Choosing security elements for the xAAL home automation system

Christophe Lohr, Philippe Tanguy, Jérôme Kerdreux

Choosing security elements for the xAAL home automation system

Christophe Lohr, Philippe Tanguy and Jérôme Kerdreux
Telecom Bretagne, Technopôle Brest-Iroise, CS 83818, 29238 Brest Cedex 3, France
Email: christophe.lohr@telecom-bretagne.eu philippe.tanguy1@telecom-bretagne.eu jerome.kerdreux@telecom-bretagne.eu

***Abstract*—The emergence of Internet of Things (IoT) and smart-home systems allows us to combine devices from different domains and to explore new usages and services. Unfortunately interoperability between devices from different technologies is a major issue to overcome before being able to offer smart services. For this purpose we have proposed the xAAL system. It is both a federating home-automation protocol and an open infrastructure designed to address issues caused by the heterogeneity of existing home-automation solutions. xAAL has been implemented, deployed and has proved its efficiency. However, early versions have been designed with functional concerns in mind. The time has come to address security. xAAL has its own specificities: a distributed system, multicast communications on a bus, etc. This paper details choices, compromises and motivations for selecting security elements that have been introduced in the new version of xAAL.**

1. Introduction

The interoperability between home-automation solutions is a key issue nowadays: how to make a device from vendor A (e.g., a switch) talking with a device from vendor B (e.g., a lamp)?

In fact, the current situation could be compared to the well known *prisoner's dilemma*: each one act for its own little benefit, without collaboration with others, and finally there is no winner [1]. Manufacturers of home-automation systems would certainly have benefit by opening their solution but no one wants to do the first step of fear to be eaten by the concurrency. So, each one has to reinvent yet another home-automation box claiming to provide all functionalities, all possible services, in every domains... without a great success for end users.

To address this issue the xAAL system has been proposed [2]. xAAL is a functional distributed architecture; all components talk to others via an IP (multicast) bus. The communications are in the form *many-to-many*. Messages are in JSON format. The set functions traditionally implemented by home-automation boxes are arranged into well-defined separated functional components: elementary gateways for technology-specific devices (translating one vendor-specific protocol into xAAL), native-xAAL devices (that embed the xAAL stack), caches, configuration database, user interfaces, scenarios automata, etc. Each component may have multiple instances, may be shared by several boxes, and may be physically located on any box. Figure 1 depicts the general architecture of xAAL.

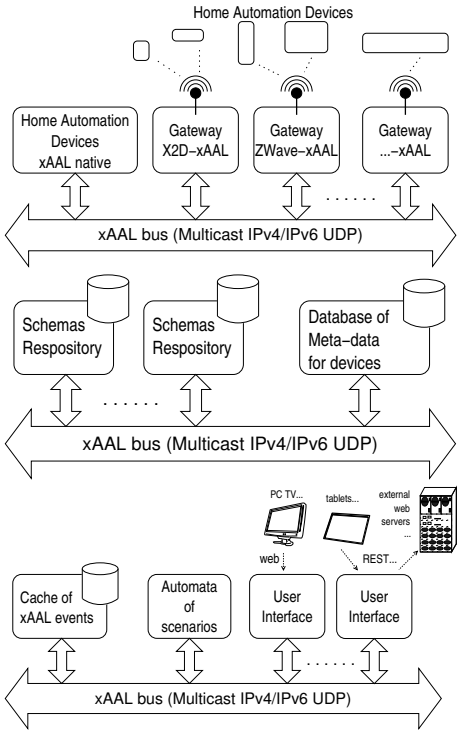

Figure 1. Functional Architecture of xAAL.

The previous version of xAAL proposed messages in clear text, with an optional cryptographic signature. Traditionally, security is not a high concern in old home-automation systems (X10, HomeEasy, etc.), even for device involved in alarms systems. However, new solutions began to embed cryptographic elements; sometime poor (RC4, rolling codes, etc.), sometime rather strong (RSA, DSA, AES, etc.). xAAL must not introduce weakness. Its communication bus over IP could be a place for attacks. Furthermore, securing many-to-many communications (a bus) is a big challenge rarely addressed by classical security layers, which consider one-to-one communications. The current paper describes experiments and choices leading to the security elements proposed for the new version of xAAL. It does not provide new functionality to xAAL except communications ciphering.

This document is organized as follows. The section 2 details preliminary studies and tests that have been driven with very pragmatic concerns in mind. The section 3 specifies the new message format with the security elements. Section 4 concludes this paper.

2. State of the art and preliminary studies

2.1. Constraints and needs

The objective is to introduce ciphering for the xAAL home-automation system. Among its specificities, this is a distributed and an open system designed for heterogeneous devices. Concretely, a consumer may acquire and plug new devices from different vendors at any time. This implies very little configuration stage for the end user, just like configuring a WiFi connection.

