Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 6, 2014

H. Tschofenig
Nokia Siemens Networks
S.S. Kumar
S. Keoh
Philips Research

July 5, 2013

A Hitchhiker's Guide to the (Datagram) Transport Layer Security Protocol for Smart Objects and Constrained Node Networks

draft-tschofenig-lwig-tls-minimal-03

Abstract

Transport Layer Security (TLS) is a widely used security protocol that offers communication security services at the transport layer. The initial design of TLS was focused on the protection of applications running on top of the Transmission Control Protocol (TCP), and was a good match for securing the Hypertext Transfer Protocol (HTTP). Subsequent standardization efforts lead to the publication of the Datagram Transport Layer Security (DTLS) protocol, which allows the re-use of the TLS security functionality and the payloads to be exchanged on top of the User Datagram Protocol (UDP).

With the work on the Constrained Application Protocol (CoAP), as a specialized web transfer protocol for use with constrained nodes and constrained networks, DTLS is a preferred communication security protocol.

Smart objects are constrained in various ways (e.g., CPU, memory, power consumption) and these limitations may impose restrictions on the protocol stack such a device runs. This document only looks at the security part of that protocol stacks and the ability to customize TLS/DTLS. To offer input for implementers and system architects this document illustrates the costs and benefits of various TLS/DTLS features for use with smart objects and constraint node networks.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119]

Table of Contents

1. Introduction

The IETF published three versions of Transport Layer Security: TLS Version 1.0 [RFC2246], TLS Version 1.1 [RFC4346], and TLS Version 1.2 [RFC5246]. Section 1.1 of [RFC4346] explains the differences between Version 1.0 and Version 1.1; those are small security improvements, including the usage of an explicit initialization vector to protect against cipher-block-chaining attacks, which all have little to no impact on smart object implementations. Section 1.2 of [RFC5246] describes the differences between Version 1.1 and Version 1.2. TLS 1.2 introduces a couple of major changes with impact to size of an implementation. In particular, prior TLS versions hard-coded the MD5/SHA-1 combination in the pseudo-random function (PRF). As a consequence, any TLS Version 1.0 and Version 1.1 implementation had to have MD5 and SHA-1 code even if the remaining cryptographic primitives used other algorithms. With TLS Version 1.2 the two had been replaced with cipher-suite-specified PRFs. In addition, the TLS extensions definition [RFC6066] and various AES ciphersuites [RFC3268] got merged into the TLS Version 1.2 specification.

All three TLS specifications list a mandatory-to-implement ciphersuite: for TLS Version 1.0 this was TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA, for TLS Version 1.1 it was TLS_RSA_WITH_3DES_EDE_CBC_SHA, and for TLS Version 1.2 it is TLS_RSA_WITH_AES_128_CBC_SHA. There is, however, an important qualification to these compliance statements, namely that they are only valid in the absence of an application profile standard specifying otherwise. The smart object environment may, for example, represent a situation for such an application profile which defines a cryptosuite that reduces memory and computation requirements without sacrificing security.

All TLS versions offer a separation between authentication and key exchange, and bulk data protection. The former is more costly performance- and message-wise. The details of the authentication and key exchange, using the TLS Handshake, vary with the chosen ciphersuite. With new ciphersuites the TLS feature-set can easily be enhanced, in case the already large collection of ciphersuites, see [TLS-IANA], does not match the requirements.

Once the TLS Handshake has been successfully completed the necessary keying material and parameters are setup for usage with the TLS Record Layer, which is responsible for bulk data protection. The provided security of the TLS Record Layer depends also, but not only, on the chosen ciphersuite algorithms; NULL encryption ciphersuites, like those specified in RFC 4785 [RFC4785], offer only integrity-without confidentiality-protection. Example ciphersuites for the TLS Record Layer are RC4 with SHA-1, AES-128 with SHA-1, AES-256 with SHA-1, RC4 with SHA-1, RC4 with MD5 It is worth mentioning that TLS may also be used without the TLS Record Layer. This has, for example, been exercised with the work on the framework for establishing a Secure Real-time Transport Protocol (SRTP) security context using the Datagram Transport Layer Security (DTLS [RFC4347]) protocol (DTLS-SRTP [RFC5763]).

