RWTH Aachen

Department of Computer Science
Technical Report

The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud

Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2016-06

RWTH Aachen · Department of Computer Science · July 2016

The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud

Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle

Communication and Distributed Systems
RWTH Aachen University, Germany
Email: {henze, hummen, matzutt, wehrle}@comsys.rwth-aachen.de

Abstract. The increasing deployment of sensor networks, ranging from home networks to industrial automation, leads to a similarly growing demand for storing and processing the collected sensor data. To satisfy this demand, the most promising approach to date is the utilization of the dynamically scalable, on-demand resources made available via the cloud computing paradigm. However, prevalent security and privacy concerns are a huge obstacle for the outsourcing of sensor data to the cloud. Hence, sensor data needs to be secured properly before it can be outsourced to the cloud.
When securing the outsourcing of sensor data to the cloud, one important challenge lies in the representation of sensor data and the choice of security measures applied to it. In this paper, we present the SensorCloud protocol, which enables the representation of sensor data and actuator commands using JSON as well as the encoding of the object security mechanisms applied to a given sensor data item. Notably, we solely utilize mechanisms that have been or currently are in the process of being standardized at the IETF to aid the wide applicability of our approach.

1 Introduction

Recent advances in ubiquitous computing and wireless sensor networks continue to obliterate the boundaries between the physical and the digital world [1, 7, 10, 14]. Sensor networks can be utilized in a large variety of deployments, ranging from personal homes over offices and cars to industrial facilities and public areas [4,12]. To cope with the resulting increase in demands for storing and processing sensor data, cloud computing elastically provides the necessary computation and storage resources [12]. Cloud computing allows the collection, processing, and storage of sensor data at large scales and as well enables the world-wide sharing of said data [5, 11]. In the context of the SensorCloud project [4], we consider a scenario in which operators of sensor networks (i.e., private users, companies, or public institutions) connect their sensor networks to the cloud [12], where collected sensor data is processed by cloud services selected by the sensor network operator [4, 11, 12, 14]. Besides the remarkable advantages of cloud computing, it is important to note that sensor data often contains sensitive information. Hence, when transferring this sensitive data to entities outside of trusted sensor networks, it might, e.g., be unintentionally forwarded to third parties or used for non-authorized purposes [6, 8, 13, 20]. Furthermore, data stored and processed in the cloud might be subject to access by the cloud provider or governmental agencies [9]. Thus, one major challenge when interconnecting sensor networks with the cloud is to account for the aforementioned security and privacy concerns.

As part of the efforts of the SensorCloud project, we developed a trust point-based security architecture for outsourcing sensor data to the cloud and the

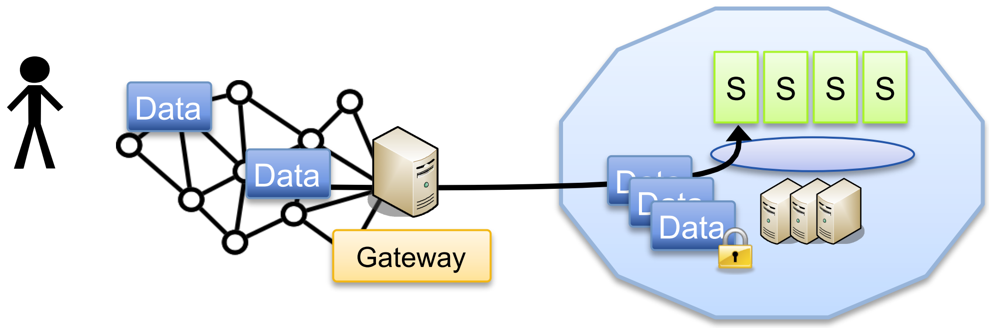

Figure 1. In the SensorCloud scenario, data flows from sensor networks through a dedicated gateway to the cloud. There, the data is stored securely and can only be accessed by authorized services.

SensorCloud security library [4,5,11,12,14]. One important challenge in securely outsourcing sensor data to the cloud lies in the representation of sensor data and the corresponding security measures taken to protect the sensor data. In this paper, we report on the SensorCloud protocol, which has jointly been developed within the SensorCloud project to represent sensor data and actuator commands using JSON and subsequently secure this data using object security mechanisms. To this end, we rely on mechanisms that have been or currently are in the process of being standardized at the IETF and provide a best practice on how to utilize and combine them in an actual system.

The remainder of this document is structured as follows. In Section 2, we present the SensorCloud scenario in more detail and provide references to more detailed descriptions of the overall security architecture. Section 3 defines the JSON-based representation of sensor data and actuator commands in Sensor-Cloud. In Section 4, we describe the security extensions to this representation to realize the secure outsourcing of sensor data to the cloud. We conclude this paper in Section 5.

