Granular Confidentiality and Integrity of JSON Messages

Tiago Santos, Carlos Serrão
ISCTE – Instituto Universitário de Lisboa
Information Sciences, Technologies and Architecture Research Center (ISTAR-IUL)
Ed. ISCTE, Av. das Forças Armadas, 1649-026, Lisbon, Portugal

Abstract

Modern web and mobile-based applications exchange information with each other and with other services, through specific APIs that extend the applications multipart functionality and enable interoperable information exchange. Currently these mechanisms are implemented through the usage of RESTful APIs and data interchange is performed using the JSON format over the HTTP or HTTPS protocol. Most of the times, due to specific security requirements, the SSL/TLS protocol is used to create a secure authenticated channel between the two-communicating service end-points, where all the content is encrypted. This is an important security feature if the sender and the receptor are the only communicating parties, however this may not be the case. In this paper, a granular mechanism for selectively offering confidentiality and integrity to JSON messages, through the usage of public-key cryptography is presented. The proposed mechanism, as take in to consideration already existing mechanisms, such as XML security, to best fit developers' acquaintance. In this paper, we will present the proposal of the syntax for the secure JSON format (SecJSON) and present a prototype implementation of that particular specification that was created to offer developers, written in Javascript and Node.JS, the possibility to offer this security mechanism into their own services and applications.

1. Introduction

Current web and mobile development follows a paradigm where most of the software development is encapsulated into self-contained entities, referred as services. Services expose standardized interfaces (API), using some existing mechanisms, to interact with other services or systems, in order to provide specific functionalities for their users. For instance, imagine a mobile application that uses the Facebook service to allow its users to update their Facebook account and uses the Weather.com service to inform its users about the weather on a given geo-location [1]. The usage of such services involves the definition of their internal functionality, the communication mechanisms and the data interchange formats that are required by the service and the service invokers. The Internet, in particular the World Wide Web, presented the opportunity for the development of standard communication environment that facilitated the service-oriented software development and deployment [2].

In modern web-based service-oriented software, one of the main mechanisms that is used to create information exchange interoperability between different Web-based services uses the Javascript Object Notation (JSON), an open standard format that uses plaintext to facilitate the transport, processing and interoperability during information serialization and de-serialization [3] cross multiple heterogeneous services and applications. According to its creator, Douglas Crockford, JSON is a natural way for representing data that can be consumed by different programming languages and different platforms or architectures [4]. In this service-oriented development model there are commonly the SOAP-based and REST-based services. SOAP relies entirely on XML to provide messaging services. It was developed as a replacement for older technologies such as Distributed Component Object Model (DCOM) and Common Object Request Broker Architecture (CORBA) that were based on binary messaging not working well over the Internet. SOAP was standardized and is part of a set of Web Services Standards. XML is used to make requests and receive responses in SOAP and this can become extremely complex. An important part of the SOAP-based web services is the Web Services Description Language (WSDL). WSDL is used to describe how a service works and what is the format of the messages and it expects to receive and send. SOAP is independent of the transport protocol and is not dependent of the HTTP protocol [5]. However, a large number of developers found SOAP cumbersome and hard to use, in particular due to the XML complexity and verbosity.

REST-based services are a lightweight alternative, using simple mechanisms such as simple URLs, Really Simple Syndication (RSS), Comma-Separated Values (CSV) or JavaScript Object Notation (JSON) to provide the communication and data exchange methods to use the service. REST-based services are dependent of the HTTP protocol using the HTTP verbs (GET, POST, PUT and DELETE) in order for the service to perform tasks. JSON is currently one

Copyright © 2017, Infonomics Society

of the common options to exchange information on REST-based services, due to its simplicity. JavaScript Object Notation (JSON) is a text format for the serialization of structured data described in RFC 4627 [4]. The JSON format is often used for serializing and transmitting structured data over a network connection.

One of the first JSON implementations targeted the communication between Javascript-based scripts and Java-based servers. Although JSON was first developed having into consideration the Javascript language, it is currently platform and programming language independent. In the last few years there has been a significant growth in the usage of this format to serialize and de-serialize information on web services, promoting the data interoperability between services running on different platforms and written on a multiplicity of programming languages. JSON can be seen today, together with HTTP, as the "glue" that enables the interoperable communication between different web-based services [6] and applications (desktop, web or mobile centric). JSON is widely used to support the communication between multiple REST-based service APIs available on web. Due to the increasing adoption of this type of REST-based web-services and JSON data interchange format, JSON security assumes extreme significance, in particular, due to the sensitive characteristics of the information that is JSON-encapsulated (also known as JSON payload) and transported between this distributed heterogeneous ecosystem.