The second major point is the choice of a bus for communications between xAAL components. This is used for automatic discovering, for event messages and also for control-command. Therefore, there is no hand check between devices, there is no room for any preliminary keys or cipher exchange between entities before communications. This is why existing solutions such as DTLS (*Datagram Transport Layer Security* [3], [4]) cannot be reused as it. Indeed, existing security layers are mostly designed for *one-to-one* communication principles. Therefore, sent messages must be self-content in terms of ciphering elements. Receivers must be able to check messages without any context.

2.2. Signature in the previous version of xAAL

The previous version of xAAL proposes authenticating messages with a signature and a private shared key. For this purpose, messages had three fields: (i) *cipher*: the name of the ciphering mechanism used to sign the message; (ii) *signature*: the signature of the message by the selected signing algorithms; (iii) *timestamp*: the date of the message in seconds since 1970-01-01 00:00:00 UTC, to avoid replay attacks.

This specification have been coded and tested with mixed outcomes.

2.2.1. A private shared key. Several types of security key may be considered:

A private shared key: Each home-owner chooses a secret key for its home-automation network and sets it into each xAAL device at home.

A private/public couple of keys certified by an authority: keys are generated at factory and preloaded on each device. The public key is signed by an authority to proof that the device is the legitimate owner of the key (e.g., TLS [5]). The signature of the authority is preloaded on each device in order to check signature of received messages.
But who is legitimate to play the role of *the authority*? Which establishment has the infrastructure to do the job seriously on the long term? Shall we introduce a business around certifications? Moreover, this drastically breaks the xAAL principle of *a distributed system for heterogeneous devices*. Also, keep in mind that in such systems, leaks of the signatures may happen: the signatures of the (intermediate) authorities can be revealed and reused to sign illegitimate keys. So, revocation protocols have to be introduced.

A private/public couple of keys signed by friends: keys are generated at factory and preloaded on each device. The public key is signed by *friends*, or by friends of friends (e.g., PGP [6]). *Friends* can be other xAAL devices of the home-automation network.
Before starting to emit messages on the bus, each device has to send its public key to a certain number of its friends and to ask them to certify its key with their key. It will also certify keys of friends with its certified key. This will make a chain of certifications.
However one may get isolated devices (i.e., friend with no one, without a certified key), or too popular devices (i.e., with many friends and a redundant chain of certifications). Moreover, there is still the question of the bootstrap: how to know its first true friends? Possibly the initial key of devices could be printed on a sticker on the device and scanned by the installer and pasted into the configuration of friend's devices.

As written in the previous version of the xAAL specifications, using a pre-shared key seems acceptable in the context of a home network: it is the choice of WiFi WPA/WPA2 Personal and HomePlug AV Powerline. This introduces a configuration stage to set the secret key into each xAAL device, however this greatly eases the implementation compared to using asymmetric keys.

2.2.2. Multiple signing algorithms. The previous version of the xAAL specifications allows multiple signing algorithms. An xAAL device may support several signing algorithms. The *cipher* field on an xAAL message tells the signing algorithm used to sign it.

Supporting multiple cryptographic algorithms is recommended by the IETF [7]. It is quite common on classical security layers (e.g., SSL, TLS, SSH, etc.): with the progresses of the science of the cryptography, weaknesses on some algorithms may be discovered, and new stronger algorithms may be designed; the cryptographic algorithms change, but the protocol remains the same. For instance RC4 has been withdrawn from TLS, and Chacha20 has been added [8]. Note that allowing such cryptographic algorithm agility increases the complexity of the protocol: additional signalling communication has to be added to negotiate cipher between communicating peers, one has to ensure that participants choose the strongest algorithm available in common and a minimal set of mandatory algorithms has to be chose and maintained.

Having such a flexible security layer is very comfortable. However, implementing such a flexible security layer is very hard, uncomfortable and a source of errors.

This leads to some questions about the design: since xAAL communications are in a *many-to-many* way (i.e., messages on a bus), there cannot be a preliminary nego-

tiation stage in which a sender and a receiver agree on a cryptographic algorithm. A sender emits a message to several receivers. So, which cryptographic algorithm to choose? Moreover, if many cryptographic algorithms are allowed at the same time on the same bus, there can be several separated conversations in parallel where devices may (or may not) understand conversations of others: possibly one "channel" per cryptographic algorithm. This breaks some other design points of XAAL: a distributed collaborative system. To avoid this one would have to ensure that there is at least one cryptographic algorithm shared by all devices of the bus among the list of ones they accept. So, why not having just one, once and for all?