2 SensorCloud Scenario

In SensorCloud, we consider a scenario where each sensor network (with an arbitrary number of sensor nodes) is connected to the cloud via a dedicated gateway as depicted in Figure 1. Our goal is to store data securely in the cloud such that it can only be processed by authorized cloud services. To this end, the gateway encrypts sensitive sensor readings using a symmetric cipher before uploading it to the cloud. The encryption process is influenced by a user-configurable access control list containing services that are authorized to (partially) obtain and process the user's sensor data. Now, only entities in possession of the symmetric key used for encrypting a piece of sensor data (referred to as a data item) have access to this specific data item. Hence, to grant a cloud service access to a given data item the gateway has to provide this cloud service with the corresponding key. To this end, the gateway asymmetrically encrypts the corresponding symmetric key with the public key of the cloud service that should gain access to the sensor data and forwards the resulting encrypted key to the respective cloud service. More details, especially with respect to the design and implementation of the underlying security architecture as well as our choice of cryptographic primitives, can be found in the corresponding publications [11, 12, 14].

Notably, sensor data originating from a single (possibly virtual) sensor node can contain multiple sensor readings from different sensors. For example, one data item measured by a meteorological sensor might consist of multiple single sensor readings such as humidity and temperature. Hence, SensorCloud supports the transmission of multiple sensor readings of one sensor node in a single message [5,12]. As a cloud service might only be granted access to parts of the sensor readings, SensorCloud supports the encryption of individual parts of sensor data, thus realizing fine-grained access control.

The processing of sensor data by a cloud service requires the verification of the integrity of received data, the decryption of the symmetric key, and finally the decryption of the actual sensor data. These operations have been implemented in the SensorCloud security library, which allows the transparent decryption of sensor data and the verification of its integrity by a cloud service [5]. It is available as open source software under the MIT license[1].

3 SensorCloud JSON Representation

The remainder of this document defines the JSON-based messages used by the SensorCloud protocol to encode data items as well as additional configuration messages. More precisely, the JSON-based message layout defined by this document is used for both internal communication within the gateway and cloud, respectively, and for communication between those entities.

3.1 Message Definition

The JSON message header MUST be included for transmissions between the gateway and the cloud. It MAY be omitted for internal data exchanges between components.

```
{
    "ver":"<number>", /* specification version */
    "seq":"<number>", /* for message acknowledgments */
    "pl":"[<messages>]" /* actual message payload */
}
```

The field `ver` contains the version number of the protocol, which defines the structure of the remainder of the message. Hence, whenever the message structure or encapsulated payloads are subject to changes, the version number MUST be increased. The receiving peer MUST support the indicated version number. Otherwise, it SHOULD notify the sending peer that the message was dropped. The version number is a positive integer and is currently defined as `1`.

The sequence number `seq` is currently unused and MUST therefore be set to `0`. However, it enables end-to-end acknowledgments at the application layer. The corresponding retransmission mechanism will need to be developed in the future. If used, `seq` is a positive integer that MUST be increased by one for each JSON

[1] `https://code.comsys.rwth-aachen.de/redmine/projects/scslib`

message header sent. A potential wrap-around of the sequence number is to be expected and implementations MUST interpret this case as an incrementation of the sequence number.

The messages to be transmitted using one header are stored within an array in the field `pl`. Section 3.2 defines a list of message types supported by SensorCloud. Additional message types can be defined as necessary. Messages MUST contain the field `typ` to indicate their message type for message processing purposes. Moreover, they MUST include the `gw` field to indicate the gateway device that participates in the communication. Among other aspects, this information is required for applying object security measures. Note that multiple messages of different types can be batched using one message header in order to reduce the overall communication overhead.

3.2 Defined Message Types

Each message indicates its message type via the field `typ`, which MUST be included in any message. The message type is a positive integer value and can currently take one of the values described in Table 1.

Value	Semantics
1	Sensor Data Message
2	Sensor Data Request
3	Configuration Message
4	Actuator Command
5	Actuator Response
400	Data Key Upload
401	Data Key Download
402	Public Key Download
403	Public Key Response

Table 1. Message types defined for the SensorCloud protocol.

The remainder of this section defines the structure of messages of the respective types in detail. Section 3.3 describes messages carrying sensor data while Section 3.4 defines sensor data queries to be sent by cloud services. Section 3.5 defines configuration messages sent to describe the layout of sensor data originating from single sensor nodes. Afterwards, Section 3.6 and Section 3.7 describe commands to be sent to sensor nodes with actuation capabilities and the format of responses to such commands, respectively. We finally describe messages related to the management of cryptographic keys as a part of the security extensions in Section 4.3.

3.3 Transmitting Sensor Data

For sensor data, the JSON representation is strongly based on the JSON definitions of the Media Types for Sensor Markup Language (SenML) [15]. SenML is currently being standardized at the IETF. Only a short example is discussed here to show how SenML integrates into the sensor data representation of SensorCloud. For specifics about the SenML representation, reading Section 6 of the corresponding SenML draft [15] is strongly encouraged.

```
{
    "typ":"1", /* type of the message payload */
    "gw":"<string>", /* unique gateway ID */
    "bn":"<string>", /* sensor device ID */
    "bt":"<number>", /* base time of sensor readings */
    "e":[{
        "n":"<string>", /* sensor ID */
        "t":"<number>", /* optional time value */
        "sv":"<string>" /* sensed value as string */
    }]
}
```

Each message encodes one data item, i.e., one piece of sensor data as described in Section 2. Each data item starts with the field `typ`, which is always set to `1`, indicating that the message contains sensor data to be uploaded. The fields `src` and `bn` contain the unique identifier of the processing gateway and the identifier of the originating sensor device, respectively. The sensor device identifier `bn` thereby MUST be unique on a per-gateway basis. Data items also carry a timestamp `bt`, a positive integer denoting the time (in milliseconds since UNIX epoch) at which the oldest sensor reading within the data item was measured.

