Smart Objects 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 uses security protocol that offers a range of security services on top of the transport layer. While the initial design had of TLS had focused on the protection of applications running on top of the Transmission Control Protocol (TCP), and used with the Hypertext Transfer Protocol (HTTP) in particular, subsequent standardized extensions also introduced support for User Datagram Protocol (UDP), Datagram Congestion Control Protocol (DCCP), and Stream Control Transmission Protocol (SCTP). The design of TLS is in the meanwhile classical for many security protocols - it separates the authentication and key exchange phase from the actual protection of the application data exchange. The same design pattern can also be found in many other security protocols, including the Internet Key Exchange protocol version 2 (IKEv2) and the IP security protocol suite.

TLS is often seen as an inadequate choice for smart object deployments due to its perceived complexity. Such a statement is in general hard to justify absent a threat analysis and a list of security goals that need to be accomplished for a specific architecture. On top of this observation there is the challenge that smart objects may be constrained in various directions, such as flash and volatile memory limitations, energy and communication restrictions. An implementation that aims to fulfill the security goals in one specific environment may need to take a very different direction compared to one that targets a different environment.

To offer input for implementers and system architects the author therefore illustrates the cost of various TLS features based on a selected lightweight implementation. Not surprisingly, every feature has a certain cost and tradeoff decisions need to be made. The writeup also looks at some recently proposed extensions.

II. TLS Introduction

The IETF published three versions of Transport Layer Security: TLS Version 1.0 [1], TLS Version 1.1 [2], and TLS Version 1.2 [3]. Section 1.1 of [2] 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 [3] 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[1]. In addition, the TLS extensions definition and various ciphersuites were merged into the TLS Version 1.2 specification. Those were previously published in separate RFCs, e.g., [4] and [5].

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, which is typically costly from a performance and message point of view, and the bulk data protection. The details of the authentication and key exchange, using the TLS Handshake, vary with the chosen ciphersuite. 1 shows the exchange based on public key based client- and server-authentication while 2 shows TLS session resumption, an abbreviated handshake that utilizes a prior TLS Handshake and already established keying material. These two exchanges[2]

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

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. The author is a member of the IAB and has contributed to the standardization efforts discussed in this document. Paper was submitted on the 24th of February 2012

[1]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.

[2]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.

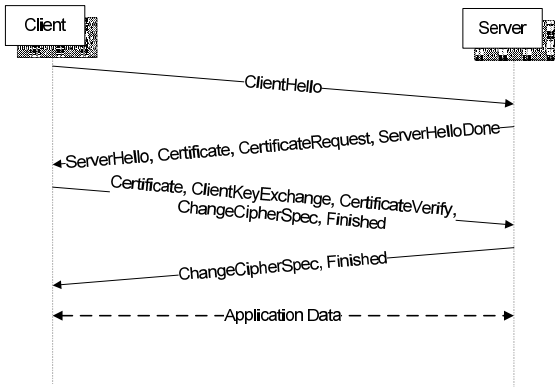

Fig. 1. TLS Exchange with Client and Server Authentication.

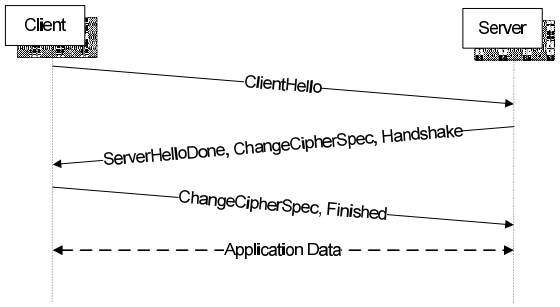

Fig. 2. TLS Session Resumption.

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 Exchange Protocol 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 [6], for examples 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 [7]).

Finally, TLS does not only provide protection for applications running on top of TCP or SCTP but with DTLS [8] the support for UDP and DCCP has been added as well.