Well, one will not talk about another ugly solution: add some proxies on the bus to translate messages from one cryptographic algorithm to another one. This duplicates messages for devices that understand the two cryptographic algorithms and may flood the bus by introducing loops if there are two proxies doing the translation in the two ways. So, forget it. Definitively.

As a consequence, it has been decided that the new version of XAAL will support one and only one cryptographic algorithm. So, the cryptographic algorithm agility recommended by the IETF will take the form in the XAAL context: when the time will come to change the ciphering algorithm, a new version will be edited.

2.2.3. A "signature" field inside the message. The previous version of XAAL specifies that the signature is written inside the message it signs. This leads to some difficulties.

Nowadays a signing mechanism has been implemented and tested. It uses HMAC SHA256 on messages with the *signature* field previously padded with zeros. The signature is then encoded in hexadecimal and replaced within the message.

This trick is rather usual: the checksum of TCP/UDP/IP packets is computed with the field padded by zeros, and once one has the right value, one writes it in the final packet.

The difficulty in XAAL is the operation of writing the *signature* field inside the message after computation on the JSON serialization of data.

There will be several sides' effects if this is performed by using a JSON library. Indeed, JSON is not a canonical format: the same data may be serialized into several ways, by adding spaces or new lines, by reordering fields inside objects, etc. This is allowed. For instance, some JSON library removes all spaces and systematically places the fields in the alphabetical order. As a consequence, after writing the *signature* field inside the message, the serialization may be very far from the one on which the signature has been computed. The receiver has no chance to retrieve the original way in which the message was serialized. It cannot check the signature.

To avoid this, one has to use something else than a JSON library to write the *signature* field. For instance, one can play with pointers inside the buffer of the message. This is feasible; however this is rather inelegant and non desirable.

As a consequence, it has been decided that new XAAL version will place the cryptographic elements outside of the message to be secured.

2.2.4. A timestamp as the nonce. The replay attack is a classical issue in security. For instance, consider the alarm of a house that can be activated and deactivated remotely via a secure channel with ciphering but without replay attack protection. A robber could record the deactivating message emitted when the home-owner stops its alarm when he arrives at home. Then, when the home-owner is away, the robber could re-inject this message as-is. He does not need to decrypt it; he does not need to break the key. The message is perfect as this. The alarm will accept it and will stop.

To avoid this, each message must be different, even those that say the same thing. For that an extra field is added inside messages whose value is different on each message: a *nonce*. A nonce does not need to be secret, a nonce does not need to be random; it just has to be different on each message.

Another required property is that a nonce must be checked by the receiver. It cannot be completely free: if the receiver accepts any nonce value, it will accept the nonce of the old replayed message, and will not avoid the replay attack. Typical security protocols have a preliminary challenge-response stage. The receiver itself chooses a value for the nonce, and indicates it to the sender. Then the sender uses it in the messages it sends. On receipt, the receiver checks that the nonce in the message is the one it expects. This strategy is fine for *one-to-one* communication schemes. However, XAAL has *many-to-many* communications.

A first naive strategy would be to ask all devices to keep in memory all past values used for the nonce, and to compare any incoming message to this list. This is not realistic.

Another strategy is to use a counter, a kind of message-id. Each device listens messages on the bus, records the last used message-id. On receipt, the device checks that the message-id in the received message is consistent with the message-id it recorded previously. At emission, it increments its recorded message-id and uses it as the nonce in the message it sends. This is more realistic, however the devices must stay awake to listen to the bus and remain synchronized to this counter. Unfortunately some home-automation devices are sleeping to save battery (sometime for a long time).

Finally, it has been decided in the previous XAAL version to use the time that passes for this counter. The nonce is a timestamp, the number of seconds since *Epoch*, 1970-01-01 00:00:00 +0000 (UTC). From a practical point of view, it is much more feasible to remain synchronized with the time, at least with a precision of several seconds or minutes. The dormant devices may embed a small clock; that consumes much less battery than listening to the network.

Strictly speaking, this does not fully avoid a replay attack. Indeed, one has to introduce an acceptance window of few seconds (or few minutes): only messages that are too old (outside this acceptance window) are rejected. From the

hardware point of view, having a precision of a few seconds (or few minutes) is OK. Requiring a stronger synchronization may become very hard. We want a good security but at an acceptable price. Therefore, a replay attack is still feasible within this acceptance window. However, the replay attacks have a rather limited interest: the attacker cannot send all messages he wants, he can just blindly inject some messages that legitimate users have emitted just before. A window of few seconds (or few minutes) is compatible with usages of people: if the robber tries to replay the deactivating message of the alarm within this delay, there are high chances that the home-owner is still in the place (and that the alarm is already off)... Moreover, due to the choice of UDP as the transport protocol, XAAL applications have to be designed to be resilient to packet losses and packets duplications. Commands of devices are designed to support messages duplications; for instance a lamp has separated *on/off* methods, but no *trigger*. In case of critical situation and to avoid replaying commands, devices should keep in memory a hash of treated requests within the acceptance window.

