Smart Object Security: Considerations for Transport Layer Security Implementations

Hannes Tschofenig*
*Nokia Siemens Networks, Email: Hannes.Tschofenig@nsn.com

I. 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 Datagram Transport Layer Security (DTLS) [1], which added the User Datagram Protocol (UDP), and the Datagram Congestion Control Protocol (DCCP). The Stream Control Transmission Protocol (SCTP), as a more recent connection-oriented transport protocol, also benefits from TLS support.

To develop a security solution for an system that contains smart objects poses challenges since the range of constraints is broad: limited flash memory, energy restrictions, low bandwidth and lossy links, and various other limitations concerning manufacturing costs and peripheral support. With the broad range of requirements, which are sometimes even conflicting, it is difficult to offer a single security protocol that meets all needs. Consequently, a protocol that achieves the security goals of one environment may not live up to the goals of a second smart object deployment.

IETF security protocols are, by some, seen as inadequate for smart object deployments. Clearly, it is difficult to provide counter arguments in light of the wide range of possible smart object application scenarios and absent a list of security goals. To offer input for implementers and system architects the author therefore illustrates the code size implications of a selected TLS features.

The author believes that TLS is a good candidate for a large percentage of smart object deployments[1] when end-to-end communication security features are desired Not surprisingly, every feature has a certain cost and tradeoff decisions need to be made. The writeup also looks at some recently proposed extensions.

Position paper for the 'Smart Object Security' workshop, Friday, 23rd March 2012, Paris. This paper represents the views of the author in his role as an individual contributor to the Internet standards process. They do not reflect the consensus of the Internet Engineering Task Force (IETF) at large, of any IETF working group, or of the Internet Architecture Board (IAB). Referenced documents may, however, reflect IETF consensus. This paper was last updated on the $26th$ of February 2012. Last updated on March, $1st$.

II. TLS Introduction

The IETF published three versions of Transport Layer Security: TLS Version 1.0 [3], TLS Version 1.1 [4], and TLS Version 1.2 [5]. Section 1.1 of [4] 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 impact for the size of the code. Section 1.2 of [5] 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 version hardcoded the MD5/SHA-1 combination in the pseudorandom function (PRF). As a consequence, any TLS Version 1.0 and Version 1.1 implementation had to have MD5 and SHA1 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[2]. In addition, the TLS extensions definition [6] and various AES ciphersuites [7] 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 the absence of an application profile standard specifying otherwise. For smart object deployments it is likely that such an application profile standard will be provided that ensures interoperability with ciphersuites that deviate from the list above.

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. Figure 1 shows the exchange based on public key based client- and server-authentication while Figure 2 shows TLS session resumption, an abbreviated handshake that utilizes a prior TLS Handshake and already established keying material. These two exchanges[3] aim to

[1] [2] introduces a device classification where class 2 devices have 50 KByte RAM and 250 KByte flash memory. This appears to be suitable environment even without knowing further details about the remaining application protocol stack requirements.

[2] Since the code in this paper only supports TLS Version 1.0 and TLS Version 1.1 the code size benefits of this design decision has not yet been taken into consideration.

[3] Note that the representation of the TLS exchange may be unusual since the TLS specification differentiates between different types of messages, such as the TLS Handshake and the ChangeCipherSpec messages, and may also send individual protocol payloads as independent messages.

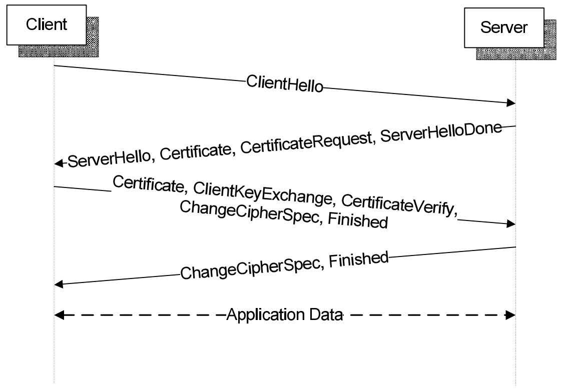
Client
Server
ClientHello
ServerHello, Certificate, CertificateRequest, ServerHelloDone
Certificate, ClientKeyExchange, CertificateVerify, ChangeCipherSpec, Finished
ChangeCipherSpec, Finished
Application Data

Fig. 1. TLS Exchange with Client and Server Authentication.

