Secure Communications in Embedded Systems

Andreas Steffen

Zürcher Hochschule Winterthur

andreas.steffen@zhwin.ch

(to appear 2004 in the CRC Industrial Information Technology Handbook)

1 Introduction

In an increasing number of distributed embedded applications the individual nodes must communicate with each other over insecure channels like e.g. the public Internet or via wireless communication links. In order to withstand malevolent attacks, the end-to-end communication channels must be secured using cryptographically strong encryption and authentication algorithms. Fortunately, powerful secure communication protocols exist today and most of them are readily available as OpenSource software implementations. The biggest obstacle to their widespread use in distributed embedded systems is the limited memory and the restricted processing power available from today's low-cost and mostly low-power micro controller platforms. In this chapter we are going to show some promising approaches how security can be brought to embedded systems running time-critical applications.

This chapter on secure communications in embedded systems will be structured as follows:

- Section 2 of this chapter will give an overview on the secure communication solutions available for distributed embedded systems.
- Section 3 treats two of the key problems that have to be coped with when implementing cryptographically strong security in embedded systems. One problem is the generation of random numbers used for keying material and the second problem is the efficient handling of the computationally expensive public key operations required in most of the modern security protocols.
- Section 4 gives a typical example of a secure communications solution for a low-end embedded platform. It is shown that an SSL/TLS-enabled Web server can be implemented with a small memory foot print.
- Section 5 concludes this chapter by summarizing the relevant facts.

2 Security in the OSI Communications Stack

Security can be implemented at different levels of the communications stack. The well-known OSI communications model defines a physical layer at the very bottom and goes up via the data link, network, and transport layers up to the topmost application layer (see Figure 1). The OSI presentation and session layers have been omitted since they are not relevant in our context. In the following paragraphs we will shortly describe how security can be applied at each layer.

Communication layers	Security protocols
Application layer	ssh, S/MIME, PGP, http digest
Transport layer	SSL, TLS, WTLS
Network layer	IPsec
Data Link layer	WEP (WLAN), A5 (GSM), PPP
Physical layer	Scrambling, Frequency Hopping

Figure 1. Security implemented at different layers of the OSI stack

2.1 Physical Layer Security

At the low end of the communications stack the physical layer does not offer much protection in a strict cryptographical sense. Often the binary payload bits are scrambled with a fixed pseudo-random code, primarily in order to avoid the occurrence of long runs of ones or zeroes and some wireless channels like e.g. Bluetooth use fast frequency hopping in order to increase the physical robustness of the link. Although these measures offer some protection from casual eavesdropping, they do not withstand any serious attempts at intercepting the communication traffic.

2.2 Data Link Layer Security

Most wireless communication systems like e.g. IEEE 802.11 WLAN, Bluetooth, or GSM employ authentication and encryption at the data link layer in order to protect the data transmission over the vulnerable air interface. Unfortunately the Wireline Equivalent Privacy (WEP) encryption protocol of the IEEE 802.11 Wireless LAN standard is known to possess several serious deficiencies that make transmission highly insecure [1, 2]. The same is true for the A5 encryption cipher used by the GSM global cellular phone system that can be cracked with a simple PC in real time [3]. So if distributed embedded systems are communicating either over WLAN or GSM wireless links, no real security can be expected from the default layer 2 encryption and authentication protocols.

The most popular layer 2 protocol over wired links is Ethernet which does not possess any inherent security mechanisms. Over asynchronous modem links or synchronous ISDN connections the payload frames of the Point-to-Point Protocol (PPP) could optionally be encrypted using the PPP Encryption Control Protocol (ECP) [4], but two big vulnerabilities remain: First, PPP frames are not authenticated, making them prone to malicious changes and second, the PPP connection set up based on the auxiliary Link Control Protocol (LCP) is not secured at all, opening it to man-in-the-middle attacks and session-hijackings.

Generally it can be summarized that with the current layer 2 protocols in use, the data-link layer is not a suitable location for implementing strong communications security.