Due to the widespread and openness of the Internet, there are currently mechanisms that allow the protection of the communication channels between the different applications and services assuring the confidentiality and authentication of the entire channel – the Secure Sockets Layer/ Transport Layer Security protocol (SSL/TLS) [7]. SSL/TLS are cryptographic protocols that offer communications security over a network, ensuring that the connection is private, the identity of the communicating parties can be authenticated and the integrity of the exchanged messages can be established. However, SSL/TLS blindly ciphers all the information that flows on the communication channel, in the same similar way. This is a limitation that makes impossible to cipher parts of message with a key and other parts of an SSL/TLS message with a different cryptographic key. Therefore, all the messages sent from a specific sender, are encrypted with the appropriate cryptographic key, in order to be decrypted by a particular receiver – the encryption is always end-to-end.

This is an SSL/TLS characteristic that is adequate for two entities secure authenticated communication, but it is not adequate to offer the possibility of ciphering the same message conditionally (for instance JSON or XML data), using different keys or using different protection mechanisms (cryptographic algorithms), which could be required by specific applications and by different users [8]. There may exist situations in which the information that needs to be sent or routed to multiple entities, even if those entities are not the final receptor of such message. Therefore, it should exist a mechanism that would allow the same JSON message/document to have multiple sections of that document that are protected in a specific manner, while others have a different protection type. With these requirements in mind, it is possible to imagine a scenario where the same JSON document can contain critical and non-critical information, protected in different ways, with distinguished ways of accessing such information (see Figure 1).

In the depicted scenario, a single payload of JSON-formatted data, contained inside the JSON structure is protected using different protection mechanisms, that are adequate for different applications and different users. The same message is sent to multiple receivers however, only the receivers with the appropriate decryption mechanisms and decryption keys are able to access the JSON data that is intended for them.

This article intends to present a secure and granular solution for the protection of confidentiality and integrity of JSON documents. The major contribution of the work presented in this article can be resumed in the presentation of the syntax and semantics of a mechanism capable of ensuring the granular confidentiality and integrity of JSON objects and the implementation of the syntax necessary to support the security mechanisms necessary. Other important contribution of this work consists of the implementation of a software library that will enable developers implementing web-services to be able to use these JSON security functionalities in an easy and straightforward manner. The article starts by providing an introduction to the modern approach to the development of distributed web services. After this, a more detailed presentation of the HTTP-based RESTful services is provided, as well as some references to the data interchange format that is currently being used on these cases, and some problems involved in the security of JSON. Following this part, a proposal and specification of a secure version of JSON (SecJSON) is provided. The following section provides a description of the implementation that was conducted to implement a library that would allow web-services developers to use the SecJSON format. Finally, some conclusions from the work are presented as well as some of its limitations.