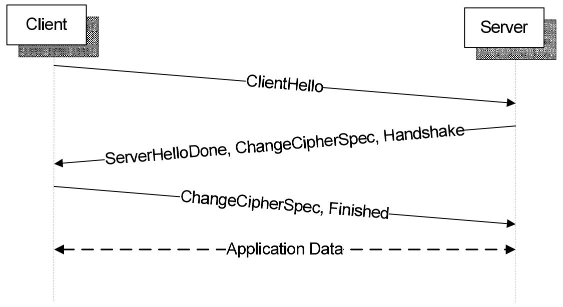
Client
Server
ClientHello
ServerHelloDone, ChangeCipherSpec, Handshake
ChangeCipherSpec, Finished
Application Data

Fig. 2. TLS Session Resumption.

illustrate only two main exchanges. With new ciphersuites the TLS feature-set can easily be enhanced, in case the already large collection of ciphersuites, see [8], 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 TLS Record Layer could be compared with the IPsec AH and IPsec ESP while the Handshake protocol can be compared with the Internet Key Exchange Version 2 (IKEv2). 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 [9], offer only integrity- without confidentiality-protection.

It is worth mentioning that TLS may 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) protocol (DTLS-SRTP, RFC 5763 [10]).

III. Code Availability

There are many reasons for reusing protocols developed by the Internet Engineering Task Force (IETF) and one reason relevant to this document is the widespread availability of high-quality open source code. Chances are good that there is already an open source TLS stack available for the smart object hardware platform. In such a case the integration effort is significant lower, which leads to faster time to market and lower development costs. This is not an irrelevant factor in the design of smart objects since costs are not only related to the hardware but also contributed by operating system and application development.

A compilation of available TLS implementations can be found at [11]. While many of these implementations provide a good starting point for a developer the axTLS project [12] already claims to offer a small code footprint. Beyond these implementations there is also open source code with EAP-TLS, which is used for network access authentication. Unlike the previous implementations this code first needs to be decoupled from the underlying Extensible Authentication Protocol (EAP) container. The WPA Supplicant [13] (for the client side) and the FreeRADIUS [14] (for the server side) contain EAP-TLS libraries that could be utilized – the WPA Supplicant already provides the support for a small internal crypto library that does not rely on external libraries, like OpenSSL.

The author had chosen axTLS and the remainder of the paper refers to that specific implementation. axTLS is a C-based implementation of TLS v1.0 and TLS v1.1 and offers the following features:

- Certificate support (X509v1, self-signed certificates, PKCS#8, PKCS#12 keys/certificates in DER/PEM format),
- Certificate verification and identity checks,
- Session resumption,
- Session renegotiation,
- Various TLS Record Layer cipher algorithms: RC4 with SHA1, AES128 with SHA1, AES256 with SHA1, RC4 with SHA1, RC4 with MD5

The axTLS implementation is focused on the support of ciphersuites that use RSA-based key transport and does not support TLS-PSK [4] or Diffie-Hellman based key exchanges. Digital Signature Algorithm (DSA) and Elliptic Curve Cryptography (ECC) based public key based algorithms are not supported either.

A useful property of the axTLS implementation is the integration of crypto-libraries. Therefore, no dependencies to external libraries exist.

IV. Communication Relationships

A security solution is strongly impacted by the communication relationships [15], which will have an impact on the code size. Having a good understanding of these relationships will be essential.

Consider the following scenario where a smart meter transmits its energy readings to other parties. The electricity company will want to make sure that the obtained meter readings can be attributed to a specific meter in a house hold. Most electricity companies will find it unacceptable to have meter readings modified in transit or by a rouge endpoint (particularly if the attack leads to a disadvantage, for example financial loss, for the electricity provider). Users in a house hold may want to ensure that only certain parties get to see the meter readings; privacy concerns come to mind.

In this example a sensor may need to only ever talk to servers of a specific electricity company or even only to a single pre-configured server[4] 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 be configured with the domain name used by the electricity company and with the root certificate that is used by the electricity company to sign certificates for their servers. 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. TLS-PSK [16], in case of shared secret usage, or raw public keys used with TLS [17] require fewer lines of code than x509 certificate usage, as it 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 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 others cases the existing deployment needs to be taken into consideration.

