Secure Javascript Object Notation (SecJSON)

Enabling granular confidentiality and integrity of JSON documents

Tiago Santos, Carlos Serrão
ISCTE – Instituto Universitário de Lisboa
Ed. ISCTE, Av. das Forças Armadas, 1649-026, Lisbon, Portugal
tfpss1@iscte.pt, carlos.serrao@iscte.pt

Abstract— **Currently, web and mobile-based systems exchange information with other services, mostly through APIs that extend the functionality and enable multipart interoperable information exchange. Most of this is accomplished through the usage of RESTful APIs and data exchange that is conducted using JSON over the HTTP or HTTPS protocol. In the case of the exchange requires some specific security requirements, SSL/TLS protocol is used to create a secure authenticated channel between the two communication end-points. This is a scenario where all the content of the channels is encrypted and is useful if the sender and the receptor are the only communicating parties, however this may not be the case. The authors of this paper, present a granular mechanism for selectively offering confidentiality and integrity to JSON documents, through the usage of public-key cryptography, based on the mechanisms that have been used also to provide XML security. The paper presents the proposal of the syntax for the SecJSON mechanism and an implementation that was created to offer developers the possibility to offer this security mechanism into their own services and applications.**

Keywords- Security; Integrity; Confidentiality; API; JSON; HTTPS; SSL/TLS

I. INTRODUCTION

Nowadays, one of the main mechanisms that is used to exchange information 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 [7] across 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. One of its first implementations targeted the communication between Javascript-based scripts and Java-based servers. Although JSON was first developed having into consideration Javascript, it is currently platform and programming language independent. In the last few years there has been a growing usage of this format to serialize and de-serialize information on web services, allowing the 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 [5] and applications. JSON is widely used to support the communication between multiple APIs available on WWW REST services. This way, JSON security issue gains momentum due to the sensitive characteristics of the information that is JSON-encapsulated (JSON payload) and transported between this distributed heterogeneous ecosystem.

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 - SSL/TLS [8]. However, SSL/TLS blindly ciphers all the information that flows on the communication channel, in a similar way. With this it is not possible to cipher the same JSON message conditionally, using different keys or using different protection mechanisms (cryptographic algorithms), which could be required by specific applications [1]. There may exist critical information that needs to be sent or redirected to multiple entities, even if those entities are not the final receptor of such message, there 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 this requirement 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 (Figure 1).

This article intends to present a secure and granular solution for JSON documents. The major contribution of the article can be resumed in the presentation of the syntax and semantics of a mechanism capable of ensuring the granular security of JSON objects and the implementation of the syntax developed in a programming language that allow easy integration into projects of other developers. The paper 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.