Copyright © 2017, Infonomics Society

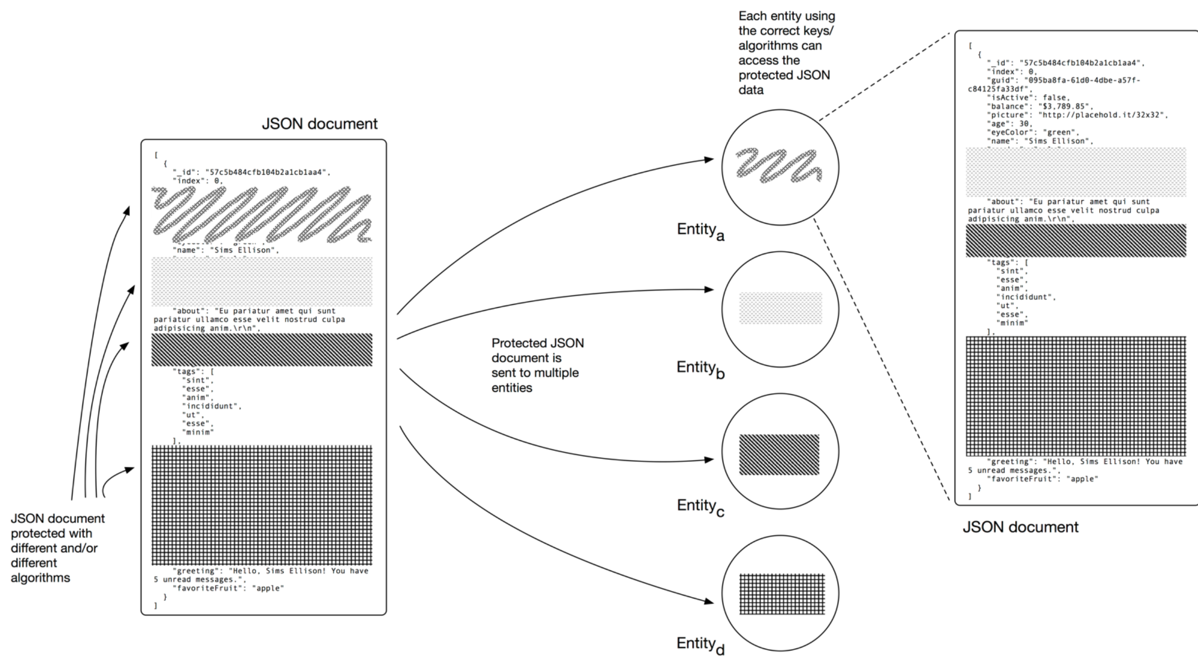

Figure 1. Scenario of the granular security of JSON interactions

2. JSON-BASED Web Services

Most business transactions currently depend on the existence of Web Services. More and more developed applications are following a service-oriented approach. This is the reason why it has become one of the most important areas of the IT industry [9]. The security inherent in this type of transactions is essential to ensure the success of an organization and automate most of their internal and external business processes. The possibility for organizations or users to interact directly with other organization's systems over open networks raise security concerns. How can organizations ensure that their own information or the information of their users reaches the final destination safely, preserving confidentiality and integrity, whenever sensitive information is routed through the WWW [9]. Looking at the state of the art, it is possible to identify different protocols and technologies to ensure the security and confidentiality on the Internet/WWW, each one of them using their own ways to protect information. One of the most used web protection mechanisms is SSL/TLS. As it was previously stated, the main functionality of the SSL/TLS protocol is to establish an encrypted and authenticated communication channel between two communication parties - the client, usually a web browser and a server.

However, as previously referred, this mechanism encrypts all information passing through the communication channel, using pre-established cryptographic primitives and keys, in the same way. Therefore, it is impossible, in a conditional and granular manner, to encrypt JSON messages, or parts of messages, with different keys or encryption schemes. This constraint can be a problem for specific use cases. The focus of SSL/TLS protocol consists in the protection of information serialization between two entities. Information is immediately deciphered on arrival at the end-point, regardless of their final destination [10]. In the case of a channel compromise, all information transmitted can be accessible to an attacker. Moreover, SSL/TLS is mostly used at the server level and not the application level – meaning that information is decrypted at the server and not at the application. In a scenario where a server is running multiple applications, with multiple users, and each of them have their specific security requirements, SSL/TLS might not be the appropriate solution to offer confidentiality and integrity to JSON messages in this case. In addition to these problems, in a scenario where sensitive JSON information is forwarded by multiple parties without them to be the final recipient of the information, if one of the parties is compromised all the information can be exposed. In this scenario, the protection of the JSON messages offered by SSL/TLS protocol is insufficient.

There are already some specific technologies for providing the security of JSON data. One of the most prominent initiatives in this field is the Javascript Object Signing and Encryption (JOSE). JOSE is a

Copyright © 2017, Infonomics Society

framework that was developed with the intention to provide a method to securely transfer claims (such as authorization information) between parties [11]. The JOSE working group standardized a mechanism to offer integrity protection (signature and MAC) and encryption as well as the format for keys and algorithm identifiers to support interoperability of security services for protocols that use JSON [12]. JOSE is currently mostly used for digital identity identification (as an alternative or a complement to OAuth) and is composed by a set of different specifications: JSON Web Token [13], Signature [14], Encryption [15], Key [16] and Algorithm specifications [17]. For developers, in particular those already involved on service-oriented software development, this means having to use a new specification and increase their learning curve. This way, for some cases, it would be better to have a lightweight approach to the JSON security problem, and to base its development on something that was already existing and more mature, such as the XML web-services security standards (WS-Security) [18]. Considering this requirement and the existing WS-Security, the Secure Javascript Object Notation (SecJSON) was developed.

3. Secure Javascript Object Notation (SECJSON)

Considering the different aspects of modern JSON documents confidentiality and integrity, and the mechanisms that are mostly offered for security on the WWW, it is possible to conclude that SSL/TLS is not suitable for all the security scenarios involving JSON. Therefore, this work was conducted to devise a security framework that could be used to offer JSON protection, in a way that it would be easy for programmers to use to implement security on their services. This section of the paper presents some of the major requirements guiding the development of SecJSON as well as the description of the approach that was followed throughout its development. The SecJSON syntax is also presented.