Sensor data from a single (virtual) sensor device can consist of readings from multiple sensors of the same device (e.g., humidity and temperature), a series of sensor readings from the same sensor, or a combination of the two. Each sensor reading is encoded as a JSON object in the array `e`. The name field `n` thereby allows to identify the specific sensor, whereas `t` allows to identify the time offset (in milliseconds) of a sensor reading relative to the base time `bt`. Objects in the array `e` MUST be ordered according to an ascending alphanumeric order with respect to the field `n`. If `n` is equal for multiple sensor readings, these elements MUST be ordered in ascending order of the time value `t`. The array `e` MUST NOT contain two or more elements with the same combination of `n` and `t` values. Hence, the granularity of `t` MUST be sufficient to guarantee this property. In SensorCloud, the granularity of `t` is milliseconds. Moreover, the individual JSON identifiers MUST appear in the order shown above. This order is required to preserve the payload structure to enable cryptographically verifying the data item's integrity later on.

The actual sensed data is stored in the field `sv` (string value). SenML already defines a number of primitive data types and `sv` is only one of them. Other data types such as `bv` (boolean value) may be used as well, but are currently not supported by SensorCloud. However, using different data types can aid casting of the sensed value in typed programming languages.

3.4 Requesting Sensor Data

Third-party services can query the cloud for sets of data items to process them. The cloud, holding a replica of the data owner's access control list (cf. Section 2), then returns the data items that are specified in the query and accessible by the service (to unburden the service from attempting to decrypt unaccessible data items). Data item queries have the following structure, which is based on the structure of data items:

```
{
    "typ":"2", /* payload is a sensor data request */
    "gw":"<string>", /* unique gateway ID */
    "srv":"<string>", /* unique service ID */
    "lim":"<number>", /* response length limit */
    "off":"<number>", /* offset in specified sensor data stream */
    "bt":["<number>"], /* base time of sensor readings */
    "bn":["<string>"], /* sensor device IDs */
    "e":[{
        "n":"<string>", /* sensor IDs */
    }]
}
```

Sensor data requests always have a message type `typ` of `2`. The fields `srv` and `gw` specify the service requesting sensor data and the gateway responsible for the queried sensor data, respectively. The fields `lim` (limit) and `off` (offset) MAY be used by the requesting service to further control the query. If `lim` is specified, the cloud MUST NOT return more than the specified number of data items in the response. If `lim` is not given, the cloud SHOULD return the whole response, i.e., all currently available data items matching the query. Furthermore, the cloud MUST ignore the first `off`-many data items it would include in the response. If `off` is not given, its default value is `0`.

The remaining fields define the query itself. The base time array `bt` specifies a time range for the creation times of data items (as given by the field `bt`, c.f. Section 3.3) to be included in the response. The maximum length of `bt` is two; in this case, the first element specifies the lower time bound and the second element specifies the upper time bound in milliseconds elapsed since UNIX epoch, respectively. Note that both bounds are including, i.e., data items to be included in the response have a value of `bt` that is larger than or equal to the lower bound and smaller than or equal to the upper bound, respectively. If only one element is given, it is treated as the lower time bound for the data item creation time. If `bt` is empty or not given, all data items matching by other criteria are to be returned. The array `bn` specifies a set of identifiers of sensor nodes whose sensor readings shall be processed by the querying cloud service. For instance, the affected sensor nodes can be derived from the cloud's copy of the data owner's access control list. However, the exact specification of how services obtain the required sensor node identifiers is out of the scope of the SensorCloud protocol. The array `e` contains JSON objects each containing a sensor identifier. The cloud MUST return only those data items that have been sensed by the sensors specified in the array `e`. In summary, when receiving a query the cloud MUST return exactly those data items that (a) have been created within the timespan given by `bt`, (b) originate from a sensor node given in `bn`, and (c) contain measurements by sensors of a type specified in the array `e`.

3.5 Configuration Definition

SensorCloud uses dedicated configuration messages to define the layout of sensor data readings emitted by a specific sensor device. This way, sensor networks consisting of devices with heterogeneous capabilities can be accounted for. The structure of configuration messages is as follows:

```
{
    "typ":"3", /* payload is a configuration message */
    "gw":"<string>", /* unique gateway ID */
    "bn":"<string>", /* sensor device ID */
    "js":"<string>", /* JSON schema */
}
```

Configuration messages always use the message type `3` in the field `typ`. The fields `gw` and `bn` contain the unique identifier of the processing gateway and the identifier of the addressed sensor device, respectively. The field `js` contains the JSON schema specifying the structure of sensor data read by the sensor device that is uniquely identified by the combination of `gw` and `bn`.