III. Code Availability

There are many reasons for reusing protocols develops by the Internet Engineering Task Force (IETF) and one reason relevant to this discussion is the widespread availability of high-quality open source code. Changes are good that there is already an open source TLS stack available for the target operating system running on a smart object. In such a case the integration effort is significant lower leading to a faster time to market and lower development costs. This is not an irrelevant factor in the design of smart object 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 [9]. While many of these implementations provide a good starting point for a developer the axTLS project [10] 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 the code first needs to be decoupled from the EAP specific encapsulation. The WPA Supplicant [11] (for the client side) and the FreeRADIUS [12] (for the server side) contains TLS libraries that could be utilized – the WPA Supplicant already provides the support for an internal crypto library that does not rely on 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

TLS can easily be extended by adding new ciphersuites, if the large range of existing ciphersuites [13] do not match the requirements. The axTLS implementation is focused on the support of ciphersuites that use RSA-based key transport and does not support TLS-PSK or Diffie-Hellman based key exchanges. DSA and ECC based public key based algorithms are not supported either.

A very useful property of the axTLS library is the independence from other libraries.

IV. Communication Relationships

When designing a security solution the communication relationships need to be kept in mind. Consider the following scenario where a smart meter transmits its energy readings to other parties. The electricity will want to make sure that the obtained meter readings can be attributed to a specific meeting in a house hold. Most energy companies would find it unacceptable to have meter readings modified in transit (particularly changed to the disadvantage of the electricity provider since this would lead to fraud and a financial loss). Users in a house hold may want to ensure that only certain parties get to see the meter readings, for example for privacy reasons.

In this example a sensor may need to only ever talk to servers of a specific electricity company or even 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 certain servers. The meter may be pre-provisioned with the domain name of the servers 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 [14], in case of shared secret usage, or raw public keys used with TLS [15] 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 already deployed base needs to be considered. To illustrate the 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 most likely even conclude that TLS support is not necessary at all.

Consider the following extension to our scenarios where the meter needs is attached to the households home WLAN network and WLAN specific security mechanisms may need to be taken care of. On top of the link layer authentication the previously explained 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 SHA1, MD5, HMAC code) the overall code footprint will be likely be larger.

V. Code Size

One potential design goal is to reduce the size of the application code run on a smart object. To reduce the code size to a minimum it is crucial to understand the communication model and security goals, as explained in Section IV. Tradeoff decisions will have to be made consider the amount of flexibility and functionality offered by the TLS stack vs. the amount of required code that has to be run on a specific hardware platform. The following tables[3] provide information about the code size of various security functions.

TABLE I
Binary Code Size for Cryptography Support Functions

Library	Code Size
MD5	5,552 bytes
SHA1	3,392 bytes
HMAC	3,600 bytes
RSA	5,272 bytes
Big Integer Implementation	11,480 bytes
AES	7,096 bytes
RC4	2,232 bytes
Random Number Generator	6,232 bytes

[3]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.

The library implementing the random number generator (RNG) 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 the C file (crypto_misc.c) that contains the RNG but those had been excluded with compile-time flags.

TABLE II
Binary Code Size for TLS-specific Code

Library Name	Code Size	Description
x509	2,912 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	6,760 bytes	The ASN1 library (*asn1.c*) contains the necessary code to parse ASN1 data.
Generic TLS Library	20,144 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	5,928 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	2,864 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	4,624 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.

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 codesize. For example, the RC4 stream cipher codesize is 2,232 bytes compared to 7,096 bytes for AES usage.
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: shared secret keys, certificates, raw public keys (or a mixture of them)?
4) What TLS version and what TLS features, such as session resumption, have to be used?

Since this paper focuses on code size requirements rather than on optimization of RAM, energy consumption, or over-the-wire communication overhead the author had modified the code to provide support for raw public keys, as described in [15]. By using [15] 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 supports this extension it the Certificate payload but includes only the SubjectPublicKeyInfo part of the certificate in it (rather than the entire certificate). 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 (for an RSA-based public key). Note that the SubjectPublicKeyInfo does not contain the raw keys, namely public exponent and the modulus, but also a small ASN1 header preamble).