3.1. SecJSON requirements

The basic rational behind the specification and development of SecJSON is to assure a security mechanism that would enable the protection of JSON data. The specific requirements of the solution can be resumed in the following:

- SecJSON should offer a protection mechanism that is independent of any other existing channel encryption mechanism – this means that SecJSON can act as a security mechanism that can be used on top (at the application level) of other underlying security mechanism, such as SSL/TLS;
- SecJSON should consider the protection of JSON data without any underlying channel encryption mechanism (for instance, SSL/TLS). This means that even if the communication channel is not encrypted, SecJSON should provide the security mechanisms to offer the appropriate protection to JSON;
- SecJSON should assume that data inside the JSON document/message could have as destiny different receptors with different access clearances;
- SecJSON should make possible to protect either the entire JSON document/message or simply protect specific parts of the JSON document/message – offer granularity in terms of protetion;
- SecJSON should also make possible the usage of multiple keys and multiple encryption algorithms to protect different sections of the same JSON document/message;
- SecJSON should be independent of any specific programming language, or encryption algorithms;
- SecJSON should be easy to implement and used by any third parties;
- Finally, SecJSON would be open and free to use by anyone.

Considering the set of identified requirements, SecJSON was specified and developed. The following sections of this article present the SecJSON specification and the implementation that was performed to allow developers to integrate SecJSON into their own development lifecycle.

3.2. SecJSON overview

The proposed Secure JSON consists in a set of rules and specifications for encrypting information and represent their results in JSON format. Data to be protected can include another JSON document, a primary type (for instance, a sequence of characters) or a structured type (for instance, an array).

SecJSON is a mechanism that was based on the XML Encryption standard, which specifies the method for encrypting data and how it can be represented in XML format [19].

The result of the encryption process consists of a SecJSON element EncryptedData, which contains encrypted information.

```
{
  "Case":"Case info",
  "Witness protection":[
    {
      "Name":"Igor",
      "id":123
  }]
}
```

The previously presented JSON object, contains sensitive information about witnesses, that needs to be protected. In an initial stage it should be identified

Copyright © 2017, Infonomics Society

where is the information that will need to be encrypted (in this case the "Witness protection" element):

```
{
  [
    "Name":"Igor",
    "id":123
  ]
}
```

After SecJSON cipher process is applied to the previously located element, it is replaced by the appropriate EncryptedData element. This element contains all necessary components to allow the SecJSON decipher process. The result is similar to the following object:

```
{
  "Case":"Case info",
  "Witness protection":{
    "EncryptedData":{
      (... SecJSON elements ...)
    }
  }
}
```

Whenever the encryption process is applied to a JSON document/message the result is a new JSON-encrypted document with a single `EncryptedData` element.

```
{
  "EncryptedData":{
    (... SecJSON elements ...)
  }
}
```

3.3. SecJSON proposed syntax

This section offers a detailed description of the syntax and features of the Secure JSON (SecJSON). The syntax is defined using the JSON-Schema in order to be similar to what occurs in the XML security. The JSON implementation should generate a JSON object accepted and validated by the JSON Schema defined and available in http://tiagomistral.github.io/SecJSON/ secjson-schema.json.

`EncryptedType` element

`EncryptedType` is the abstract type from which `EncryptedData` and `EncryptedKey` are derived. While these two element types are very similar with respect to their content models, a syntactical distinction is useful for processing these elements.

Although JSON Schema does not support abstract elements, a representation of this element is useful to facilitate the interpretation of the syntax.

`EncryptionMethod` element

`EncryptionMethod` is an optional element that describes the encryption algorithm applied to the original data to obtain the ciphered counterpart. If the element is absent, the encryption algorithm must be known by the recipient or the decryption will fail.

`CipherData` element

`CipherData` is a mandatory element that provides the encrypted data. It must either contain the encrypted octet sequence as a Base64 encoded text of the `CipherValue` element, or provide a reference to an external location containing the encrypted octet sequence via the `CipherReference` element.

`CipherReference` element

If `CipherValue` is not supplied directly, the `CipherReference` identifies a source which, when processed, yields the encrypted octet sequence.

The actual value is obtained as follows. The `CipherReference` URI contains an identifier that is dereferenced. Should the `CipherReference` element contain an optional sequence of `Transforms`, the data resulting from dereferencing the URI is transformed so as to yield the intended cipher value.

`EncryptedData` element

The `EncryptedData` element is the core element in the JSON encrypted structure syntax. Not only does its `CipherData` child contain the encrypted data, but it is also the element that replaces the encrypted element, or serves as the new document root.