3.6 Actuator Command Definition

Sensor network devices can also be equipped with actuators, allowing authorized entities, e.g., cloud services authorized by the sensor network owner, to externally trigger certain actions based on current sensor measurements. In SensorCloud, actuator messages are used to forward actuator commands. Their structure is based on the structure of sensor data payloads and looks as follows:

```
{
    "typ":"4", /* type of the message payload */
    "gw":"<string>", /* unique gateway ID */
    "srv":"<string>", /* unique service ID */
    "bn":"<string>", /* actuator device ID */
    "seq":"<number>", /* Sequence number (optional) */
    "fn":"<string>", /* Function name (optional) */
    "e":[{
        "n":"<string>", /* Parameter key */
        "sv":"<string>" /* Parameter value */
    }]
}
```

Actuator command messages use a message type `typ` of `4`. The field `gw` specifies the destination gateway for the actuator command. The destination gateway decides whether or not to forward an actuator command based on internal access control lists containing those cloud services that have been authorized by the data owner to access fractions of her sensor data. The field `srv` contains a unique identifier of the triggering cloud service to allow the gateway to match the actuator command to an entry in the access control list. The actuator device to execute the command can be uniquely identified by the destination gateway by considering the field `bn`. Optionally, an actuator command can also carry a sequence number `seq`, which is used by cloud services if a response to the command is expected, e.g., an acknowledgment. In this case, the cloud service MUST ensure that `seq` is chosen in a way that the mapping between actuator commands sent and responses received is unambiguous. The optional field `fn` MAY be used to specify the name or identifier of a function to be called at the gateway for RPC-like interactions [3]. The array `e` contains a set of parameters in the form of key-value pairs (`n` as key, `sv` as value) that shall be passed to the called function. If `fn` is not included, the default behavior is to set the parameters given by the fields `n` in `e` at the actuator `bn` to the respective values `sv`.

Each actuator command message may only contain a single actuator command. If multiple actuator commands are to be initiated, multiple messages of type `4` can be batched using a single message header, which was defined in Section 3.1.

3.7 Actuator Response Definition

Actuator commands in SensorCloud can trigger actuator responses, e.g., acknowledging a successful execution of the triggered action. In such a case, an actuator response is sent to the originating cloud service, thereby enabling RPC-like interactions [3]. Note that the sequence number `seq` MUST be set in the corresponding actuator command in order to enable the receiving service to match the response to the corresponding request. The structure of an actuator response message is as follows, in analogy to the structure of an actuator command message:

```
{
    "typ":"5", /* type of the message payload */
    "gw":"<string>", /* unique gateway ID */
    "srv":"<string>", /* unique service ID */
    "bn":"<string>", /* actuator device ID */
    "seq":"<string>", /* Sequence number */
    "fn":"<string>", /* Function name (optional) */
    "e":[{
        "n":"<string>", /* name */
        "sv":"<string>" /* value as string */
    }]
}
```

To denote an actuator response message, its message type `typ` has the fixed value of `5`. The fields `gw` and `srv` denote the unique identifiers of the gateway and cloud service involved in the actuator command, respectively. These fields are copied from the corresponding actuator command message. Additionally, the fields `bn`, `seq`, and `fn` are copied from the actuator command message to enable the cloud service receiving the response to map it to any pending state it holds for the actuator command. If `fn` is not included in the original actuator command message, it is also not included in the response and the service assumes that the actuator responds to only setting the parameter values specified in the actuator command message. The array `e` contains a (possibly empty) set of return values from the previously called function. For instance, this array can contain an error message (e.g., `n` is set to `"err"` and `sv` contains an error code or error message).

Each actuator command response may only refer to a single actuator command. Analogously to actuator command messages, multiple actuator command responses can be batched using a single message header as defined in Section 3.1.

4 Security Extensions

SensorCloud bases the protection of sensitive sensor readings on specifications of the JOSE WG at the IETF. More precisely, JSON Web Encryption (JWE) [17] is employed to encrypt sensitive sensor information, whereas integrity protection and authentication for the complete data item is provided via JSON Web Signature (JWS) [16]. For specifics, reading the respective specifications [16, 17] is strongly encouraged. In the following, we first describe the integration of JWE into the SensorCloud protocol in Section 4.1, then Section 4.2 documents how JWS is used in this protocol, and finally Section 4.3 describes the key management in SensorCloud.

4.1 Encryption

If values are to be encrypted, JWE extends the data item with information about the ciphersuite used as specified by the JWE JSON Serialization. Furthermore, any binary data occurring (e.g., ciphertexts or initialization vectors) are encoded using the `base64url` encoding. Four values are represented in a JWE: the header, initialization vector, ciphertext, and authentication tag. In SensorCloud, JWE-encrypted JSON structures are encapsulated in a JSON array `ev` (encrypted value), replacing the field `sv` of an unencrypted value:

```
"ev":[{
    "unprotected":{
        "alg":"dir",
        "enc":"AESGCM256",
        "kid":"<string>",
        "typ":"<string>"
    },
    "iv":"<initialization vector (base64url-encoded)>",
    "ciphertext":"<ciphertext (base64url-encoded)>",
    "tag":"<authentication tag (base64url-encoded)>"
}]
```