2.3 Network Layer Security

The most flexible approach to security can be realized at the network layer since this is the lowest communications layer that allows the implementation of end-to-end security in routed, multi-hop networks. If we restrict ourselves to the ubiquitous IP network layer protocol then the corresponding IP security protocol (IPsec) [5, 6] is the preferred choice. The IPsec protocol comes in two flavours: The Authentication Header (AH) protocol [7] secures IP packets from unwanted modifications by adding a cryptographic check sum, thus guaranteeing the data integrity of the payload and most of the IP header including the source and destination addresses. The Encapsulating Security Payload (ESP) protocol [8] offers strong encryption of IP packets and is usually preferred over AH since optional authentication of the payload data is also available with ESP. Only in the special cases where encryption is not permitted by national law or where it would to put too much of a computational burden on the embedded platforms, AH is used. Both ESP and AH are special IP protocols with protocol numbers 50 and 51, respectively. Because these protocols don't have port numbers they cannot be masqueraded in the same way as UDP or TCP datagrams.

IPsec further differentiates between Transport Mode which is mainly used for point-to-point connections between two single hosts and Tunnel Mode which tunnels the traffic between two sub-networks that are protected by a security gateway. Since in embedded systems usually direct host-to-host connections predominate, we will restrict our discussion to the AH and ESP transport mode payloads only, which are shown in Figure 2.

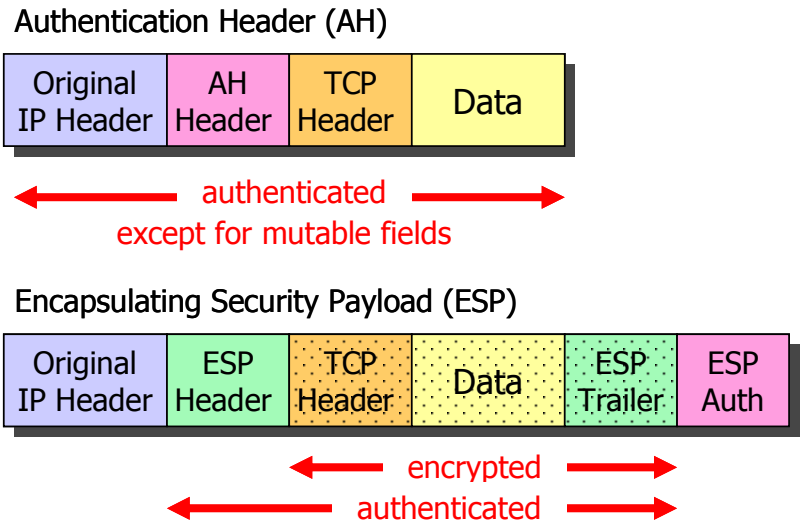

Figure 2. IPsec Transport Mode

The required overhead for ESP protection is shown in more detail in Figure 3. The ESP header consists of a 32 bit Security Parameters Index (SPI) which is used by the receiver to look up the session keys plus negotiated cryptographic parameters for the given IPsec connection. Each direction in a bi-directional IPsec connection has an SPI value of its own. The following 32 bit monotonically increasing Sequence Number prevents replay attacks. The ESP trailer is used to pad the payload data to an integer multiple of the block size required by the symmetric block cipher. The standard Triple DES (3DES) encryption algorithm works on 64 bit blocks of data, so that in the worst case 7 bytes must be padded, whereas the Advanced Encryption Standard (AES), the official successor to 3DES, uses 128 bit data blocks, so that the overhead becomes 15 bytes at the worst. Add to this 2 bytes for the pad length and next header fields. The block encryption algorithms also need an initialisation vector (IV) comprising the size of one data block. This means an additional 8 bytes of overhead for 3DES and 16 bytes for AES, respectively. Finally the cryptographic checksum used for ESP authentication takes up another 12 bytes, resulting to a total overhead of 37 bytes for 3DES encryption and 53 bytes for AES. This is not much if large volumes of data are transferred as for e.g. during an ftp download, but it can have a large impact if many small IP packets are transmitted, e.g. when running a telnet session.