`KeyInfo` element

There are two ways that the keying material needed to decrypt `CipherData` can be provided:

- The `EncryptedData` or `EncryptedKey` element specify the associated keying material via a child of KeyInfo element.
- The keying material can be determined by the recipient by application context and thus need not be explicitly mentioned in the transmitted JSON document.

`EncryptedKey` element

The `EncryptedKey` element is used to transport encryption keys from the originator to a known recipient(s). It may be used as a stand-alone JSON document, be placed within an application document, or appear inside an `EncryptedData` element as a child of a `KeyInfo` element. The key value is

always encrypted to the recipient(s). When `EncryptedKey` is decrypted the resulting octets are made available to the `EncryptionMethod` algorithm without any additional processing.

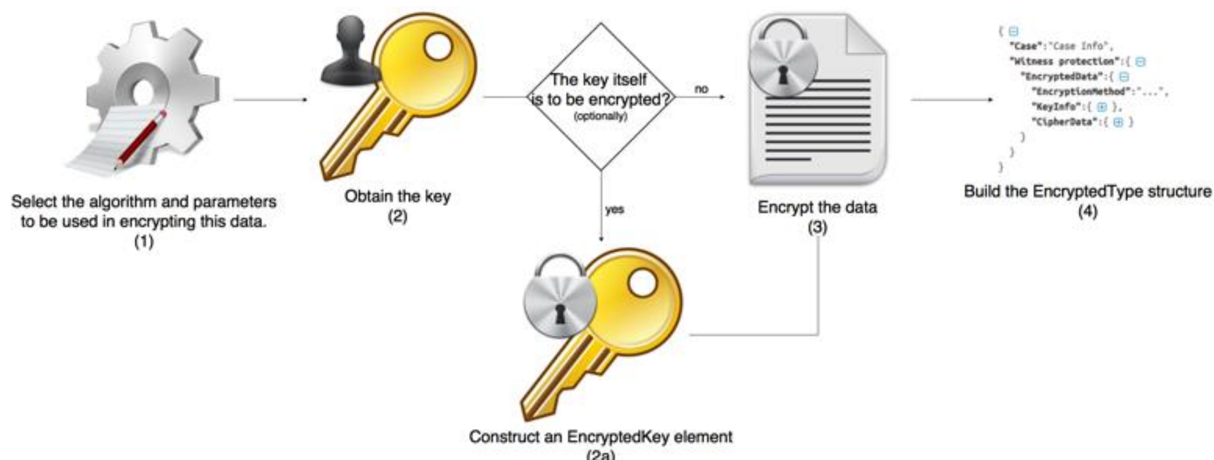

Figure 2. SecJSON encryption process

3.4. SecJSON Processing Rules

This section describes the operations that need to be performed as part of the encryption and decryption processing by any implementation of the SecJSON specification. Again, in a similar way as it occurred in the the definition of SecJSON elements, the SecJSON processing rules are based on the same rules that are used by XML Encryption standard [19].

The conformance requirements are specified over the following roles:

Application: the application which makes a request of an SecJSON implementation via the provision of data and parameters necessary for its processing;

Encryptor: a SecJSON implementation with the role of encrypting data;

Decryptor: a SecJSON encryption implementation with the role of decrypting data.

For each data item to be encrypted (**Error! Reference source not found.**) as an element derived from `EncryptedType`, the **encryptor** must:

1. Select the algorithm (and parameters) to be used in encrypting this data.
2. Obtain and (optionally) represent the key.

a. If the key is to be identified (via naming, URI, or included in a child element), construct the `KeyInfo` as appropriate.

b. If the key itself is to be encrypted, construct an `EncryptedKey` element by recursively applying this encryption process. The result may then be a child of `KeyInfo`, or it may exist elsewhere and may be identified in the preceding step.

3. Encrypt the data:

a. obtain the octets by serializing the data in UTF-8 (or other encoding choose by **application**). Serialization may be done by the **encryptor**. If the **encryptor** does not serialize, then the application must perform the serialization.

b. Encrypt the octets using the algorithm and key from steps 1 and 2.

c. Unless the **decryptor** will implicitly know the type of the encrypted data, the **encryptor** should provide the type for representation.