The JWE header, stored in the field `unprotected`, contains the fields `alg`, `enc`, `kid`, and `typ`. This field is not covered by the authentication tag and therefore it does not need to be `base64url`-encoded. For SensorCloud, we fix the value of the `alg` (algorithm) field to `dir` (direct), indicating that the symmetric key used to decrypt the ciphertext is not encrypted asymmetrically itself, i.e., the given keying material is passed directly to the ciphersuite given by the field `enc`. The ciphersuite is fixed as `AESGCM256`, i.e., AES with keys of length 256 bit using the Galois Counter Mode (GCM). The field `kid` (key identifier) is used to store the identifier of the key used for encryption, i.e., its SHA-1 hash value. The field `typ` encodes the data type of the encrypted value, e.g., `sv` in SensorCloud. The initialization vector used during the symmetric encryption is stored in the field `iv`, whereas `tag` holds the authentication tag, which is optionally used depending on the AES mode of operation, e.g., GCM. Finally, `ciphertext` contains the encrypted value itself. For example, if one sensor reading is to be encrypted and one is not, this gives the following representation:

```
{
    "typ":"1",
    "gw":"<string>",
    "bn":"<string>",
    "bt":"<number>",
    "e":[
    {
        "n":"<string>",
        "ev":[
        {
            "unprotected":
            {
                "alg":"dir",
                "enc":"AESGCM256",
                "kid":"<string>",
                "typ":"sv"
            },
            "iv":"<string (base64url-encoded)>",
            "ciphertext":"<string (base64url-encoded)>",
            "tag":"<string (base64url-encoded)>"
        }]
    },
    {
        "n":"<string>",
        "sv":"<string>"
    }]
}
```

After decryption, the field identifier `ev` is replaced with the `typ` given in the `unprotected` header, i.e., `sv` in SensorCloud and the field's content is replaced with the decrypted value. Then, the data item can be processed normally.

Advanced Encryption Schemes. SensorCloud allows for two more advanced schemes for the encryption of sensor data: encrypting the whole measurement array of one data item and packing the encryptions of multiple values into one JWE object.

The definition of `ev` also allows to encrypt the array `e` in the sensor data payload in its entirety instead of on a per-sensor-reading basis. In this case, the plaintext is composed of a canonical serialization of the array `e` and the JWE object `ev` replaces the array `e`. This reduces the encryption overhead in comparison to the case where each sensor value is encrypted separately. However, the trade-off is that information about the individual sensor readings, e.g., the sensor identifier `n`, is encrypted as well, hence selectively processing only subsets of the sensed data is not possible anymore without decrypting the whole array. Thus, the encryption of the entire `e` array is discouraged.

Additionally, the use of `ev` is not restricted to only encrypting single sensor readings. Instead, the definition also allows for multiple fields to be encrypted and encapsulated in a single `ev` array if they are at the same JSON hierarchy level.

In SensorCloud, the encryption of non-sensor-value fields (e.g. `gw`, `bt`) within a data item is discouraged because they contain important information necessary for indexing of sensor data in the cloud. However, configuration messages may contain fields more suited for encryption. To further clarify the use of `ev` with multiple sensitive fields at the same hierarchy level, assume that a configuration message is extended with an additional field `ref`:

```
{
    "typ":"3",
    "gw":"<string>",
    "bn":"<string>",
    "ref":"<string>", /* reference ID */
    "js":"<string>",  /* JSON schema */
}
```

Further assume that both fields `ref` and `js` contain sensitive data and must therefore be encrypted. This yields the following encrypted representation, where both encrypted values are packed into the same array `ev`:

```
{
    "typ":"3",
    "gw":"<string>",
    "bn":"<string>",
    "ev":[
    {
        "unprotected":
        {
            "alg":"dir",
            "enc":"AESGCM256",
            "kid":"<string>",
            "typ":"ref"
        },
        "iv":"<string (base64url-encoded)>",
        "ciphertext":"<string (base64url-encoded)>",
        "tag":"<string (base64url-encoded)>"
    },
    {
        "unprotected":
        {
            "alg":"dir",
            "enc":"AESGCM256",
            "kid":"<string>",
            "typ":"js"
        },
        "iv":"<string (base64url-encoded)>",
        "ciphertext":"<string (base64url-encoded)>",
        "tag":"<string (base64url-encoded)>"
    }]
}
```

In addition to the case mentioned above, a message can also contain multiple `ev` arrays as long as they do not fall into the same scope, e.g., they are not on the same hierarchy level or a certain value in multiple objects in an array shall be encrypted. Each `ev` array thereby contains all encrypted fields within the same scope. To illustrate this, assume that sensor data messages are extended with two new fields `bp` and `bq`, both of which shall be encrypted:

```
{
    "typ":"1",
    "gw":"<string>",
    "bn":"<string>",
    "bt":"<number>",
    "bp":"<string>",
    "bq":"<string>",
    "e":[
    {
        "n":"<string>",
        "sv":"<string>"
    },
    {
        "n":"<string>",
        "sv":"<string>"
    }]
}
```

In this case, the corresponding encrypted message looks as follows (each value `sv` is encrypted as well):

```
{
    "typ":"1",
    "gw":"<string>",
    "bn":"<string>",
    "bt":"<number>",
    "ev":[{... /* encryption of bp */},{... /* encr. of bq */}],
    "e":[
    {
        "n":"<string>"
        "ev":[{... /* encr. of sv */}]
    },
    {
        "n":"<string>"
        "ev":[{... /* encr. of sv */}]
    }]
}
```

Note that this representation contains three `ev` fields: one at the first hierarchy level that contains the encrypted values of `bp` and `bq` as well as another two that encrypt the individual sensor readings, respectively.