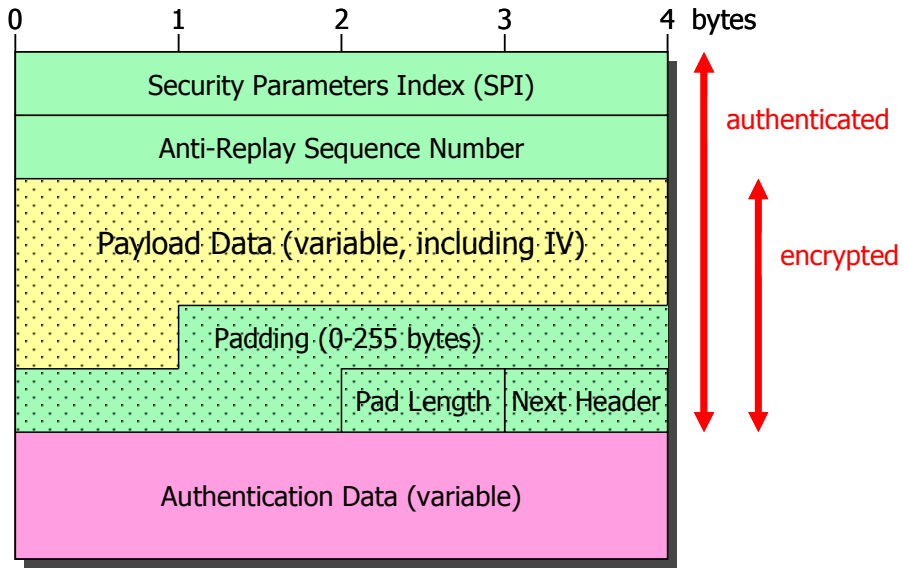

Figure 3. ESP payload structure

The kernel part of an IPsec stack which implements the real-time ESP encapsulation of IP packets can be made quite compact. For distributed systems running an embedded Linux operating system like e.g. µClinux [9], a free but nevertheless mature IPsec solution is available from the OpenSource Linux FreeS/WAN project [10]. Several commercial IPsec stacks for embedded platforms are also offered on the market. The RAM footprint is usually kept between 200-300 kBytes.

The most serious drawback of IPsec is the complexity involved in setting up protected connections. Although in principle the configuration could be done manually by specifying two common Security Parameters Index (SPI) values for the two directions plus a set of shared authentication and encryption keys, this procedure would be too error-prone to be feasible in a practical application, especially if frequent re-keying is

required. Therefore automatic connection set-up is preferred using the standardized Internet Key Exchange (IKE) protocol [11] which uses the well-known UDP port 500. IKE is a rather complex protocol comprising a multitude of configuration options. It consists of a Phase 1 called Main Mode where six messages are exchanged in order to establish trust and to compute a common secret key between the two end points. Main Mode is followed by a Quick Mode or Phase 2 negotiation where another three messages are needed to set up the actual connection. A typical Main Mode exchange with mutual authentication based on pre-shared secrets is shown in Figure 4.

- The initiator sends a proposal, listing all supported encryption and authentication algorithms. The responder selects among them one set of common algorithms and sends the selection back to the initiator.
- The second message pair establishes a common secret with the help of a Diffie-Hellman (DH) Key Exchange. In order to prevent replay attacks also a pair of random values N_I and N_R are exchanged.
- Starting with the third message exchange all further IKE communication is encrypted in order to prevent man-in-the-middle attacks. The encryption key is derived from the common DH secret, the random values N_I and N_R and the pre-shared secret known only to the two end points. Over the secure channel an ID string which establishes the identity and a hash value authenticated with the pre-shared secret are exchanged.