4. Build the `EncryptedType` structure. An `EncryptedType` structure represents all of the information previously discussed including the type of the encrypted data, encryption algorithm, parameters, key, type of the encrypted data, etc.

a. If the encrypted octet sequence obtained in step 3 is to be stored in the `CipherData` element within the `EncryptedType`, then the encrypted octet sequence is base64 encoded and inserted as the content of a `CipherValue` element.

b. If the encrypted octet sequence is to be stored externally to the `EncryptedType` structure, then store or return the encrypted octet sequence, and represent the URI and transforms (if any) required for the **decryptor** to retrieve the encrypted octet sequence within a `CipherReference` element.

5. Process `EncryptedData`

a. If the type of the encrypted data is a JSON element, then the **encryptor** must be able to return the `EncryptedData` element to the application. The application may use this as a new JSON document or insert it into an another. The **encryptor** should be able to replace the unencrypted 'element' or 'content' with the `EncryptedData` element. When

b. an application requires an JSON element or content to be replaced, it supplies the JSON document context in addition to identifying the element or content to be replaced. The **encryptor** removes the identified element or content and inserts the `EncryptedData` element in its place.

If the type of the encrypted data is not 'element' or element 'content', then the **encryptor** must always return the `EncryptedData` element to the application. The application may use this as a new JSON document or insert it into an another.

`EncryptedType` derived element to be decrypted (see **Error! Reference source not found.**), the **decryptor** must:

1. Process the element to determine the algorithm, parameters and `KeyInfo` element to be used. If some information is omitted, the application is responsible for supply it.
2. Locate the data encryption key according to the `KeyInfo` element. If the data encryption key is encrypted, locate the corresponding key to decrypt it. Or, one might retrieve the data encryption key from a local store using the provided attributes or implicit binding.
3. Decrypt the data contained in the `CipherData` element.

a. If a `CipherValue` child element is present, then the associated text value is retrieved and base64 decoded so as to obtain the encrypted octet sequence.

b. If a `CipherReference` child element is present, the URI and transforms (if any) are used to retrieve the encrypted octet sequence.

c. The encrypted octet sequence is decrypted using the algorithm/parameters and key value already determined from steps 1 and 2.

4. Process decrypted data.

a. The cleartext octet sequence obtained in step 3 is interpreted as UTF-8 encoded character data.

b. The **decryptor** must permit the return of resulting data in a JSON structure with defined encoding. The **decryptor** is not required to perform validation on the serialized JSON.

c. The **decryptor** should support the ability to replace the `EncryptedData` element with the decrypted JSON element or simple content. The **decryptor** is not required to perform validation on the result of this replacement operation. The **application** supplies the JSON document context and identifies the `EncryptedData` element being replaced. If the document into which the replacement is occurring is not UTF-8, the **decryptor** must transcode the UTF-8 encoded characters into the target encoding.

4. SECJSON Implementation

In order to validate the SecJSON specification and usage and in order to make it available for third party developers, an implementation of SecJSON was built using Node.js. Node.js (or simply Node) is an open-source platform for server-side and web applications [20] development entirely based on JavaScript and JSON format, which is an advantage for its adoption throughout this article. Besides the already mentioned advantages, Node.js also has a Node Package Manager (NPM), which is the default package manager for Node.js [20]. This allow that new libraries stay available to developers, making code reutilization easy and efficient on development [21].

4.1. secjson.js

Throughout this section the main Node.js functions developed according to the syntax defined in the previous sections, are presented. The implementation of XML Encryption for Node.js was considered as the starting point for this implementation, and it may be accessed from `https://github.com/auth0/node-xml-encryption`.

4.2. Encryption process

The encryption process is responsible for receiving content and other parameters to encrypt and return a JSON object according to the defined syntax. As required parameters, this function requires content to encrypt, public key, PEM x509 certificate, and optionally set the element to encrypt using a JSON path. When invoked, this operation, sequentially applies the methods needed to encrypt the content provided:

- `findKeyEncryptValue`: if a JSON path is defined, the element will be located in the JSON structure.

- `generate_symmetric_key`: generate a symmetric key to encrypt the user-defined content.