4.2 Signing

In order to protect message integrity, messages can be signed digitally by their respective senders. How to digitally sign JSON objects such as SensorCloud messages is specified by JSON Web Signatures (JWS) [16], using the `base64url` encoding for any binary values that shall be integrity-protected and the signature itself. By signing a message, the message is extended with a field `sig` containing a JWS header and the signature:

```
"sig":{
    "signatures":[{
        "header":{"alg":"ES256"},
        "signature":"<signature contents (base64url-encoded)>"
    }]
}
```

To provide a sufficient security level until at least the year 2030 [2], SensorCloud signs messages using ECDSA over the elliptic curve P-256 and SHA-256 as the corresponding hash function. This is indicated by the `header` parameter `alg`, which is set to `"ES256"`. Due to the SensorCloud limitation to a single signature per message, we limit the length of the signatures array to one element. The key used for signing can be derived from the gateway identifier `gw` given in the message to be signed. If no `gw` field exists, the `header` MUST additionally include a field `kid` containing the gateway identifier.

By signing each message individually, the receiver of a message is able to verify each message individually after transmission. Furthermore, messages are always signed as a whole. To ensure consistency for signing a message, an empty object `sig` is appended to the message prior to signing. Additionally, the message is canonicalized following the recommendations of Canonical JSON [19]. Hence, a message may look as follows before the signature algorithm is applied:

```
{
    "typ":"1",
    "gw":"<string>",
    "bn":"<string>",
    "bt":"<number>",
    "e":[
    {
        "n":"<string>",
        "ev":[
        {
            "unprotected":
            {
                "alg":"dir",
                "enc":"AESGCM256",
                "kid":"<string>",
                "typ":"<string>"
            },
            "iv":"<string (base64url-encoded)>",
            "ciphertext":"<string (base64url-encoded)>",
            "tag":"<string (base64url-encoded)>"
        }]
    }],
    "sig":{}
}
```

Then, the signature is computed over the `base64url`-encoded SHA-256 hash value of the message. Finally, the field `sig` is filled by including the JWS header and the signature value as the single element of the `signatures` array:

```
{
    "typ":"1",
    "gw":"<string>",
    "bn":"<string>",
    "bt":"<number>",
    "e":[
    {
        "n":"<string>",
        "ev":[
        {
            "unprotected":
            {
                "alg":"dir",
                "enc":"AESGCM256",
                "kid":"<string>",
                "typ":"<string>"
            },
            "iv":"<string (base64url-encoded)>",
            "ciphertext":"<string (base64url-encoded)>",
            "tag":"<string (base64url-encoded)>"
        }]
    }],
    "sig":{
        "signatures":[{
            "header":{"alg":"ES256"},
            "signature":"<string (base64url-encoded)>"
        }]
    }
}
```

As a result of this approach, multiple sensor readings SHOULD be split into several messages if they are not to be used or stored together. Otherwise, all sensor readings that previously have been retrieved from the same message have to be collected again to verify a message.

4.3 Key Management

In this section, we describe the key management within SensorCloud. First, we define the messages sent for uploading and downloading data keys, respectively.

In SensorCloud, a data key refers to a symmetric cryptographic key used to encrypt a series of data items. Then, we argue why the SensorCloud protocol does not define a dedicated message to upload public keys of participating entities, i.e., gateways and cloud services. Finally, we define the message to be sent when a participating entity has to download the public key of another entity.

Data Key Upload. Data keys are symmetric keys used to encrypt and decrypt data items. In SensorCloud, gateways periodically exchange the data keys used for different streams of sensor data. In order to enable cloud services to process the uploaded sensor data, gateways need to share data keys with those cloud services that are authorized to access this sensor data. Therefore, for each service authorized to read some sensor data a gateway uploads copies of the needed data keys, encrypted with the respective service's public key, to the cloud. The structure of messages for the upload of a data key is as follows:

```
{
    "typ":"400", /* message type */
    "gw":"<string>", /* gateway ID */
    "srv":"<string>", /* service ID */
    "bt":["<number>", "<number>"], /* key validity timerange */
    "bn":"<string>", /* sensor node ID */
    "e":[{
        "n":"<string>", /* sensor ID */
        "kid":"<string>", /* key ID */
        "k":"<string (base64url-encoded)>" /* key material */
    ]}
}
```

Upload messages of data keys always have a message type `typ` of `400`. The fields `gw` and `srv` hold the unique identifiers of the gateway uploading the encrypted data key and the cloud service that is able to obtain the key, respectively. The array `bt` specifies a validity timespan for the data key and MUST be of length two, where the first element is the lower bound of the key's validity and the second element is the upper bound, respectively. Both times are given as milliseconds since UNIX epoch and the timespan includes the upper and lower bounds. The gateway SHOULD NOT encrypt data items using a data key that is already expired. The field `bn` contains the identifier of a sensor node for which new data keys are to be uploaded. In the array `e`, a new data key for each sensor `n` on the sensor node referred to by `bn` can be given by specifying the key's identifier, i.e., its SHA-1 hash value, in the field `kid` and the keying material (in `base64url`-encoded form) in the field `k`. Single sensors of a sensor node MAY be omitted from the array `e`, e.g., if the readings of different sensors are subject to different security requirements and therefore use data keys of differing validity periods.

