“Lightweight” TLS

Hannes Tschofenig

IETF#85, Atlanta/USA, November 2012

What is useful guidance for TLS implementations in smart objects?

For example: Does your smart object talk to only a small set of pre-defined servers?

High Level Decision Points

- Assuming you know
  - what security threats do you care about, and
  - what security services do you have to offer.
- What do you optimize for?
  - Code size
  - Bandwidth
  - Number of roundtrips
  - Computational overhead
  - Energy consumption

Decision Factors

- How do the communication relationships look like?
- What credentials do need to be supported?
  - Pre-shared secrets
  - Passwords
  - Asymmetric credentials (RSA, ECC)
  - Combinations of the above.
- Various authentication and key exchange protocols available depending on detailed needs.
- Numerous algorithms for usage with data traffic protection
- Session Resumption (with and without server-side state)
- Alternative key validation techniques
- Possibility to replace record layer
- Relationship to other protocols (e.g., EAP)

Style of Writing?

- Not a good idea to replicate text from TLS specifications.
- Suggest to include implementation specific details as examples to illustrate cost (as a rough estimate).
  - Input from other implementers needed!
- Should the style be aligned with “lightweight IKEv2” document?

Backup

Example TLS Implementation Details

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

Note: The code was compiled under Ubuntu Linux using the -Os compiler flag setting for a 64- bit AMD machine.

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 during compile time), to parse them, and populate corresponding data structures.

Parts omitted by raw public key implementation