Timekeeping is a common issue of distributed systems. In fact there are two problems: first knowing what time it is, and then staying synchronized without much drift. Everything depends on the required precision. There are several technical solutions: the clock of a GPS, the atomic clock of Frankfurt transmitted via the DCF77 longwave radio signal (fine in Europe), and the clock transmitted via Radio Data System by local radio stations, etc. Those solutions are quite expensive and unrealistic for small home-automation devices.

A more realistic solution is to get the time via Internet (since XAAL devices are already on the network). The plain old *Time Protocol* [9] is no more in use. Nowadays, the *Network Time Protocol* (NTP [10]) is used. Many public servers are available. (Private servers, usually those of the "first strata", require authentication.) According to the algorithm specifications of NTP, the typical accuracy on the Internet ranges from about 5ms to 100ms. This is too precise for XAAL. NTP also includes a variant: the Simple Network Time Protocol (SNTP), with an accuracy of about one second. In fact NTP and SNTP use the same protocol (from outside there is no way to distinguish if the client or the server implements NTP rather than SNTP). The differences are in the internal routines to mitigate several time sources and to compensate clock drifts, which may consume CPU resource with NTP. Another more lightweight strategy is the *ntpdate* approach: this application just asks once the time is it to an NTP server and sets it to its internal clock. For instance this is also the strategy of the NTP implementation in Arduino [11]. (This is more or less the way the plain old Time Protocol do the job.)

Note that there are very few attacks on the NTP protocol. Sometimes NTP servers are used to collect IPs of clients for later attacks, or NTP servers may be used to amplify distributed denial-of-service (DDoS) attacks, but there is no real attack on the NTP protocol itself. In fact, it is pretty hard to fool an NTP client on the time of the day. This is also a reason why there will not be an XAAL device that gives the time to other devices. First, there are little chances that we design something better than NTP. Then, such a device would be a prime target for an attacker. And finally, XAAL communication with this device could not be secured, since devices will not be synchronized.

The *ntpdate* approach (or similar) is a low-cost strategy that addresses the first issue: knowing what time it is. Remains the second issue about keeping its internal clock synchronized without too much drift. Typically, one uses quartz crystal oscillators. The accuracy of such quartz crystal oscillators depends on the temperature variations. Fortunately, it is shown that this compensates itself along day's cycles. The accuracy is of 1 second per day, 15 seconds per month, 1 minute per year [12]. It is good enough for XAAL.

As a consequence, it has been decided that new XAAL version, as for the previous one, will use a timestamp as a nonce with an acceptance window of few seconds or few minutes.

2.3. Ciphering in the previous version of XAAL

Signed messages provides authentication: a device has the proof that a command comes from a legitimate sender. However, the content of the message is still in clear. A spy may know that a given message is the one to deactivate the alarm, or that someone is doing something in the bathroom... To avoid this, messages have to be ciphered.

The specification of previous XAAL version does not propose ciphering. However, some tests and implementations have been driven, with little arrangements of the format of messages.

2.3.1. Security with Poly1305/Chacha20. As stated above, it has been decided to use one and only one security algorithm for XAAL. Among the large list of well-known algorithm, Poly1305/Chacha20 has been selected [13], [14]. According to the author, it is at least as stronger than others (e.g. AES); it is much faster, requiring less memory, less CPU. (It works on an Intel 8051! [15])

Even if Poly1305/Chacha20 is not necessarily well known by non-experts in security, it is now in the cipher suites for TLS 4. Several libraries are available in different programming languages.

2.3.2. A timestamp as the binary nonce. As stated above, a timestamp can be used for the nonce. Chacha20 supports a nonce of 64 bits. Some systems return the number of seconds since Epoch on 64 bits. Some others return it on 32 bits. In fact, this will be the same until Sunday February 7 2106 at 07:28:15 UTC. In the mean time, the other bits are just zeros. So the unused bits can be used to code something else that varies, for instance microseconds. As a result one can build a nonce that changes a lot. In terms of implementation, this is quite usual to get seconds and microseconds on systems by functions like *gettimeofday()*. (Modern systems

claim to provide nanoseconds; microseconds are enough for us.)

In fact, coding microseconds requires 20 bits. Coding seconds since Epoch requires today 31 bits. In one hour 12 bits are changing. As a result, a nonce built as described above (seconds on 32 bits + micro-seconds on 32 bits) will have a variability of more or less 32 bits. Well, among other compromises made about the nonce, we consider that this is good enough.