Data Key Download. In order to be able to decrypt and then process data items, cloud services must first obtain the respective data keys. To this extend, cloud services send messages of the following structure to the cloud:

```
{
    "typ":"401", /* message type */
    "gw":"<string>", /* gateway ID */
    "srv":"<string>", /* service ID */
    "kid":"<string>", /* key ID */
}
```

The message type `typ` of `401` indicates a data key download message. In this message, the service identified by `srv` requests to download the data key identified by `kid` and managed by the gateway identified by `gw` from the cloud. The service includes its own identifier into the message in order to enable the cloud to respond with the data key specifically encrypted for the requesting service. Furthermore, the cloud responds with a message of type `400` as specified in the previous section.

Public Key Upload. SensorCloud does not feature a designated upload message for public keys in the SensorCloud protocol. Instead, asymmetric key pairs, which determine an entity's identity, are expected to be issued by the cloud. For instance, the cloud MAY require gateways and services to be registered by other means, e.g., via a web front-end, and issue a client certificate upon successful registration. During this process, the respective data owner or service provider locally creates a key pair and in return obtain the client certificate, which is bound to the data owner or service provider's real identity.

Public Key Download. In contrast to the creation of public keys, their retrieval from the cloud must be an automated process. The corresponding message of the SensorCloud protocol has the following structure:

```
{
    "typ":"402", /* message type */
    "id":"<string>" /* entity ID */
}
```

The message type `typ` of `402` indicates that a public key is about to be downloaded. The requesting entity specifies the identifier of the entity of which the public key shall be retrieved in the field `id`. On the one hand, gateways need to request services' public keys in order to provide them with the data keys needed to process sensor data. On the other hand, services need to request a gateway's public key in order to verify the integrity of sensor data downloaded from the cloud.

The cloud then responds with the respective public key, using a message of the following structure:

```
{
    "typ":"403", /* message type */
    "key":"<string (PEM-encoded)>" /* public key */
}
```

The message type `typ` of `403` indicates a message that contains a public key that is being obtained from the cloud. Furthermore, the field `key` holds the respective entity's public key encoded using the PEM encoding [18].

5 Conclusion

When securely outsourcing sensor data to the cloud, it is important to standardize the representation of sensor data as well as the encoding of the necessary security mechanisms. In this paper, we presented the SensorCloud protocol, which has been developed as a joint effort in the SensorCloud project as a common representation for sensor data and actuator commands as well as the necessary security mechanisms using JSON. The SensorCloud protocol is one important building block of our trust point-based security architecture for sensor data in the cloud and the SensorCloud security library which realize the secure outsourcing of sensor data to the cloud [4, 5, 11, 12, 14]. In our design of the SensorCloud protocol, we intentionally relied on approaches that have been or currently are in the process of being standardized at the IETF. This does not only ease the wide applicability of our approach but also provides a best practice on how to utilize and combine these standardized building blocks in an actual system.

5.1 Additional Reading

This paper deliberately focuses on the documentation of the SensorCloud protocol. Further information on the underlying security architecture has been presented in a number of scientific publications as follows.

First conceptual and prototypical considerations of the SensorCloud security architecture have been presented at IEEE CloudCom 2012 [14]. This publication outlines the designed security architecture and shows its feasibility through initial measurements. Subsequently, we published an extension of our SensorCloud security architecture in the International Journal of Grid and High Performance Computing [11]. In this work, we present the integration of the SensorCloud protocol into the design of our security architecture and discuss extended measurements of our prototypical implementation. The scientific concept of our SensorCloud Security Library has been presented at AASNET 2014 [5]. Finally, we contributed a chapter to the edited book *Trusted Cloud Computing*, where we provide a complete overview over the developed trust point-based security architecture [12].

In addition, we describe the interdisciplinary approach of the SensorCloud project [4] and discuss potential extensions to also provide privacy of outsourced sensor data [7, 8].

Acknowledgments

The authors would like to thank everyone who contributed to this document (see attached list of contributors). The SensorCloud project was funded by the German Federal Ministry of Economic Affairs and Energy (BMWi) under project funding reference number 01MD11049. The responsibility for the content of this publication lies with the authors.