It is fair to say that TLS and consequently DTLS offers a fair degree of flexibility. What specific security features of TLS are required for a specific smart object application scenario depends on various factors, including the communication architecture and the threats that shall be mitigated.

The goal of this document is to provide guidance on how to use existing DTLS/TLS extensions for smart objects and to explain their costs in terms of code size, computational effort, communication overhead, and (maybe) energy consumption. The document does not try to be exhaustive, as the list of TLS/DTLS extensions is enhanced on a frequent basis. Instead we focus on extensions that those working in the smart object community often found valuable in their practical experience. A non-goal is to propose new extensions to DTLS/TLS to provide even better performance characteristics in specific environments.

2. Overview

A security solution to be deployed is strongly influenced by the communication relationships [RFC4101] between the entities. Having a good understanding of these relationships will be essential to define the threats and decide on how to customize the security solution. Some of these considerations are described in [I-D.gilger-smart-object-security-workshop].

Consider the following scenario where a smart-meter transmits its energy readings to other parties. The public utility has to ensure that the meter readings it obtained can be attributed to a specific meter in a household. It is simply not acceptable for public utility to have any meter readings tampered in transit or by a rogue endpoint (particularly if the attack leads to a disadvantage, for example financial loss, for the utility). Users in a household may want to

ensure that only certain authorized parties are able to read their meter; privacy concerns come to mind.

In this example, a smart-meter may only ever need to talk to servers of a specific utility or even only to a single pre-configured server. Clearly, some information has to be pre-provisioned into the device to ensure the desired behavior to talk only to selected servers. The meter may come pre-configured with the domain name and certificate belonging to the utility. The device may, however, also be configured to accept one or multiple server certificates. It may even be pre-provisioned with the server's raw public key, or a shared secret instead of relying on certificates.

Lowering the flexibility decreases the implementation overhead. If shared secrets are used with TLS-PSK [RFC4279] or raw public keys are used with TLS [I-D.ietf-tls-oob-pubkey], fewer lines of code are needed than employing X.509 certificate, as will be explained later in this document. A decision for constraining the client-side TLS implementation, for example by offering only a single ciphersuite, has to be made in awareness of what functionality will be available on the TLS server-side. In certain communication environments it may be easy to influence both communication partners while in other cases the existing deployment needs to be taken into consideration.

To illustrate another example, consider an Internet radio, which allows a user to connect to available radio stations. A device like this will be more demanding than an IP-enabled weighing scale that only connects to the single web server offered by the device manufacturer. A threat assessment may even lead to the conclusion that TLS support is not necessary at all in this particular case.

Consider the following extension to our earlier scenario where the smart-meter is attached to a home WLAN network and the inter-working with WLAN security mechanisms need to be taken care of. On top of the link layer authentication, a transport layer or application layer security mechanism needs to be implemented. Quite likely the security mechanisms will be different due to the different credential requirements. While there is a possibility for re-use of cryptographic libraries (such as the SHA-1, MD5, or HMAC) the overall code footprint will very likely be larger.

Furthermore, security technology that will be deployed by end-user consumer market products and large enterprise customer products will need to be customized completely different. While the security building blocks may be reused, there is certainly a big difference between in terms of the architecture, the threats and effort that will be spent securing the system.

3. Design Decisions

To evaluate the required TLS functionality a couple of high level design decisions have to be made:

- What type of protection for the data traffic is required? Is confidentiality protection in addition to integrity protection required? Many TLS ciphersuites also provide a variant for NULL encryption [RFC4279]. If confidentiality protection is required, a carefully chosen set of algorithms may have a positive impact on the code size. Re-use of crypto-libraries (within TLS but also among the entire protocol stack) will also help to reduce the overall code size.

- What functionality is available in hardware? For example, certain hardware platforms offer support for a random number generator as well as cryptographic algorithms (e.g., AES). These functions can be re-used and allow to reduce the amount of required code. Using hardware support not only reduces the computation time but can also save energy due to the optimized implementation.

- What credentials for client and server authentication are required: passwords, pre-shared secrets, certificates, raw public keys (or a mixture of them)? Is mutual authentication required? Is X509 certificate handling necessary? If not, then the ASN.1 library as well as the certificate parsing and processing can be omitted. If pre-shared secrets are used then the big integer implementation can be omitted.