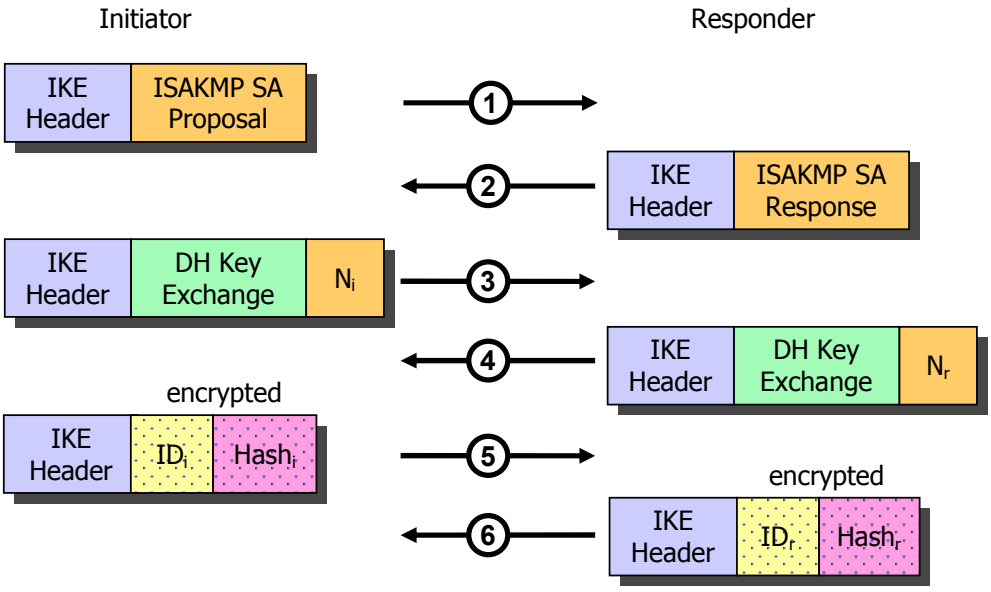

Figure 4. IKE Main Mode authentication based on pre-shared secrets

Another variant of the IKE Main Mode depicted in Figure 5 uses RSA public key authentication (only the last two IKE messages are shown, the first four being the same as in Figure 4). The use of RSA signatures in connection with X.509 certificates, which are a convenient means of distributing the needed RSA public keys, is the most powerful approach to IPsec, especially if a large number of connections must be maintained.

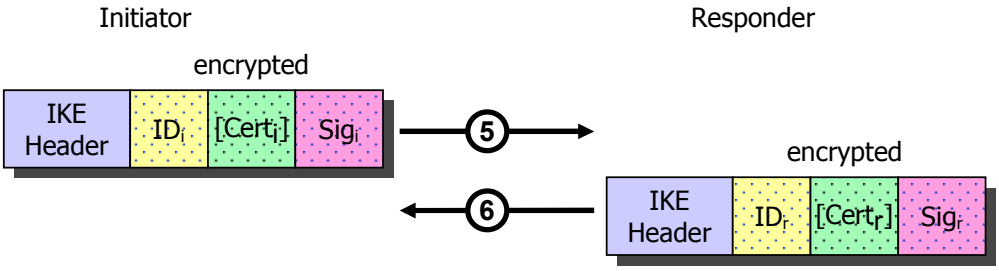

Figure 5. IKE Main Mode authentication based on X.509 certificates

X.509 certificate support for embedded Linux IPsec solutions is available as a patch [12] developed by the Security Group of the Zurich University of Applied Sciences in Winterthur that can be applied to the FreeS/WAN distribution [10]. Depending on the number of implemented features an IKE daemon can become quite large. Footprints between 200 kBytes for a very lean client using pre-shared secrets only and more than 1 MBytes for a full-fledged IKE implementation are not uncommon.

The complexity of the IKE protocol is the main reason that use of IPsec is usually restricted to high-end 32 bit embedded platforms that possess the required memory and computational resources.

2.4 Transport Layer Security

The most generic solution at the transport level is the well-known Secure Sockets Layer (SSL) [13, 15], as well as its official IETF successor called Transport Layer Security (TLS) [14, 15] and its lean variant WTLS for wireless applications. The SSL/TLS protocols are layered on top of the reliable TCP transport layer protocol and are typically used in client/server applications, e.g. a central controller managing several distributed nodes, but SSL/TLS can also be used for peer-to-peer communication.

By taking a look at Figure 6 we see the fundamental difference between network and transport layer security. Whereas IPsec requires large and potentially dangerous changes in the kernel of the given operating system, SSL/TLS leaves the built-in TCP/IP communications stack untouched because it uses regular TCP sockets. From the application's point of view IPsec has the big advantage that the interface to the transport layer doesn't change at all, so that the application program is not even aware of the existence of an additional security layer and any transport protocol (TCP, UDP, ICMP, etc.) can be used. With SSL/TLS special secure sockets must be opened by the application and TCP is the only transport layer supported. Fortunately, thanks to the wide-spread popularity of SSL/TLS, many applications now work with secure sockets which in fact are very similar to normal TCP sockets.

SSL/TLS uses a record structure to encrypt and authenticate the application data and to embed them into a TCP/IP stream. Similar to IPsec and its companion IKE connection negotiation protocol, the SSL/TLS handshake protocol shown in Figure 7 is used to set up a secure SSL/TLS tunnel.