List of Contributors

The following individuals contributed to this specification of the SensorCloud protocol:

– Anupam Ashish, QSC AG
– Benjamin Assadsolimani, RWTH Aachen University
– Daniel Catrein, QSC AG
– Dominik Chmiel, RWTH Aachen University
– Martin Henze, RWTH Aachen University
– Lars Hermerschmidt, RWTH Aachen University
– René Hummen, RWTH Aachen University
– Roman Matzutt, RWTH Aachen University
– Antonio Navarro Pérez, RWTH Aachen University
– Thomas Partsch, Cologne University of Applied Sciences
– Christian Röller, QSC AG
– Daniel Scholz, Cologne University of Applied Sciences
– Andre Skusa, symmedia GmbH

References

1. Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E.: A Survey on Sensor Networks. IEEE Communications Magazine 40(8) (2002)
2. Barker, E.: Recommendation for Key Management Part 1: General. Tech. rep. (jan 2016), http://dx.doi.org/10.6028/nist.sp.800-57pt1r4
3. Birrell, A.D., Nelson, B.J.: Implementing Remote Procedure Calls. ACM Transactions on Computer Systems 2(1) (1984)
4. Eggert, M., Häußling, R., Henze, M., Hermerschmidt, L., Hummen, R., Kerpen, D., Navarro Pérez, A., Rumpe, B., Thißen, D., Wehrle, K.: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators. In: Krcmar, H., Reussner, R., Rumpe, B. (eds.) Trusted Cloud Computing. Springer (2014)
5. Henze, M., Bereda, S., Hummen, R., Wehrle, K.: SCSlib: Transparently Accessing Protected Sensor Data in the Cloud. In: The 6th International Symposium on Applications of Ad hoc and Sensor Networks (AASNET). Procedia Computer Science, vol. 37. Elsevier (2014)
6. Henze, M., Großfengels, M., Koprowski, M., Wehrle, K.: Towards Data Handling Requirements-aware Cloud Computing. In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science. IEEE (2013)
7. Henze, M., Hermerschmidt, L., Kerpen, D., Häußling, R., Rumpe, B., Wehrle, K.: User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In: 2014 International Conference on Future Internet of Things and Cloud (FiCloud). IEEE (2014)
8. Henze, M., Hermerschmidt, L., Kerpen, D., Häußling, R., Rumpe, B., Wehrle, K.: A Comprehensive Approach to Privacy in the Cloud-based Internet of Things. Future Generation Computer Systems 56 (2016)
9. Henze, M., Hiller, J., Hohlfeld, O., Wehrle, K.: Moving Privacy-Sensitive Services from Public Clouds to Decentralized Private Clouds. In: 2016 IEEE International Conference on Cloud Engineering Workshops. IEEE (2016)
10. Henze, M., Hiller, J., Hummen, R., Matzutt, R., Wehrle, K., Ziegeldorf, J.H.: Network Security and Privacy for Cyber-Physical Systems. In: Song, H., Fink, G.A., Jeschke, S., Rosner, G.L. (eds.) Security and Privacy in Cyber-Physical Systems: Foundations and Applications. Wiley (2016), to be published
11. Henze, M., Hummen, R., Matzutt, R., Catrein, D., Wehrle, K.: Maintaining User Control While Storing and Processing Sensor Data in the Cloud. International Journal of Grid and High Performance Computing (IJGHPC) 5(4) (2013)
12. Henze, M., Hummen, R., Matzutt, R., Wehrle, K.: A Trust Point-based Security Architecture for Sensor Data in the Cloud. In: Krcmar, H., Reussner, R., Rumpe, B. (eds.) Trusted Cloud Computing. Springer (2014)
13. Henze, M., Hummen, R., Wehrle, K.: The Cloud Needs Cross-Layer Data Handling Annotations. In: 2013 IEEE Security and Privacy Workshops (SPW). IEEE (2013)
14. Hummen, R., Henze, M., Catrein, D., Wehrle, K.: A Cloud Design for User-controlled Storage and Processing of Sensor Data. In: 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom). IEEE (2012)

15. Jennings, C., Shelby, Z., Arkko, J., Keranen, A.: Media Types for Sensor Markup Language (SenML). IETF Internet-Draft draft-ietf-core-senml-00 (2016), https://tools.ietf.org/html/draft-ietf-core-senml-00
16. Jones, M., Bradley, J., Sakimura, N.: JSON Web Signature (JWS). IETF RFC 7515 (Proposed Standard) (2015), https://www.ietf.org/rfc/rfc7515.txt
17. Jones, M., Hildebrand, J.: JSON Web Encryption (JWE). IETF RFC 7516 (Proposed Standard) (2015), https://www.ietf.org/rfc/rfc7516.txt
18. Josefsson, S., Leonard, S.: Textual Encodings of PKIX, PKCS, and CMS Structures. IETF RFC 7468 (Proposed Standard) (2015), http://www.ietf.org/rfc/rfc7468.txt
19. One Laptop per Child: Canonical JSON, http://wiki.laptop.org/go/Canonical_JSON
20. Pearson, S., Benameur, A.: Privacy, security and trust issues arising from cloud computing. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom). IEEE (2010)

2014-03	Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide
2014-04	Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06	Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07	Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations
2014-08	Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs
2014-09	Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014
2014-14	Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks
2014-15	Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code
2014-16	Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results
2015-01 *	Fachgruppe Informatik: Annual Report 2015
2015-02	Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications
2015-05	Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
2015-06	Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"
2015-07	Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon
2015-08	Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization
2015-09	Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models
2015-11	Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus
2015-12	Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic
2015-13	Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2015-14	Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines
2016-01 *	Fachgruppe Informatik: Annual Report 2016