To sum up, having such timestamp in messages (i.e., seconds since Epoch + microseconds since the beginning of this second) can be used for two things: first as a cryptographic nonce for the Poly1305/Chacha20 algorithm, and also to check the age of the message within a temporal window (just in looking at seconds).

Note about the *millennium bug* (i.e., Sunday February 7 2106 at 07:28:15 UTC), if no other XAAL version is proposed before this date: counters will loop back, but the nonce will be computed in the same way. Nothing special will occur, except that some devices will loop back before others, depending on the precision of their clock (that could be of few seconds or minutes). So, regular packets may be rejected. (A priori this does not matter: remember that XAAL does not provide warranty on the transmission of messages.) Once everyone looped back, messages are accepted as before.

Note about the size of the nonce: The original Chacha20 algorithm uses a nonce of 64 bits. The RFC 7539 [16] that adds Chacha20 into TLS modifies it to support a 96 bit nonce, in order to fit TLS recommendations. Indeed, the nonce may be generated by a pseudo-random function; a larger nonce may avoid collisions. Collisions of nonce are not very serious, but it is best to avoid them. According to the way XAAL builds the nonce, there could be collisions between packets before and after February 2106, or if two packets are emitted on the bus on the same time on the same microsecond: a priori this is managed by collision avoidance mechanisms of Ethernet or WiFi. Nevertheless, collision of nonce may happen in theory if two devices have slightly desynchronized clocks, just enough for the messages they emit just one after the other will exhibit the same timestamp with the same microsecond. It is assumed that this scenario is extremely rare. And if it happens, this should cause nothing special in practice.

2.3.3. Targets as public data. Through encryption, messages are unreadable for those that do not have the key. However, this becomes heavy if all devices must decrypt all messages, even those that are not for them, just to know if messages are for them or not. For efficiency, devices should be able to filter messages on the targets field (i.e., compare targets addresses of the message with their own address) before decryption.

Fortunately, the Poly1305/Chacha20 algorithm proposes an AEAD mode (*Authenticated Encryption with Additional Data*). With this mode, the message is encrypted and signed, but the signature may also cover additional data that can appear in clear. Such an *additional data* is the right place for the targets field. Devices can quickly filters received messages, and the field is also protected by the key. An attacker cannot rebuild and inject a message by taking the encrypted part of a message and add the target field of another one.

Note: for the same motivation, the field *version* (which indicates the XAAL protocol version of the current message) has also been moved as public data to ease devices to quickly filter messages. However, it was also covered by the signature. But this is perhaps unnecessary. For now, we cannot find a scenario of an attack based on tricking the version field.

2.3.4. A binary security layer. To experiment ciphering of XAAL, the original message format has been modified. The fields *cipher* and *signature* have been removed. The fields *targets*, *timestamp* and *version* have been moved outside.

It would have been fine to present the moved fields and the security elements in a JSON format also, that could have been placed before or after the encrypted message. Unfortunately, because JSON is non canonical, there can be extra spaces before the first opening brace, or after the last closing brace. There is no way to really know where does start a JSON message, and where it ends. Most JSON libraries skip the spaces before the first opening brace, and stop decoding at the closing brace. This is a valid interpretation of the JSON serialization, but there could be others. Those extra spaces, if any, could belong to the JSON message itself, or to a data that is placed side to the JSON. As a consequence, if one puts the encrypted message side to those JSON fields, there is no way to really know where the encrypted part starts or ends. The receiver cannot be sure it is decrypting the right data.

Finally, for this experiment, it has been decided to present the moved fields and the security elements in a binary format rather than in a JSON format, just before the encrypted message, which is also in a binary format. This is the so-called *security layer*.

The binary format of this security layer is as follows:

The *version*: composed by a *major* and a *minor* number (two unsigned on 8 bits);
The *targets number*: an unsigned on 16 bits in big endian (network byte order);
The *targets*: a vector of a size of *targets number* where each cell is an uuid on 128 bits (a vector of 16 bytes);
The *public nonce*: composed of *seconds* since Epoch as an unsigned on 32 bits in big endian, and of *microseconds* as an unsigned on 32 bits in big endian.
The *application layer*: the payload of the security layer. It is built by encrypting the usual XAAL JSON message (without the field *version targets cipher signature timestamp*), using the Poly1305/Chacha20 algorithm and the above nonce. The "public data" is the buffer composed by *version*, *targets number* and *targets*.

The tests conducted have shown that it is feasible and that it is pretty effective.

There are some cons against this solution. First, the devices addresses (uuid [17]) are presented in two ways: in a binary way in the security layer (the targets), and in a JSON way inside the application layer (the source). This is rather inelegant. And then, the implementation is source of error (think to segmentation fault while handling the variable size of the vector of targets), and complicated in other languages than C.