To illustrate another example, consider an Internet radio who allows a user to connect to available radio stations will be more constrained than an IP-enabled scale that connects only to the Web server offered by the manufacturer of that device. For the Internet radio case a threat assessment may even lead to the conclusion that TLS support is not necessary at all.

Consider the following extension to our earlier scenario where the meter is attached to a home WLAN network and the interworking 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.

V. Code Size

Smart objects are constrained in different ways; flash memory is only one of many constraints. The choice of protocol mechanisms will be impacted by whatever constraint is getting most attention from an engineer. Some design goals are also contradicting with each other. In this paper we focus on code size optimizations. Tradeoffs will have to be made by reducing flexibility and functionality. Consider the following simple example: TLS session resumption leads to an improvement in the initial handshake but requires additional code. The following tables provide information about the code size of various security functions. The code was compiled under Ubuntu Linux using the -Os compiler flag setting for a 64-bit AMD machine. The same environment had been used for all other code compilations in this paper.

TABLE I
Binary Code Size for Cryptography Support Functions

Library	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

The library implementing the random number generator (RNG) *crypto_misc.c* allows OS support to be re-used but for the purpose of this illustration it had been disabled and therefore the RNG implementation is purely in software. A hardware platform that provides RNG support may likely lead to a smaller footprint. There are also other support functions that come with *crypto_misc.c* but those had been excluded with compile-time flags.

The code in Table II includes support for session resumption as well as the necessary functions for certificate validation and identity checking.

Finally, there is the implementation application that opens a TCP connection to a predefined server address, performs the TLS handshake (with server-side authentication only, RSA encrypted key transport) and establishes the TLS record layer with the RC4-SHA1 ciphersuite. Without the above-mentioned code statically linked to the TLS implementation the object code is 3,816 bytes large.

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

1) 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. If confidentiality protection is demanded a carefully chosen set of algorithms may have a positive impact on the code size. For example, the RC4 stream cipher code size is 1,496 bytes compared to 7,096 bytes for AES usage. Re-use of crypto-libraries (within TLS but also among the entire protocol stack) will also help to reduce the overall code size.
2) 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.
3) What credentials for client and server authentication are required: passwords, pre-shared secrets, certificates, raw public keys (or a mixture of them)?
4) What TLS version and what TLS features, such as session resumption, can or have to be used?

This paper focuses on code size requirements rather than on optimization of RAM, energy consumption, or over-the-wire communication overhead. As detailed in Table III, the

[4] The server the meter needs to interact with may change over time with the help of software update or configuration changes. These changes to the software of the meter may then also lead to newly provisioned security credentials, such as certificates.

TABLE II

Binary Code Size for TLS-specific Code

Library Name	Code Size	Description
x509	2,776 bytes	The x509 related code (*x509.c*) 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 (*asn1.c*) contains the necessary code to parse ASN1 data.
Generic TLS Library	15,928 bytes	This library (*tls1.c*) is separated from the TLS client specific code (*tls1_clnt.c*) 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 (*tls1_clnt.c*) 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	The functions defined in *os_port.c* 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 (*openssl.c*) is also not absolutely necessary.
Certificate Processing Functions	4,456 bytes	These functions defined in *loader.c* 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.

TABLE III

Binary Code Size for Raw Key Support in TLS

Library Name	Code Size	Description
ASN1 Parser	3,232 bytes	The necessary support from the ASN1 library (*asn1.c*) 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 (*tls1.c*) was slightly enlarged since additional functionality for loading keys into the bigint data structure, which was previously found in *loader.c* and in *x509.c* is now included in this file. 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 (*tls1_clnt.c*) nows contains additional code for the raw public key support, for example in the ClientHello msg. Most functions were left unmodified.

author modified the code of Table II to provide support for raw public keys, as described in [17]. By using [17] the TLS client adds an extension of type ”cert_type” to the extended client hello message to indicate support for raw public keys. If the TLS server supports this extension it includes only the SubjectPublicKeyInfo part of the certificate (rather than the entire certificate) in the Certificate message. This leads to a reduction of the data passed over to the client; in our example with the default certificate provided by the axTLS implementation 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 ASN1 header preamble.

Various code optimizations are possible when raw public keys are used instead of certificates. Table III shows the impact to the code size.

