A brief introduction to xAAL v0.5-draft rev 1

Christophe Lohr   Jérôme Kerdreux   Philippe Tanguy

August 24, 2015

Abstract

This paper gives a brief description of security elements of xAAL such defined by the IHSEV/HAAL team of Telecom Bretagne and its status in 2015.

Be aware that this is an ongoing work.

Please read specifications of xAAL version v0.4 first. The current document has to be been seen as an adding.

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. 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. In short, 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.

On previous version of xAAL, messages are 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 on alarms systems. However, new solutions began to embed cryptographic elements; sometime poor (RC4, rolling codes, etc.), sometime rather strong (RSA, DSA, AES, etc.). As xAAL acts as a bridge between those systems, xAAL must not introduce weakness. The current document describes experiments and choices leading to the security elements proposed for the new version of xAAL.

The xAAL system is detailed in specification v0.4. The current document describes version v0.5 which provides the ciphering of communications of the version v0.4. The version v0.5 does not provide new functionality to xAAL except communications ciphering. People interested by security are invited to consider version v0.5. If not, please consider version v0.4.

This document is organized as follow. 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. The section 4 concludes this paper.

2 Preliminary studies

2.1 Signature of xAAL v0.4

Version v0.4 of xAAL proposes authenticating messages with a signature and a private shared key. For this purpose, messages have three fields:

- `cipher`: The name of the ciphering mechanism used to sign the message;
- `signature`: The signature of the message by the selected signing algorithms;
- `timestamp`: The date of the message in seconds since 1970-01-01 00:00:00 UTC. This field aims to avoid replay attacks.

2.1.1 A private shared key

There are several types of security keys:

- *A private shared key*: Each homeowner choose 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). 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 principle of *a distributed system* for xAAL. Also, keep in mind that in such systems, leaks of the signatures may happen: the signatures of the (intermediate) authorities can be revelated and reused to sign illegitimate keys. So, revocation protocols has 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). *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 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 friends devices...

As written in specifications of version v0.4, using a pre-shared key seems acceptable in the context of a home network: it is the choice of WiFi WPA/WPA2 Personal and of 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.

2.1.2 Multiple signing algorithms

The specification of xAAL v0.4 allows multiple signing algorithms. An xAAL device may support several signing algorithms. The `cipher` field on an xAAL message tells the signing algorithm choosed to sign it.

Supporting multiple cryptographic algorithms 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.

Well, 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 can't be a preliminary negotiation 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 in the same time on the same bus, there can be several separated conversations on 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, I won't 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 is two proxies doing the translation in the two ways. So, forget it. Definitively.

As a consequence, it has been decided that xAAL version v0.5 will support one and only one cryptographic algorithm. (See bellows to know which one is selected; the important point here it that there is only cryptographic algorithm.)

2.1.3 A "signature" field inside the message

Version v0.4 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 write 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 different 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 wich the signature has been computed. The receiver has no chance to retrieve the original way in which the message was serialized. It can't check the signature.

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

As a consequence, it has been decided that xAAL version v0.5 will put the cryptographic elements outside of the message to be secured.

2.1.4 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 homeowner stops its alarm when he arrives at home. Then, when the homeowner 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 can't 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 check 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 to ask all devices to keep in memory all past values used for the nonce, and to compare any incoming message to this list. Well, this is not realistic.

Another strategy is to use a counter, a kind of message-id. Each device listens messages on the bus, record the last used message-id. At 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 use 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 xAAL version v0.4 to use the time that passes for this counter. The nonce is a timestamp, the number of seconds since the *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 to old (outside this acceptance window) are rejected. From the hardware point of view, having a precision of few seconds (or few minutes) is ok. Requiring a stronger synchronization may becomes very hard. We want a good security but at an acceptable price. Therefore, a replay attack is still feasible within this acceptance window. We gauged this risk and estimated that this compromise is acceptable. To be honest, the replay attacks have a rather limited interest: the attacker can not send all messages he wants, he can just blindly inject some messages that legitimate users has 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 homeowner is still in the place (and that the alarm is already off)...

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), 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 already do network). The plain old *Time Protocol* (RFC868) is no more in use. Nowadays, the Network Time Protocol (NTP, RFC 5905) 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. The RFC 5905 also propose 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 consumes CPU resource with NTP. Another more lightweight strategy is the *ntpdate* approach: one just ask once the time is it to an NTP server and set it to its internal clock. Note that this is also the strategy of the NTP implementation in Arduino. (This is more or less the way the plain old Time Protocol do the job.)

Note that there are very few attack on the NTP protocol. Sometimes NTP servers are 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 it is. This also a reason why there won't be an xAAL device that gives the time to other devices. First, there are little chance 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 won't 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 days cycles. The accuracy is of 1 second per day, 15 seconds per month, 1 minute per year. This is good enough for xAAL.

As a consequence, it has been decided that xAAL version v0.5, as for xAAL version v0.4, will use a timestap as a nonce, with an acceptance window of few seconds or few minutes.

2.2 Ciphering of xAAL v0.4

Sign 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 has to be ciphered.