- What TLS version and what TLS features, such as session resumption, can or have to be used? In the case of DTLS, generic fragmentation and reordering requires large buffers to reassemble the messages, which might be too heavy for some devices.

- Is it possible to design only the client-side TLS stack, or necessary to provide the server-side implementation as well? Handshake messages sent are different sizes for the client and server which creates energy consumption differences (due to the fact that more power is spent during transmission than while receiving data in wireless devices).

- Which side will be more powerful? Resource-constrained sensor nodes running CoAPS might be server only, while nodes running HTTPS are most like clients only that post their information to a normal Web server. The constrained side will most likely only implement a single ciphersuite. Flexibility is given to a more powerful counterpart that supports many different ciphersuite for various connected devices.

- Is it possible to hardwire credentials into the code rather than loading them from storage? If so, then no file handling or parsing of the credentials is needed and the credentials are already available in a form that they can be used within the TLS implementation.

4. Performance Numbers

In this section we summarize performance measurements available from certain implementation experiences. This section is not supposed to be exhaustive as we do not have all measurements available. The performances are grouped according to extensions (TLS-PSK, raw-public key and certificate based) and further grouped by performance measures (memory, code size, communication overhead, etc.). Where possible we extract the different building blocks found in TLS and present their performance measures individually.

4.1. Pre-Shared Key (PSK) based DTLS implementation

This section provides performance numbers for a prototype implementation of DTLS-PSK described in [I-D.keoh-lwig-dtls-iot] and evaluates the memory and communication overheads.

4.1.1. Prototype Environment

The prototype is written in C and runs as an application on Contiki OS 2.5 [Dunkels-contiki], an event-driven open source operating system for constrained devices. They were tested in the Cooja simulator and then ported to run on Redbee Econotag hardware, which features a 32-bit CPU, 128 KB of ROM, 128 KB of RAM, and an IEEE 802.15.4 enabled radio with an AES hardware coprocessor. The prototype comprises all necessary functionality to adapt to the roles as a domain manager or a joining device.

The prototype is based on the "TinyDTLS" [Bergmann-Tinydtls] library and includes most of the extensions and the adaptation as follows:

(1) The cookie mechanism was disabled in order to fit messages to the available packet sizes and hence reducing the total number of messages when performing the DTLS handshake.

(2) Separate delivery was used instead of flight grouping of messages and redesigned the retransmission mechanism accordingly.

(3) The "TinyDTLS" AES-CCM module was modified to use the AES hardware coprocessor.

The following subsections further analyze the memory and

communication overhead of the solution.

4.1.2. Code size and Memory Consumption

Table 1 presents the codesize and memory consumption of the prototype differentiating (i) the state machine for the handshake, (ii) the cryptographic primitives, and (iii) the DTLS record layer mechanism.

The use of DTLS appears to incur large memory footprint both in ROM and RAM for two reasons. First, DTLS handshake defines many message types and this adds more complexity to its corresponding state machine. The logic for message re-ordering and retransmission also contributes to the complexity of the DTLS state machine. Second, DTLS uses SHA2-based crypto suites which is not available from the hardware crypto co-processor.

	DTLS	
	ROM	RAM
State Machine	8.15	1.9
Cryptography	3.3	1.5
Key Management	1.0	0.0
DTLS Record Layer	3.7	0.5
TOTAL	16.15	3.9

Table 1: Memory Requirements in KB

4.1.3. Communication Overhead

The communication overhead is evaluated in this section. In particular, the message overhead and the number of exchanged bytes under ideal condition without any packet loss is examined.

Table 2 summarizes the required number of round trips, number of messages and the total exchanged bytes for the DTLS-based handshake carried out in ideal conditions, i.e., in a network without packet losses. DTLS handshake is considered complex as it involves the exchange of 12 messages to complete the handshake. Further, DTLS runs on top the transport layer, i.e., UDP, and hence this directly increases the overhead due to lower layer per-packet protocol headers.