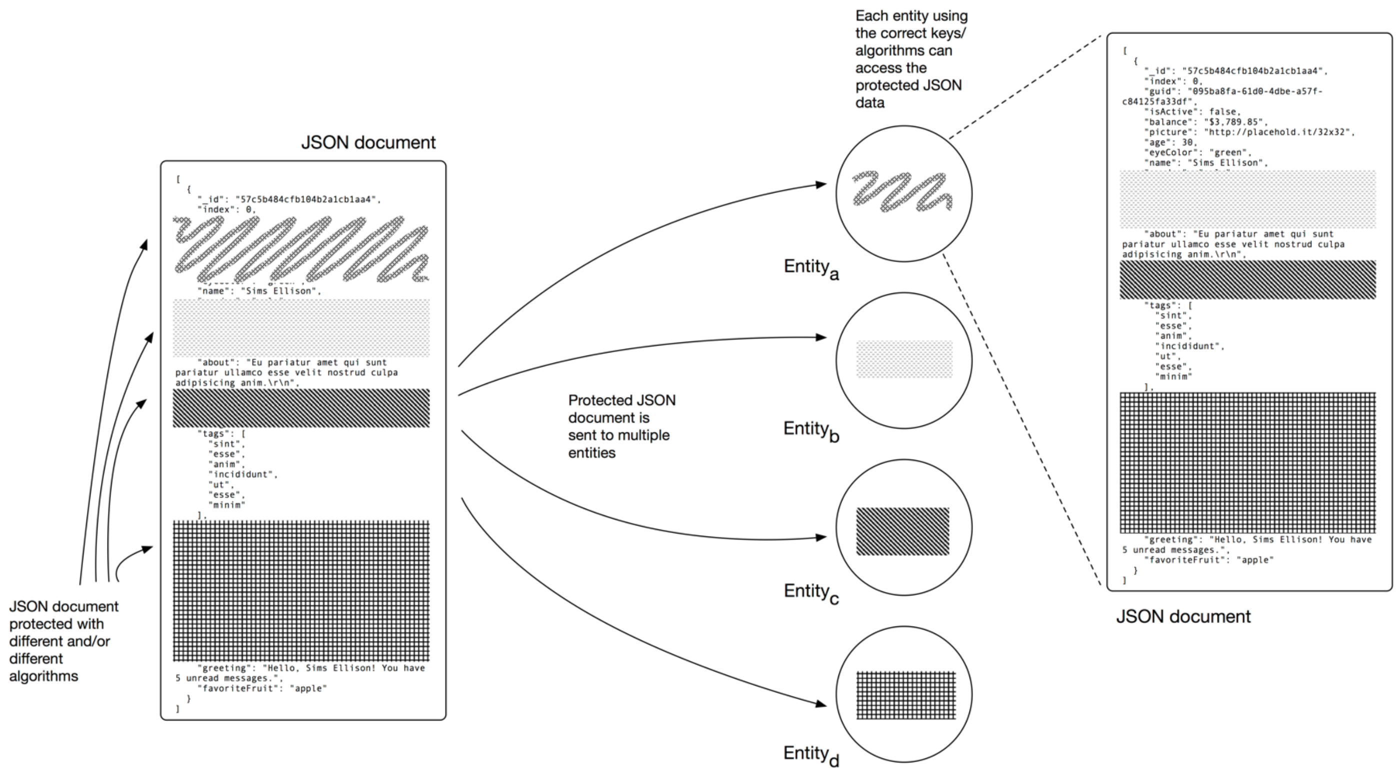

JSON document
JSON document protected with different and/or different algorithms
Protected JSON document is sent to multiple entities
Each entity using the correct keys/ algorithms can access the protected JSON data
Entity_a
Entity_b
Entity_c
Entity_d
JSON document

Figure 1. An overview on how SecJSON works

II. JSON-BASED WEB SERVICES

Most business transactions currently depend on the existence of Web Services. This is the reason why it has become one of the most important areas of the IT industry [6]. The security inherent in this type of transaction 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 raise security concerns. How can organizations ensure that their or the users information reaches the final destination safely, preserving confidentiality and integrity, whenever sensitive information is routed through the WWW [6]. 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. 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 [4]. 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.

III. SECURE JAVASCRIPT OBJECT NOTATION (SECJSON)

Considering the different aspects of 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 protect JSON, 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 a the description of the approach that was followed throughout its development. The syntax of SecJSON is also presented.

A. 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);
- 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;
- 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 useby third parties.

Having into consideration the requirements that were identified, the SecJSON specification and development is presented in the following sections.

B. SecJSON overview

The proposed Secure JSON consists in a set of rules and specifications for encrypting information and rep- resent their results in JSON format. Data to be protected can be 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 (Imamura et al., 2002).

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
  }]
}
```

Considering the above JSON object, where it is required to offer witnesses information protection. In an initial stage it should be identified where is the in- formation 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 ...)
  }
}
```

C. SecJSON proposed syntax

This section offers a detailed description of the syntax and features for SecJSON. The syntax is defined using JSON-Schema. 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 latter element types are very similar with respect to their content models, a syntactical distinction is useful for processing.

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 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 syntax. Not only does its `CipherData` child contain the encrypted data, but it's 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:

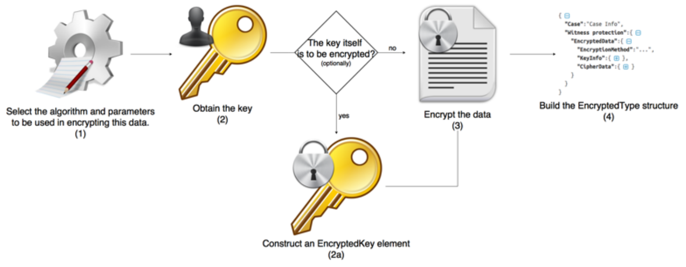
- The `EncryptedData` or `EncryptedKey` element specify the associated keying material via a child of KeyInfo element.
- The keying material can be determined by the recipient by application context and thus need not be explicitly mentioned in the transmitted JSON document.