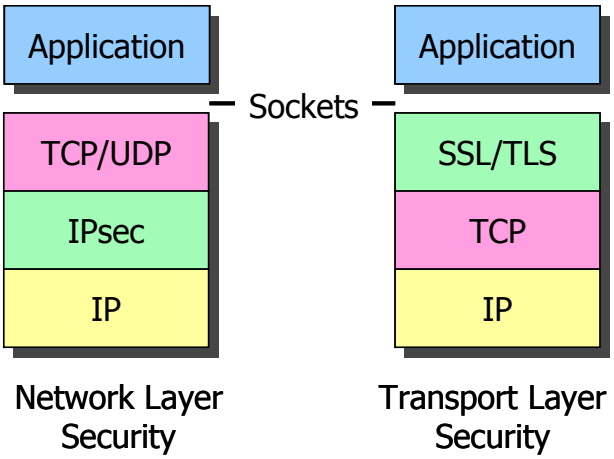

Figure 6. Comparison network vs. transport layer security

SSL/TLS Handshake Protocol

- The handshake is initiated by the client who sends a ClientHello message which lists all its suites of supported encryption, authentication and compression protocols.
- The server answers by selecting one of the offered suites and includes the decision in a ServerHello message. Both client and server add random values R_C and R_S, respectively to their hello messages in order to make each session unique, thereby preventing replay attacks.,
- A server usually possesses a trusted server certificate which it sends to the client right after the ServerHello message. By including an optional CertificateRequest the client can be forced to authenticate itself in turn by requiring it to send a client certificate, too.
- The ServerHelloDone message terminates the first server message block and prompts the client to take over the protocol lead again.
- If requested to do so, the client sends its certificate together with a signature contained in the CertificateVerify message. As mentioned above, client-side authentication is optional and not used very often.
- The client always generates a 48 bytes random pre-master secret, encrypts it with the public key extracted from the received server certificate and puts the protected secret in the ClientKeyExchange message. Based on the pre-master secret the client then computes the master secret and derives all required encryption and authentication session keys. It then activates the chosen cipher suite, signals this to the server by issuing a ChangeCipherSpec message and adds an already encrypted Finished message that can be used by the peer side to test the successful establishment of the secure connection.
- The server in turn decrypts the received ClientKeyExchange message using its private key. With the extracted pre-master secret the server can then generate the same master secret and derived sessions keys as the client's. By sending the ChangeCipherSpec and Finished messages the server signals its readiness to switch over to the secure channel, too.

From this moment on, all application data is exchanged in encrypted and authenticated form.