As one can guess, various code optimizations are possible when raw public keys are used instead of certificates. The table below shows the impact to the code size.

The functionality provided in *loader.c* was reduced to the bare minimum since the ability to load different types of certificates from the filesystem was removed. The remaining code was incorporated into *tls1.c*, as stated above. The author believes that with a few hours additional work further reductions in code size are accomplishable at the expense of flexibility, protocol performance, and developer convenience. For example, the code above still offers session resumption support - a feature that consumes a small amount of space while significantly improving the handshake performance. Also the usage of TLS session resumption without server-side state, see RFC 5077 [16], is a reasonable option for devices that communicate infrequently.

Of course, the transition to raw public keys does not change the code size of the core crypto functions, particularly those needed with the record layer.

TABLE III
BINARY CODE SIZE FOR RAW KEY SUPPORT IN TLS

Library Name	Code Size	Description
x509	1,944 bytes	The x509 related code (*x509.c*) now only provides the necessary function to invoke the parsing of the TLS server provided raw public key.
ASN1 Parser	3,232 bytes	The necessary support from the ASN1 library (*asn1.c*) is hugely reduced and concerns only the evaluation of the SubjectPublicKeyInfo block.
Generic TLS Library	16,472 bytes	This size of this library (*tls1.c*) was reduced slightly since additional functionality for loading keys into a data structure previously found in *loader.c* are now included in this file. Most of the obsolete code relates to certificate processing and various functions to retrieve certificate related data (e.g., the X509 distinguished name, subject alternative name).
TLS Client Library	4,864 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, but many other functions are left unmodified.
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).
OpenSSL Wrapper Functions	931 bytes	The OpenSSL wrapper library *openssl.c* was left untouched.

VI. CONCLUSION

TLS is a good example of a wildly[4] success security protocol that can be tailored to fit the needs of a specific deployment environment. This customization property offers the basis for a 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.

The 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 raw public keys. The author will do further work to reduce the code size (for example using the TLS 1.2 features) and to cross-compile the code to ARM-based platforms for better comparison with investigations by other groups.

REFERENCES

[1] T. Dierks and C. Allen, "The TLS Protocol - Version 1.0," Jan. 1999, RFC 2246, Request For Comments.
[2] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol - Version 1.1," Apr. 2006, RFC 4346, Request For Comments.
[3] ——, "The Transport Layer Security (TLS) Protocol - Version 1.2," Aug. 2008, RFC 5246, Request For Comments.
[4] D. Eastlake, "Transport Layer Security (TLS) Extensions: Extension Definitions," Jan. 2011, RFC 6066, Request For Comments.

[4]The IAB published a document, RFC 5218 [17], on the success criteria for protocols.

[5] P. Chown, "Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)," Jun. 2002, RFC 3268, Request For Comments.
[6] 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.
[7] 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.
[8] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security," Apr. 2006, RFC 4347, Request For Comments.
[9] Wikipedia, "Comparison of TLS Implementations," Feb. 2012, http://en.wikipedia.org/wiki/Comparison_of_TLS_Implementations.
[10] C. Rich, "axTLS embedded SSL," Feb. 2012, http://axtls.sourceforge.net/.
[11] J. Malinen, "Linux WPA/WPA2/IEEE 802.1X Supplicant," Feb. 2012, http://hostap.epitest.fi/wpa_supplicant/.
[12] A. DeKok, "The FreeRADIUS Project," Feb. 2012, http://freeradius.org/.
[13] IANA, "Transport Layer Security (TLS) Parameters," Feb. 2012, http://www.iana.org/assignments/tls-parameters/tls-parameters.xml.
[14] P. Eronen and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)," Dec. 2005, RFC 4279, Request For Comments.
[15] 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.
[16] 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.
[17] D. Thaler and B. Aboba, "What Makes for a Successful Protocol?" Jul. 2008, RFC 5218, Request For Comments.