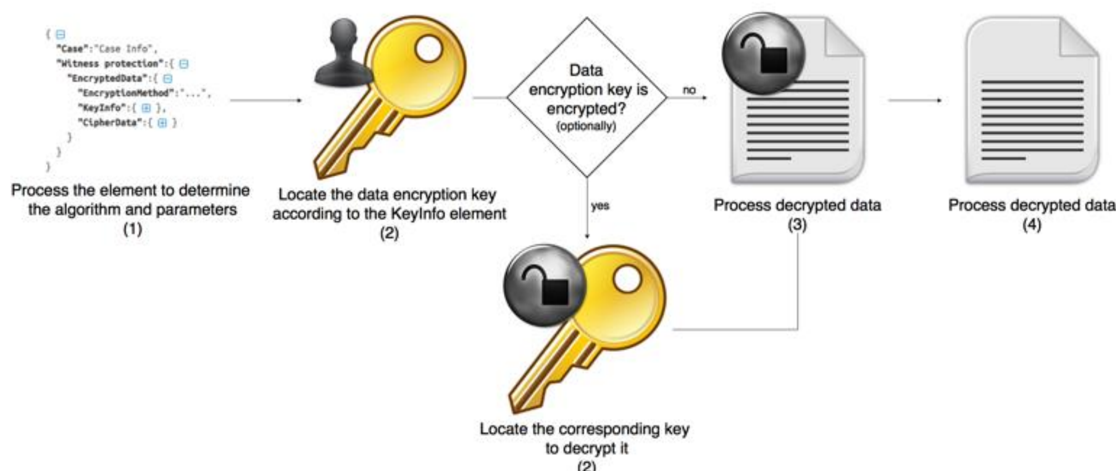

Figure 3. SecJSON decryption process

- `encrypt_content`: encrypt the user-defined content with the key generated in the previous point.
- `encrypt_key`: encrypt the symmetric key used for encryption with public key provided by the user.

The following section of source-code represents a small example on how to use the SecJSON library to encrypt some JSON data (the **JSONDATA** part, should be replaced by the actual JSON data to encrypt). The "encrypt" function receives a set of options to setup the encryption process (namely the encryption key to use) and encrypts the data.

```
var secjson = require('secjson');

var options = {
    rsa_pub: fs.readFileSync(__dirname +
'/test-auth0_rsa.pub'),
    pem: fs.readFileSync(__dirname + '/test-
auth0.pem'),
    encryptionAlgorithm:
'http://tiagomistral.github.io/SecJSON#aes12
8-cbc',
    keyEncryptionAlgorighm:
'http://tiagomistral.github.io/SecJSON#rsa-
oaep-mgf1p'
};

secjson.encrypt('<JSONDATA>', options,
function(err, result) {
    console.log(result);
});
```

4.3. Decryption process

The decryption process is responsible for obtaining the decrypted content. As parameters this function requires a JSON object according to SecJSON syntax and a private key. The methods needed to decrypt the content provided, will then be called, in sequence:

- `findKeyDecryptValue`: if a JSON path is defined, the element will be located in the JSON structure.
- `JSON.parse`: validate JSON object provided.
- `decryptKeyInfo`: Decipher the element content `EncryptedData.KeyInfo.CipherData` with the private key provided, getting the symmetric key used in the encryption process.
- `switch(encryptionAlgorithm)`: Decipher the payload with the symmetric key obtained in the previous point. This process is dependent on the element `EncryptedData.EncryptionMethod`, whose information corresponds to that used cryptographic algorithm (AES 128, AES 256 or TripleDES).

The following section of source-code represents a small example on how to use the SecJSON library to decrypt some previously encrypted JSON data. The "decrypt" function receives a set of options to setup the decryption process (namely the appropriate decryption key to use) and decrypts the data.

```
var decryptOptions = {
  key: fs.readFileSync(__dirname + '/test-
auth0.key')
};

secjson.decrypt(encryptResult,
decryptOptions, function(err, dec) {
```

```
  console.log(dec);
});
```

5. Conclusions

The distribution of services over the Internet has grown in the past years as one of the most interesting trends in software development [22]. A proliferation of web-based APIs has popped up allowing developers to extend their services with the ones developed by third parties. HTTP-based RESTful services have become one of the most relevant ways to implement distributed web-services and JSON has emerged has the data interoperability standard that enables transparent data transfer between different implementation technologies [23].

Data transfer between all of these services, includes critical private information that requires specific protection. Most of the times, the SSL/TLS protocol can be used to provide end-to-end channel encryption however, some specific cases may require more than simply channel encryption. For instance, there are some situations in which the data contained in a JSON document can contain sensitive information that cannot be disclosed to all the possible entities at the same time. This information can have different protection layers, ciphered with multiple keys and using different encryption methods. These are some of the questions that SSL/TLS cannot answer.

Having this into consideration, the authors propose and describe a secure JSON approach, based on previous XML and web services security work, that offers the required requirements that extend the protection used by traditional end-to-end channel encryption approaches. The goal of the presented work is not to act as a replacement for SSL/TLS protocol but rather to complement it while offering an additional security layer to the security of the JSON content transmitted over secure or insecure network connections. The implementation of this JSON security framework consisted on three main parts: the definition of a syntax that allows encryption and decryption of a JSON document, implementation and delivery of a prototype of the defined syntax and validation of implementation. The validation of the implementation concluded that the SecJSON solution is a complementary solution to SSL/TLS, allowing the support of granular security solutions for JSON protection and the development of an additional security layer on top of SSL/TLS. Also, the similarity with other related XML security solutions, makes SecJSON an easy to learn to solution to all the developers that are used to use a similar approach.