The specification of xAAL version v0.4 does not propose ciphering. However, some tests and implementations have been driven, with little arrangements of the format of messages.

2.2.1 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.

According to experts, it is at least as much strong than others (e.g. AES). According to authors, it is much more faster, requiring less memory, less CPU. (This works on an Intel 8051!)

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.

We are convinced that this is a right choice.

2.2.2 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 others bits are just zeros. So the unused bits can be used to code something else that varies, for instance microseconds. As a result on can build a nonce that change 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 *millennum bug* (i.e., sunday February 7 2106 at 07:28:15 UTC): 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 of the precision of their clock (that could 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 RFC7539 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's best to avoid them. According to the way xAAL build the nonce, there could be collisions between packets before and after February 2106 (don't worry!), 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.

2.2.3 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 before decrypting.

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 filed is also protected by the key. An attacker can't 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 filed `version` has also been moved as public data to ease devices to quickly filter messages. In the test described here, it was also covered by the signature. It is perhaps unnecessary. For now, we can't find a scenario of an attack based on tricking the version field.

2.2.4 A binary security layer

To experiment ciphering of xAAL v0.4, 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 put the encrypted message side to those JSON fields, there is no way to really know where the encrypted part starts or ends. The receiver can't 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 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* sice 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 build 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 test 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) 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.

So, the idea is there, but the format has to be improved.

2.3 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:

- RFC 7165: *Use Cases and Requirements for JSON Object Signing and Encryption (JOSE).* This gives the frame for the JOSE working group.
- RFC 7515: *JSON Web Signature (JWS).* This specifies the message format and cryptographic mechanisms used to sign and authenticate messages.
- RFC 7516: *JSON Web Encryption (JWE).* This specifies the message format and cryptographic mechanisms used to cipher messages.
- RFC 7517: *JSON Web Key (JWK).* This specifies the format of keys.
- RFC 7518: *JSON Web Algorithms (JWA).* This describes registers where cryptographic algorithms are recorded. (They are not recorded inside RFCs but in registers pointed by RFCs. This allows more flexibility.)
- RFC 7520: *Examples of Protecting Content using JavaScript Object Signing and Encryption (JOSE).* 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 of cryptographic mechanisms are possible. Surprisingly, Poly1305/Chacha20 is not in the list at the time this document is written (2015).

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 require 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 is 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 good candidate.

2.4 Concise Binary Object Representation (CBOR)

The RFC 7049 *Concise Binary Object Representation (CBOR)* specifies a format to serialize data in a binary way. In short, CBOR does exactly the same things than 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 is 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 sequence, but there is no tag to express when a CBOR serialization ends (and that something else is starting after).

Today there are some programming libraries (or piece of code) for CBOR. CBOR seems do become more and more used, and more specifically in the area of Internet of Thinks (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.

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.

2.5 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 cpryptographic algorithm;
- A binary nonce build 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 Security in xAAL v0.5

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.) Others values should be rejected.
- `targets`: A string build 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 build as the base64 encoding of the ciphered *application layer*.

The *application layer* (which is embedded inside the *security layer*) is build as described in version 0.4 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 1 and 2 give an example of an xAAL message (the security layer) with its decoded payload (the application layer).

```
{
  "version": "0.5",
  "targets": "[ \"174255ad-e2a1-41e9-bf23-8dbcdfc812d4\" ]",
  "timestamp": [ 1439824426, 467313 ],
  "payload": "8sbrvczRc5NpWRpG+vwro...mlkKeeSAUc5Dj9SKoe82="
}
```

Figure 1: Example of an XAAL message (*securitiy layer*)

```
{
  "header": {
    "source": "06b71935-c5bc-4b09-8f2b-dae3d3b8ce77",
    "devType": "thermometer.basic",
    "msgType": "reply",
    "action": "getAttributes"
  },
  "body": {
   "temperature": 33.0
 }
}
```

Figure 2: 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` filed 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 him 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

To be honest, 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

Well, simply use base64 to encode/decode the encrpyted data. 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 buid the cryptographic key from a passphrase

The Poly1305/Chacha20 algorithm use 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:

- function: `crypto_pwhash_scryptsalsa208sha256()`
- for the salt: a buffer of zeros
- for the *opslimit*: `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_INTERACTIVE` (512k cycles)
- for the *memlimit*: `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_INTERACTIVE` (16 Mbytes)

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 version on the same bus. However, this leads to several separate communication channels in parallel. The devices with a given security key or a given xAAL version can not talk to the devices with another key or version. They will systematically reject plenty of packets. This is inefficient.

If several key or xAAL version are needed, it is recommended to use different xAAL buses.

4 Conclusion

This document specifies xAAL version v0.5. This new version provides security of communications. Functionalities remain the same than in xAAL version v0.4.

This document tells experiments and studies we drove and explains choices and motivations we done to design this new security layer.

Main points are:

- A pre-shared private key;
- The use of the Poly1305/Chacha20 algorithm;
- A timestamp to avoid replay attack, and an acceptance window on the age of messages;
- A JSON format for this security layer.