The tests convinced us that using an existing format to present elements of the security layer is preferable to a binary ad-hoc format, even if it is less compact.

2.4. JavaScript Object Signing and Encryption (JOSE)

The JOSE working group of the IETF aims to provide security of JSON messages. A series of RFCs are proposed:

Use Cases and Requirements for JSON Object Signing and Encryption (JOSE) [18]: This gives the frame for the JOSE working group.
JSON Web Signature (JWS) [19]: This specifies the message format and cryptographic mechanisms used to sign and authenticate messages.
JSON Web Encryption (JWE) [20]: This specifies the message format and cryptographic mechanisms used to cipher messages.
JSON Web Key (JWK) [21]: This specifies the format of keys.
JSON Web Algorithms (JWA) [22]: This describes registers where cryptographic algorithms are recorded. (They are not recorded inside RFCs but in registers pointed by RFCs. This allows more flexibility.)
Examples of Protecting Content using JavaScript Object Signing and Encryption (JOSE) [23]: A cooking-book with examples and best practices.

Strictly speaking, JOSE is more a way to express cryptographic elements into a JSON format rather than a solution to encrypt JSON messages. The payload could be something else than a JSON message, even if everything was designed with JSON in mind.

JOSE is very flexible. Numerous cryptographic mechanisms are possible. Surprisingly, Poly1305/Chacha20 is not in the list at the time this document is written (2016).

A key point of JOSE is an intensive use of the base64 encoding/decoding. Indeed, encrypted data are in a binary form by design, while JSON is in a textual form by design. The use of base64 is rather natural. For the implementation this requires new buffers to store data while encoding/decoding. However, base64 is typically used in JSON contexts. This strategy could be used for XAAL: rather than placing ciphered data *before* or *after* the security layer, this can be placed *inside* as a base64 encoded string.

For now there are very few programming libraries for JOSE. But JOSE is rather young (May 2015). This will come later.

As a conclusion, for now JOSE is too complex (flexible) to be used within XAAL. However, if in the future it is decided to support several cryptographic mechanisms within XAAL, JOSE could become a good candidate.

2.5. Concise Binary Object Representation (CBOR)

The *Concise Binary Object Representation (CBOR)* [24] specifies a format to serialize data in a binary way. In short, CBOR does exactly the same things as JSON, except that the result is in binary rather than in text. Moreover, CBOR can handle binary data directly, without base64 encoding/decoding.

However, like JSON, CBOR is not a canonical format: the same data can be serialized in several ways. Fields can be reordered, numbers and lists may be serialized in different way. There are no more issues with extra spacing, but things are not perfect neither. A special tag (*0xd9d9f7*) can be used to express when a CBOR serialization starts within a byte stream, but there is no tag to express when a CBOR serialization ends within the bytes stream (and that something else is starting after).

Today there are some programming libraries (or piece of code) for CBOR. CBOR seems to become more and more used, and more specifically in the area of Internet of Things (well, it has been designed for that). Moreover, there are several efforts to provide security on CBOR, as JOSE does for JSON. The IETF recently published drafts of RFCs for this [25].

For now, JSON is fine for XAAL. This is a well-known format, and messages are in clear text (readable by humans). Those points are important for the promotion of XAAL and the acceptance by developers. Even if today it is too early, it is highly possible that a future release of XAAL will use CBOR in the place of JSON. This could improve messages format without changing XAAL principles and functionalities.

2.6. Findings of the preliminary studies

The above studies lead us to draw major principles for securing XAAL. To sum up, the main points are:

- A pre-shared symmetric key;
- Poly1305/Chacha20 as the only cryptographic algorithm;
- A binary nonce built as a timestamp since the Epoch (seconds + microseconds);
- An acceptance window for the timestamp of messages;
- The list of targets in clear, but covered by the signature;
- A security layer in JSON;
- An application layer in JSON, very close to previous XAAL releases;
- The ciphered application layer encoded in base64 and inside the security layer as a string.

3. Defining security elements for the new version of XAAL

3.1. Definition of a message

A message is in JSON format. This called the *security layer*. This is a JSON object whose fields are:

- *version*: The string *"0.5"*. (The version of the protocol.) Other values should be rejected.
- *targets*: A string built as the JSON serialization of the array of destination addresses (uuid) of the message. An empty list means a broadcast message. An empty string is not allowed (the message should be rejected).
- *timestamp*: An array of two (and exactly two) integers: the first number is the number of seconds since the Epoch (1970-01-01 00:00:00 +0000 UTC), and the second number is the number of microseconds since the beginning of this second.
- *payload*: A string built as the base64 encoding of the ciphered *application layer*.