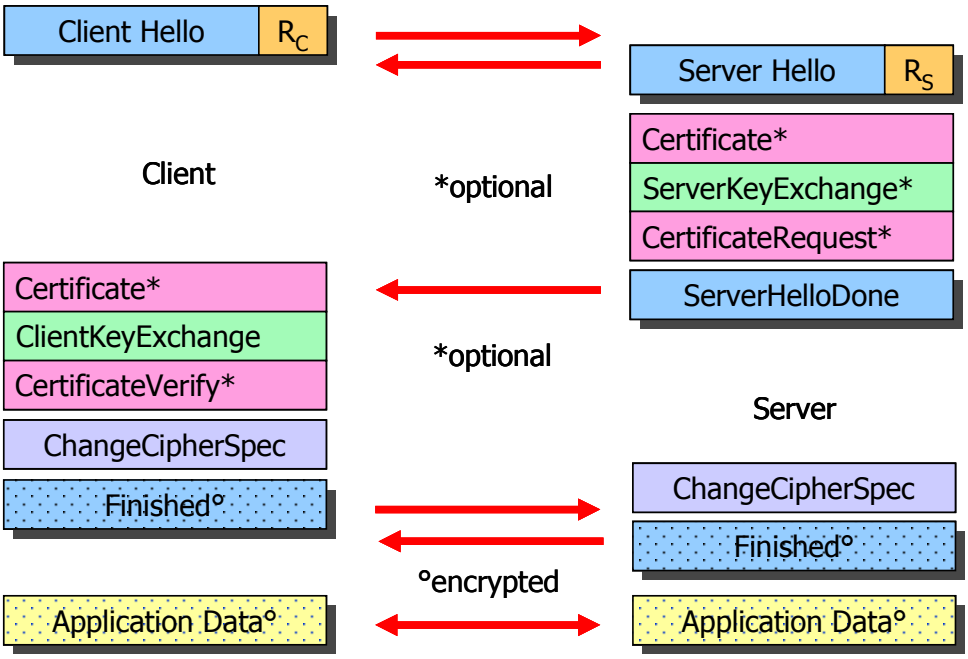

Figure 7. SSL/TLS handshake protocol

Every newly opened socket needs to be initialized first using the SSL/TLS handshake protocol. If e.g. the obsolete HTTP 1.0 protocol is used to access a web site, then each graphical element of a HTML page will be downloaded over a socket of its own. This would mean that every time a computationally expensive RSA public key authentication based on a 1024 bit key would have to be executed. In order to avoid a full handshake operation, SSL/TLS offers the possibility to resume or clone an already established session in the abbreviated fashion detailed in Figure 8.

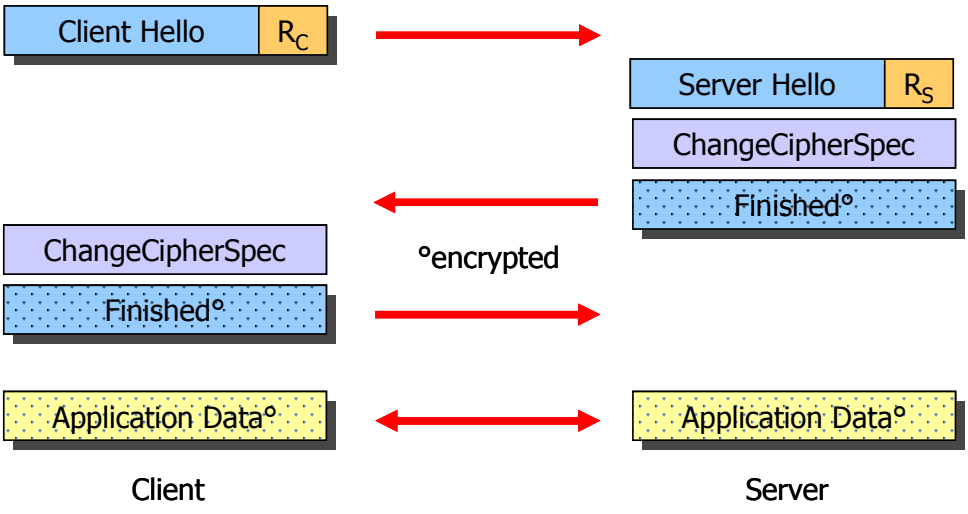

Figure 8. Resuming SSL/TLS session

If the client wants to resume a session or open an additional socket then it sends the session ID of the existing session as part of the ClientHello message. If the server still has the corresponding session data including the pre-master secret in its cache then it will reply by immediately sending the ChangeCipherSpec and Finished messages. Neverthe less, using a fresh set of the random values R_C and R_S, a set of new session keys are derived from the cached pre-master secret.

A readily available OpenSource SSL/TLS implementation is the popular OpenSSL package [16]. Due to its innumerable options and supported algorithms the required ssl and crypto libraries add up to a huge size of 1.6 Mbytes. OpenSSL in its direct form is therefore only suitable for high-end 32 bit platforms that usually run an embedded Linux operation system. In section 4 we will demonstrate how a lightweight SSL/TLS stack can be derived from the original OpenSSL library.

2.5 Application Layer Security

A typical example of an application layer security application is the popular secure shell (ssh) [17], which allows a secure login for administration and monitoring purposes - functions that were classically done by system administrators using the insecure and now deprecated telnet protocol. The authentication mechanisms are similar to those used in the IKE and SSL/TLS handshake protocols. Either raw RSA keys or X.509 certificates are supported.

Another possible security approach at the application level would be to authenticate and encrypt the message bodies using the S/MIME [16] or PGP/GnuPG [18] packages known from secure email. But be aware that additional precautions must be taken to thwart replay attacks using recorded past messages.