	DTLS
No. of Message	12
No. of round trips	4
802.15.4 headers	168B
6LowPAN headers	480B
UDP headers	96B
Application	487B
TOTAL	1231B

Table 2: Communication overhead for Network Access and Multicast Key Management

4.1.4. Message Delay, Success Rate and Bandwidth

Section 5.3 provided an evaluation of the protocol in an ideal condition, thus establishing the the baseline protocol overhead. The prototype was further examined and simulated the protocol behavior by tuning the packet loss ratio. In particular, the impact of packet loss on message delay, success rate and number of messages exchanged in the handshake were examined.

Figure 4 shows the percentage of successful handshakes as a function of timeouts and packet loss ratios. As expected, a higher packet loss ratio and smaller timeout (15s timeout) result in a failure probability of completing the DTLS handshake. When the packet loss ratio reaches 0.5, practically no DTLS handshake would be successful.

```
      100 |+
P         | +
E    80 |  ++
R         |    ++
C    60 |      +
E         |       +
N    40 |        +
T         |         ++
A    20 |            +
G         |             +++++
E     0 +------------------+++++++++-->
         0 0.1 0.2 0.3 0.4 0.5

          packet loss ratio (15s timeout)
```

Figure 1: Average % of successful handshakes

Delays in network access and communication are intolerable since they lead to higher resource consumption. As the solution relies on PSK, the handshake protocol only incurs a short delay of a few milliseconds when there is no packet loss. However, as the packet loss ratio increases, the delay in completing the handshake becomes significant because loss packets must be retransmitted. Our implementation shows that with a packet loss ratio of 0.5, the the times to perform network access and multicast key management could take up to 24s.

Finally, another important criterion is the number of messages exchanged in the presence of packet loss. A successful handshake could incur up to 35 or more messages to be transmitted when the packet loss ratio reaches 0.5. This is mainly because the DTLS retransmission is complex and often requires re-sending multiple messages even when only a single message has been lost.

4.2. Certificate based and Raw-public key based TLS implementation

4.3.1. Prototype Environment

The following code was compiled under Ubuntu Linux using the -Os compiler flag setting for a 64-bit AMD machine using a modified version of the axTLS embedded SSL implementation.

4.3.2. Code size

For the cryptographic support functions these are the binary sizes:

Cryptographic functions	Code size
MD5	4,856 bytes
SHA1	2,432 bytes
HMAC	2,928 bytes
RSA	3,984 bytes
Big Integer Implementation	8,328 bytes
AES	7,096 bytes
RC4	1,496 bytes
Random Number Generator	4,840 bytes

Table 3: Code-size for cryptographic functions

The TLS library with certificate support consists of the following parts:

x509 related code: 2,776 bytes
The x509 related code provides functions to parse certificates, to

copy them into the program internal data structures and to perform certificate related processing functions, like certificate verification.

ASN1 Parser: 5,512 bytes
The ASN1 library contains the necessary code to parse ASN1 data.

Generic TLS Library: 15,928 bytes
This library is separated from the TLS client specific code to offer those functions that are common with the client and the server-side implementation. This includes code for the master secret generation, certificate validation and identity verification, computing the finished message, ciphersuite related functions, encrypting and decrypting data, sending and receiving TLS messages (e.g., finish message, alert messages, certificate message, session resumption).

TLS Client Library: 4,584 bytes
The TLS client-specific code includes functions that are only executed by the client based on the supported ciphersuites, such as establishing the connection with the TLS server, sending the ClientHello handshake message, parsing the ServerHello handshake message, processing the ServerHelloDone message, sending the ClientKeyExchange message, processing the CertificateRequest message.

OS Wrapper Functions: 2,776 bytes
These functions aim to make development easier (e.g., for failure handling with memory allocation and various header definitions) but are not absolutely necessary.

OpenSSL Wrapper Functions: 931 bytes
The OpenSSL API calls are familiar to many programmers and therefore these wrapper functions are provided to simplify application development. This library is also not absolutely necessary.

Certificate Processing Functions: 4,456 bytes
These functions provide the ability to load certificates from files (or to use a default key as a static data structure embedded during compile time), to parse them, and populate corresponding data structures.

4.3.2. Raw Public Key Implementation

Of course, the use of raw public keys does not only impact the code size but also the size of the exchanged messages. When using raw public keys (instead of certificates) the "certificate" size was reduced from 475 bytes to 163 bytes (using an RSA-based public key). Note that the SubjectPublicKeyInfo block does not only contain the

raw keys, namely the public exponent and the modulus, but also a small ASN.1 header preamble.