The *application layer* (which is embedded inside the *security layer*) is built as described in previous version of XAAL with the following modification: the fields *version*, *targets*, *cipher*, *signature* and *timestamp* are withdrawn of the *header*. The rest is the same.

Figures 2 and 3 give an example of an XAAL message (the security layer) with its decoded payload (the application layer).

```
{ "version": "0.5",
  "targets": "[ \"174255ad...dbcdfc812d4\" ]",
  "timestamp": [ 1439824426, 467313 ],
  "payload": "8sbrvczRc5Np...SAUc5Dj9SKoe82="
}
```

Figure 2. Example of an XAAL message (*security layer*)

```
{ "header": {
    "source": "06b71935-...-dae3d3b8ce77",
    "devType": "thermometer.basic",
    "msgType": "reply",
    "action": "getAttributes"
  },
  "body": {
   "temperature": 33.0
  }
}
```

Figure 3. The decrypted payload of an XAAL message (*application layer*)

3.2. Applying Poly1305/Chacha20

3.2.1. The targets array as a string. Please note that the *targets* field is a string. This is not an array of uuids, this is the JSON serialization of an array of uuids.

Two arguments for this: first, it is not so complicated for a device to parse it and to check if the message is for it or not. Then, this string may bee seen as a buffer of bytes and can directly be used as the *public additional data* for the Poly1305/Chacha20 algorithm.

3.2.2. The timestamp as an array of two integers. The format of this field is mapped on the function *gettimeofday()* (SVr4, BSD 4.3. POSIX.1-2001...) which return the date of the day as a pair of two unsigned numbers (seconds and microseconds). Just send it as it.

Then, the binary nonce (64 bits) to be used with Poly1305/Chacha20 is composed of the seconds and microseconds (in this order) as two 32 bits unsigned in big-endian.

3.2.3. The encrypted payload encoded in base64. Encrypted data are encoded/decoded in base64 [26]. Due to its heritage from the email, the base64 encoding may insert line breaks every 72 chars, this is unnecessary here.

3.3. Recommendations

3.3.1. To build the cryptographic key from a passphrase. The Poly1305/Chacha20 algorithm uses a binary key on 256 bits. A fine way to select a "good" key is to build it from a passphrase using a cryptographic hashing algorithm.

It is proposed to use the dedicated function provided for this purpose in the reference Chacha20 library (the *sodium* library), and derived libraries: the *crypto_pwhash_scryptsalsa208sha256()* function [27]. Recommended parameters are used: 512k cycles for the *opslimit* and 16 Mbytes for the *memlimit*.

The only point is the question about choosing the salt. Ideally, the salt should be something which differs from one home-automation network to another home-automation network, but should also be well known by each device on the same home-automation network.

For instance, the key of a WPA-Personnal WiFi connection may be entered either as a string of 64 hexadecimal digits, or as a passphrase of ASCII characters. In such a case, the key is derived from the passphrase HMAC-SHA1 and the name of the WiFi network as the salt (IEEE Std. 802.11i-2004, Annex H.4.1). In the case of XAAL, there is no network name. In the absence of a good idea for defining such a salt, one proposes to use a buffer of zeros.

3.3.2. To choose a window of acceptance for the timestamp. According to our preliminary studies, we think that an acceptance window of two minutes should be fine.

3.3.3. To have several keys on the same bus. An XAAL bus is designed by an IP multicast address, and an UDP port. This is possible to have several security keys on the same bus, as it is possible to use several XAAL versions on the same bus. However, this leads to several communication channels in parallel. The devices with a given security key or a given XAAL version cannot talk to the devices

with another key or version. They will systematically reject plenty of packets. This is inefficient.

If several keys or XAAL versions are needed, it is recommended to use different XAAL buses.

4. Conclusion

The XAAL system is both a federating home-automation protocol and an open infrastructure designed to address issues caused by the heterogeneity of existing home-automation solutions. Among others specificities, XAAL has a *many-to-many* communication scheme on a bus, making unsuitable usual security solution designed to *one-to-one* communications. This paper has presented experiments and studies that have been driven. It detailed choices, compromises and motivations that underlie the design of a new secured communication layer for XAAL. Main points are: (i) A pre-shared private key; (ii) The use of the Poly1305/Chacha20 algorithm; (iii) A timestamp to avoid replay attack, and an acceptance window on the age of messages; (iv) A JSON format for the security layer.

This proposal is coded and tested in our living'lab [28]. The next step is to deploy it in real field and to test it on the long term with real end-users.

Acknowledgments

This project has received funding from the European Community's Seventh Framework Program for research, technological development and demonstration under grant agreement No.611366 (PRECIOUS [29]).