`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.

D. 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, as the definition of SecJSON elements, the rules are based on the same rules used by XML Encryption standard [3].


Select the algorithm and parameters to be used in encrypting this data. (1)
Obtain the key (2)
The key itself is to be encrypted? (optionally)
no
yes
Encrypt the data (3)
Build the EncryptedType structure (4)
Construct an EncryptedKey element (2a)

```
{
  "Case":"Case Info",
  "Witness protection":{
    "EncryptedData":{
      "EncryptionMethod":"...",
      "KeyInfo":{ },
      "CipherData":{ }
    }
  }
}
```

Figure 2. SecJSON encryption process

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.

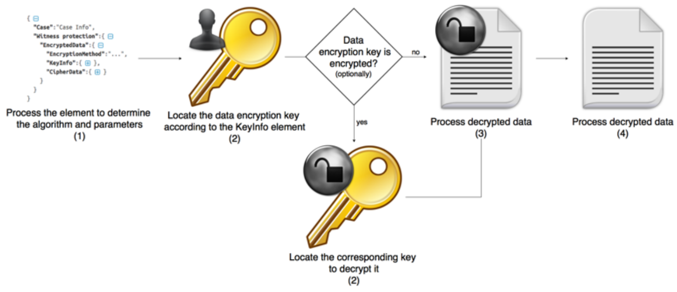
For each data item to be encrypted (Figure 2) as an element derived from `EncryptedType`, the **encryptor** must:

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 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.

    b. 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.



```
{
  "Case":"Case Info",
  "Witness protection":{
    "EncryptedData":{
      "EncryptionMethod":"...",
      "KeyInfo":{ },
      "CipherData":{ }
    }
  }
}
```

Process the element to determine the algorithm and parameters (1)
Locate the data encryption key according to the KeyInfo element (2)
Data encryption key is encrypted? (optionally)
no
yes
Process decrypted data (3)
Process decrypted data (4)
Locate the corresponding key to decrypt it (2)

Figure 3. SecJSON decryption process

`EncryptedType` derived element to be decrypted (Figure 3), 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.

IV. 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 performed using Node.js. Node.js (or simply Node) is an open-source platform for server-side and web applications [2] 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 [2]. This allow that new libraries stay available to developers, making code reutilization easy and efficient on development [9].

A. 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`.

B. 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.

- `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.

C. *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).

V. Conclusions

The distribution of services over the Internet has grown in the past years as one of the most interesting trends in software development. 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.

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 security work, that offers the required requirements that extend the protection used by tradicional end-to-end channel encryption approaches. The implementation of this project consists of three main points: 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 valid alternative to SSL/TLS, displaying superior performance in several specific scenarios. The ultimate goal of this work was achieved with the release of the prototype in the NPM, which may be accessed from `https://www.npmjs.com/package/secjson`.

The definition and development of SecJSON were a real challenge but limited the time to software optimization. It would be interesting to extend this project in order to achieve superior performance fostering greater adoption market.

References

[1] Abd El-Aziz, A. and Kannan, A. (2014). Json encryption. In Computer Communication and Informatics (ICCCI), 2014 International Conference on, pages 1–6. IEEE.

[2] Cantelon, M., Harter, M., Holowaychuk, T., and Rajlich, N. (2014). Node. js in Action. Manning.

[3] Imamura, T., Dillaway, B., and Simon, E. (2002). Xml encryption syntax and processing. W3C recommendation, 12:2002.

[4] Maeda, K. (2012). 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 Inter- national Conference on, pages 177–182. IEEE.

[5] McCune, R. R. (2011). Node. js paradigms and bench- marks. STRIEGEL, GRAD OS F, 11.

[6] Ratnasingam, P. (2002). The importance of technology trust in web services security. Information Management & Computer Security, 10(5):255–260.

[7] Severance, C. (2012). Discovering javascript object nota- tion. Computer, (4):6–8.

[8] Stephen, T. (2000). Ssl and tls essentials–securing the web.

[9] Tilkov, S. and Vinoski, S. (2010). Node. js: Using javascript to build high-performance network programs. IEEE Internet Computing, 14(6):80.