For the raw public key implementation the following components where needed (in addition to a subset of the cryptographic support functions):

Minimal ASN1 Parser: 3,232 bytes
The necessary support from the ASN1 library now only contains functions for parsing of the ASN1 components of the SubjectPublicKeyInfo block.

Generic TLS Library: 16,288 bytes
This size of this library was slightly enlarged since additional functionality for loading keys into the bigint data structure was added. On the other hand, code was removed that relates to certificate processing and functions to retrieve certificate related data (e.g., to fetch the X509 distinguished name or the subject alternative name).

TLS Client Library: 4,528 bytes
The TLS client-specific code now contains additional code for the raw public key support, for example in the ClientHello message. Most functions were left unmodified.

5. Summary and Conclusions

TLS/DTLS can be tailored to fit the needs of a specific deployment environment. This customization property allows it to be tailored to many use cases including smart objects. The communication model and the security goals will, however, ultimately decide the resulting code size; this is not only true for TLS but for every security solution.More flexibility and more features will ultimately translate to a bigger footprint.

There are, however, cases where the security goals ask for a security solution other than TLS. With the wide range of embedded applications it is impractical to design for a single security architecture or even a single communication architecture.

6. Security Considerations

This document discusses various design aspects for reducing the footprint of (D)TLS implementations. As such, it is entirely about security.

7. IANA Considerations

This document does not contain actions for IANA.

8. Acknowledgements

The authors would like to thank the participants of the Smart Object Security workshop, March 2012. The authors greatly acknowledge the prototyping and implementation efforts by Pedro Moreno-Sanchez and Francisco Vidal-Meca who worked as interns at Philips Research.

9. References

9.1. Normative References

[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, April 2006.

[SHA] National Institute of Standards and Technology, , "Secure Hash Standard", FIPS 180-2, Aug 2002.

9.2. Informative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", RFC 5763, May 2010.

[RFC4785] Blumenthal, U. and P. Goel, "Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)", RFC 4785, January 2007.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS)

Extensions: Extension Definitions", RFC 6066, January 2011.

[RFC3268] Chown, P., "Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)", RFC 3268, June 2002.

[RFC4101] Rescorla, E. and IAB, "Writing Protocol Models", RFC 4101, June 2005.

[RFC4279] Eronen, P., Ed., and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, December 2005.

[TLS-IANA] IANA, "Transport Layer Security (TLS) Parameters: http://www.iana.org/assignments/tls-parameters/tls-parameters.xml", Oct 2012.

[I-D.ietf-tls-oob-pubkey] Wouters, P., Tschofenig, H., Gilmore, J., Weiler, S., and T. Kivinen, "Out-of-Band Public Key Validation for Transport Layer Security (TLS)", draft-ietf-tls-oob-pubkey-07 (work in progress) February 2013.

[I-D.keoh-lwig-dtls-iot] Keoh, S., Kumar, S., and Garcia-Morchon, O., "Securing the IP-based Internet of Things with DTLS", draft-keoh-lwig-dtls-iot-01, February 2013.

[I-D.gilger-smart-object-security-workshop] Gilger, J., and Tschofenig, H., "Report from the 'Smart Object Security Workshop', March 23, 2012, Paris, France", draft-gilger-smart-object-security-workshop-01.txt, February 2013.

[Dunkels-Contiki] Dunkels, A., Gronvall, B., and Voigt, T. "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors", In Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, IEEE, 2004.

[Bergmann-Tinydtls] Bergmann, O. "TinyDTLS - A Basic DTLS Server Template", http://tinydtls.sourceforge.net, 2012.

Authors' Addresses

Hannes Tschofenig
Nokia Siemens Networks
Linnoitustie 6
Espoo,   02600
Finland

Phone: +358 (50) 4871445
Email: Hannes.Tschofenig@gmx.net
URI: http://www.tschofenig.priv.at

Sandeep S. Kumar
Philips Research
High Tech Campus 34
Eindhoven 5656 AE
NL

Email: sandeep.kumar@philips.com

Sye Loong Keoh
Philips Research
High Tech Campus 34
Eindhoven 5656 AE
NL

Email: sye.loong.keoh@philips.com