References

[1] A. Chaverot. (2015, juin) Maison connecte: vers un chec? (in french) – Connected Home: towards fail? La Tribune. [Online]. Available: http://www.latribune.fr/opinions/tribunes/maison-connectee-vers-un-echec-480797.html

[2] C. Lohr, J. Kerdreux, and P. Tanguy, "xAAL: A Distributed Infrastructure for Heterogeneous Ambient Devices," *Journal of intelligent systems*, vol. 24, no. 3, pp. 321–331, Aug. 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01187869

[3] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347, january 2012. [Online]. Available: https://tools.ietf.org/html/rfc6347

[4] B. Moeller and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks," RFC 7507, april 2015. [Online]. Available: https://tools.ietf.org/html/rfc7507

[5] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, august 2008. [Online]. Available: https://tools.ietf.org/html/rfc5246

[6] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," RFC 4880, november 2007. [Online]. Available: https://tools.ietf.org/html/rfc4880

[7] R. Housley, "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms," RFC 7696, november 2015. [Online]. Available: https://tools.ietf.org/html/rfc7696

[8] P. Popov, "Prohibiting RC4 Cipher Suites," RFC 7465, february 2015. [Online]. Available: https://tools.ietf.org/html/rfc7465

[9] J. Postel and K. Harrenstien, "Time Protocol," RFC 868, may 1983. [Online]. Available: https://tools.ietf.org/html/rfc868

[10] D. Mills, U. Delaware, J. Martin, J. Burbank, and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification," RFC 5905, june 2010. [Online]. Available: https://tools.ietf.org/html/rfc5905

[11] (2016) Arduino Playground - NTPclient. [Online]. Available: http://playground.arduino.cc/Code/NTPclient

[12] M. A. Lombardi, "The Accuracy and Stability of Quartz Watches," pp. 57–59, February 2008. [Online]. Available: http://www.nist.gov/manuscript-publication-search.cfm?pub_id=50647

[13] D. J. Bernstein, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ch. The Poly1305-AES Message-Authentication Code, pp. 32–49. [Online]. Available: http://dx.doi.org/10.1007/11502760_3

[14] ——. (2016) Poly1305-AES: a state-of-the-art message-authentication code. [Online]. Available: http://cr.yp.to/mac.html

[15] ——. (2016) Poly1305-AES for the 8051. [Online]. Available: https://cr.yp.to/mac/8051.html

[16] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols," RFC 7539, may 2015. [Online]. Available: https://tools.ietf.org/html/rfc7539

[17] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace," RFC 4122, july 2005. [Online]. Available: https://tools.ietf.org/html/rfc4122

[18] R. Barnes, "Use Cases and Requirements for JSON Object Signing and Encryption (JOSE)," RFC 7165, april 2014. [Online]. Available: https://tools.ietf.org/html/rfc7165

[19] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Signature (JWS)," RFC 7515, may 2015. [Online]. Available: https://tools.ietf.org/html/rfc7515

[20] M. Jones and J. Hildebrand, "JSON Web Encryption (JWE)," RFC 7516, may 2015. [Online]. Available: https://tools.ietf.org/html/rfc7516

[21] M. Jones, "JSON Web Key (JWK)," RFC 7517, may 2015. [Online]. Available: https://tools.ietf.org/html/rfc7517

[22] ——, "JSON Web Algorithms (JWA)," RFC 7518, may 2015. [Online]. Available: https://tools.ietf.org/html/rfc7518

[23] M. Miller, "Examples of Protecting Content Using JSON Object Signing and Encryption (JOSE)," RFC 7520, may 2015. [Online]. Available: https://tools.ietf.org/html/rfc7520

[24] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)," RFC 7049, october 2013. [Online]. Available: https://tools.ietf.org/html/rfc7049

[25] E. Wahlstroem, M. Jones, and H. Tschofenig, "CBOR Web Token (CWT)," draft-wahlstroem-ace-cbor-web-token-00, december 2015. [Online]. Available: https://tools.ietf.org/html/draft-wahlstroem-ace-cbor-web-token-00

[26] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings," RFC 4648, october 2006. [Online]. Available: https://tools.ietf.org/html/rfc4648

[27] D. J. Bernstein. (2016) The Sodium crypto library (libsodium). [Online]. Available: https://download.libsodium.org/doc/

[28] (2016) Experiment'Haal: an AAL Living'Lab. IHSEV Telecom Bretagne. [Online]. Available: https://www.telecom-bretagne.eu/recherche/plates-formes_technologiques/experiment-haal/

[29] (2016) The Precious Project - PREventive Care Infrastructure based On Ubiquitous Sensoring. [Online]. Available: http://www.thepreciousproject.eu/