Security Best Practices," https://dev.twitter.com/docs/security-best-practices, 2011.

[9] Facebook, "Facebook Authentication," http://developers.facebook.com/docs/authentication/, 2012.

[10] IETF, "The OAuth 2.0 Authorization Protocol," http://tools.ietf.org/html/draft-ietf-oauth-v2-23, 2012.

[11] Dropbox, "REST API," https://www.dropbox.com/developers/reference/api, 2012.

[12] F. Lascelles, "RESTful Web services and signatures," http://flascelles.wordpress.com/2010/10/02/restful-web-services-and-signatures/, October 2010.

[13] J. Galvin, S. Murphy, S. Crocker, and N. Freed, "Security multiparts for mime: Multipart/signed and multipart/encrypted," IETF, Network Working Group, Tech. Rep., October 1995, http://tools.ietf.org/html/rfc1847.

[14] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big" web services: making the right architectural decision," in *WWW*. ACM, 2008, pp. 805–814.

[15] M. A. Rahaman and A. Schaad, "Soap-based secure conversation and collaboration," in *ICWS*. IEEE Computer Society, 2007, pp. 471–480.

[16] P. Sandoz, S. Pericas-Geertsen, K. Kawaguchi, M. Hadley, and E. Pelegri-Llopart, "Fast web services," *Sun Developer Network*, 2003.

[17] T. Suzumura, T. Takase, and M. Tatsubori, "Optimizing web services performance by differential deserialization," in *ICWS*. IEEE Computer Society, 2005, pp. 185–192.