One of the requirements of the work presented on this article was the provision of an open and free SecJSON library, that could be used by developer to implement security on their own REST-based web services. This library was implemented as a Node.js packages and released using NPM, which may be accessed from https://www.npmjs.com/ package/ secjson.

The definition and development of SecJSON was a real challenge but limited the time to software optimization. It would be interesting to extend this project in order to perform comparisons between existing alternative security solutions for JSON and the one described here.

6. References

[1] F. Müller and F. Thiesing, “Social networking APIs for companies—An example of using the Facebook API for companies,” in Computational Aspects of Social Networks (CASoN), 2011 International Conference on, 2011, pp. 120–123.

[2] G Denaro, M Pezze, D Tosi, and Daniela Schilling, “Towards self-adaptive service-oriented architectures,” in TAV-WEB ’06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications, 2006, pp. 10--16.

[3] C. Severance, “Discovering javascript object notation,” Computer (Long. Beach. Calif)., vol. 4, no. 45, pp. 6–8, 2012.

[4] D. Crockford, “The application/json media type for javascript object notation (json),” 2006.

[5] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, “Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI,” IEEE Internet Comput., vol. 6, no. 2, p. 86, 2002.

[6] R. R. McCune, “Node. js paradigms and benchmarks,” STRIEGEL, Gr. OS F, vol. 11, 2011.

[7] S. Thomas, SSL & TLS Essentials: Securing the Web, Pap/Cdr. Wiley, 2000.

[8] A. A. A. El-Aziz and A. Kannan, “JSON encryption,” in Computer Communication and Informatics (ICCCI), 2014 International Conference on, 2014, pp. 1–6.

[9] P. Ratnasingam, “The importance of technology trust in web services security,” Inf. Manag. Comput. Secur., vol. 10, no. 5, pp. 255–260, 2002.

[10] K. Maeda, “Performance evaluation of object serialization libraries in XML, JSON and binary formats,” in Digital Information and Communication Technology and it’s Applications (DICTAP), 2012 Second International Conference on, 2012, pp. 177–182.

[11] M. Miller, “Using JavaScript Object Notation (JSON) Web Encryption (JWE) for Protecting JSON Web Key (JWK) Objects,” 2013.

Copyright © 2017, Infonomics Society

[12] E. Stark, M. Hamburg, and D. Boneh, "Symmetric cryptography in javascript," in Computer Security Applications Conference, 2009. ACSAC'09. Annual, 2009, pp. 373–381.

[13] M. Jones, J. Bradley, and N. Sakimura, "Json web token (jwt)," 2015.

[14] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Signature (JWS)," 2015.

[15] M. Jones and J. Hildebrand, "Json web encryption (jwe)," 2015.

[16] M. Jones, "JSON web key (JWK)," 2015.

[17] M. Jones, "JSON Web Algorithms (JWA)," 2015.

[18] A. Nadalin, G. T. AmberPoint, P. D. BEA, H. L. BEA, S. C. CommerceOne, T. D. ContentGuard, G. L. ContentGuard, T. J. P. ContentGuard, S. S. C. Commerce, G. V. Documentum, and others, "Web Services Security," SOAP Messag. Secur. Version, vol. 1, 2002.

[19] T. Imamura, B. Dillaway, E. Simon, and others, "XML encryption syntax and processing," W3C Recomm., vol. 10, 2002.

[20] M. Cantelon, M. Harter, T. J. Holowaychuk, and N. Rajlich, Node. js in Action. Manning, 2014.

[21] S. Tilkov and S. Vinoski, "Node. js: Using JavaScript to build high-performance network programs," IEEE Internet Comput., vol. 14, no. 6, p. 80, 2010.

[22] K. M. Dhara, M. Dharmala, and C. K. Sharma, "A Survey Paper on Service Oriented Architecture Approach and Modern Web Services," 2015.

[23] M. W. Khan and E. Abbasi, "Differentiating Parameters for Selecting Simple Object Access Protocol (SOAP) vs. Representational State Transfer (REST) Based Architecture," J. Adv. Comput. Networks, vol. 3, no. 1, 2015.

Copyright © 2017, Infonomics Society