Note that the functionality provided in *loader.c* was reduced to the bare minimum by removing the ability to load different types of certificates from the filesystem. The remaining code was together with the x509 functionality incorporated into *tls1.c*. The author believes that further reductions in code size are accomplishable at the expense of flexibility, protocol performance, and application developer convenience. For example, the code above still offers session resumption support - a feature that consumes a small amount of code space while significantly improving the handshake performance. Also the usage of TLS session resumption without server-side state, see RFC 5077 [18], is a reasonable option for devices that communicate infrequently. Code for RFC 5077 is not included in this library.

Of course, the transition to raw public keys does not change the code size of the core crypto functions, such as MD5, SHA-1, and RC4. However, the RSA code size was reduced to 3,544 bytes since the function for decryption is not needed in our setting.

VI. Conclusion

TLS is a good example of a wildly[5] success security protocol that can be tailored to fit the needs of a specific deployment environment. This customization property offers the basis for a relatively small code footprint. The communication model and the security goals will, however, ultimately decide about the resulting code size; this is not only true for TLS but for every security solution.

More flexibility and more features will translate to a bigger footprint. Generic complains about the large size of TLS stacks are not useful and should be accompanied by a description of the assumed functionality. To support the author’s opinion this position paper provides information about the amount of required code for various functions and considers most recent work from the IETF TLS working group for the support of

[5] The IAB published a document, RFC 5218 [19], on the success criteria for protocols. A ”wildly successful” protocol far exceeds its original goals, in terms of purpose (being used in scenarios far beyond the initial design), in terms of scale (being deployed on a scale much greater than originally envisaged), or both.

raw public keys[6].

The author is convinced that TLS is a suitable security protocol (with the standardized extensions) for usage in many smart object deployments and only minor extensions, as currently being developed in the IETF TLS working group, are needed to support an even large set of use cases. There are, however, cases where the security goals ask for a security solution other than TLS. With the wide range of smart object applications it is impractical to design for a single security architecture or even a single communication architecture.

As a next step the author suggests to collect the community experience on TLS about the implementation, system design and deployment decisions that impact the code footprint, size and number of exchanged messages, RAM usage, and energy consumption. This information will be valuable for those working on smart object security.

References

[1] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security," Apr. 2006, RFC 4347, Request For Comments.

[2] C. Bormann, "Guidance for Light-Weight Implementations of the Internet Protocol Suite," Jan. 2012, IETF draft (work in progress), draft-bormann-lwig-guidance-01.txt.

[3] T. Dierks and C. Allen, "The TLS Protocol - Version 1.0," Jan. 1999, RFC 2246, Request For Comments.

[4] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol - Version 1.1," Apr. 2006, RFC 4346, Request For Comments.

[5] ——, "The Transport Layer Security (TLS) Protocol - Version 1.2," Aug. 2008, RFC 5246, Request For Comments.

[6] D. Eastlake, "Transport Layer Security (TLS) Extensions: Extension Definitions," Jan. 2011, RFC 6066, Request For Comments.

[7] P. Chown, "Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)," Jun. 2002, RFC 3268, Request For Comments.

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

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

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

[11] Wikipedia, "Comparison of TLS Implementations," Feb. 2012, http://en.wikipedia.org/wiki/Comparison_of_TLS_Implementations.

[12] C. Rich, "axTLS embedded SSL," Feb. 2012, http://axtls.sourceforge.net/.

[13] J. Malinen, "Linux WPA/WPA2/IEEE 802.1X Supplicant," Feb. 2012, http://hostap.epitest.fi/wpa_supplicant/.

[14] A. DeKok, "The FreeRADIUS Project," Feb. 2012, http://freeradius.org/.

[15] E. Rescorla, "Writing Protocol Models," Jun. 2005, RFC 4101, Request For Comments.

[16] P. Eronen and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)," Dec. 2005, RFC 4279, Request For Comments.

[17] P. Wouters, J. Gilmore, S. Weiler, T. Kivinen, and H. Tschofenig, "TLS Out-of-Band Public Key Validation," Jan. 2012, IETF draft (work in progress), draft-ietf-tls-oob-pubkey-01.txt.

[18] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State," Jan. 2008, RFC 5077, Request For Comments.

[19] D. Thaler and B. Aboba, "What Makes for a Successful Protocol?" Jul. 2008, RFC 5218, Request For Comments.

[6] The author will do further work to reduce the code size (for example by using the new TLS 1.2 PRF) and to cross-compile the code to an ARM-based platform for better comparison with investigations of other teams.