On many embedded systems a Web server is running by default, allowing the configuration and monitoring of the embedded application via an interactive Web interface. A straightforward approach in this case would be to use the built-in security mechanisms of the HTTP protocol. Two authentication variants are available: Basic and Digest Access Authentication [19], whereas encryption is not supported and must be provided by other means. The Basic Access Authentication scheme defined by the HTTP 1.0 protocol is not secure at all and should not be used because the username / password pair of the client is transmitted in the clear and can easily be intercepted. The Digest Access Authentication introduced by the newer HTTP 1.1 protocol is much more secure because the secret password never crosses the communication channel. Instead a random challenge string sent by the server is hashed together with the client's username and password to form an MD5 digest which is sent back as a response to the HTTP server. This digest is vulnerable to brute-force dictionary attacks, though. If a weak password is chosen by the user, then an offline cracking attempt will find the password in less than an hour if it has been chosen from a multi-lingual dictionary and only be been slightly modified e.g. by appending a number.

The big advantage of the HTTP Digest Access Authentication is the relative ease of its implementation and use, whereas the drawbacks are the lack of an encryption option and the potential vulnerability of the openly transmitted digest to dictionary attacks due to weakly chosen passwords.

3 Key Problems in Embedded Systems

In this section we will examine to two of the key problems that are typical for low-end embedded system platforms. The first topic is the generation of true random material in a low-entropy environment and the second item will concentrate on the computationally intensive operations required by the standard RSA public key cryptosystem as used by IPsec's Internet Key Exchange protocol, the SSL/TLS handshake protocol, and the SSH public key based authentication.

3.1 Generation of True Random Numbers

The security of modern cryptographic algorithms is founded exclusively on the true randomness of the chosen session keys. The same is valid for the challenge/response protocols used for peer authentication that are based on the prerequisite that the challenge strings will never occur twice during the whole lifetime of the system. As a consequence a lot of random or pseudo-random material must be generated in order to satisfy the demands of a truly secure communication channel.

In a workstation environment usually sufficient random information can be continuously collected from the timing of keyboard strokes and mouse movements. On unattended servers and firewalls only disk access timing and arrival times of network packets remain as practical entropy sources, whereas the latter source might raise the concern of some paranoid crypto experts who postulate that the network traffic timing could potentially be manipulated by a malicious attacker. Fortunately modern Intel-based platforms often possess a chip set [20] with a built-in random generator (RNG) that produces random numbers in copious quantities.

On low-end diskless embedded platforms it becomes increasingly difficult to gather any random material at all. The usual makeshift solution consists of transferring an initialisation file containing about 1024 true random bytes to the target platform during system configuration. This pool of random data is then used as a seed for a pseudo-random generator. After each access to the generator the new state of the random pool is saved back to the file, thus guaranteeing that the random generator will always start from the last used state and will therefore never repeat itself.

Since embedded systems are often used to control real-time processes, there might be random information readily available at the numerous control inputs. As an alternative a noise source or some other reliable random device could be attached to some of the unconnected inputs if available. Practical results undertaken at the Zurich University of Applied Sciences in Winterthur, using both a noisy diode and a free-running oscillator have shown that with little hardware effort, a low-cost, low-complexity random number generator can be built suited to low-end embedded platforms. Great care must be taken to de-skew the generated stream of raw random bits, since it is very difficult to achieve a perfectly even distribution of 'zeros' and 'ones'. Either the simple von Neumann de-skewing algorithm [21] can be used or even better, all raw bits gathered from hardware random sources should first be hashed into a random pool using the MD5 or SHA-1 digest algorithms in order to improve the statistical properties of the derived key material.

3.2 Public Key Operations

The second obstacle in providing strong cryptography to embedded systems is the extremely time-consuming modular exponentiation operation

$$y = x^e \bmod n$$

that is the corner stone of both the RSA and Diffie-Hellman public key algorithms. Usually the basis x, the exponent e, and the modulus n are large integers having a size k of 512..2048 bits. Using its RSA private key an SSL/TLS server running on an embedded system must compute a singular modular exponentiation in order to decrypt the pre-master secret that had been previously encrypted by the SSL/TLS client with the server's RSA public key. The processing time needed to execute this crucial RSA decryption operation was measured on the 20 MHz IPC@CHIP platform described in section 4. The results are listed in Table 1.

RSA key size k [bits]	Processing time t [s]
512	8
768	22
1024	48
1536	150
2048	335

Table 1. Processing time for an RSA decryption

The selected RSA key size k is equivalent to the size of the modulus n used in the modular exponentiation. The observed processing time t varies from a mere 8 seconds for the insecure key size k of 512 bits, to 48 seconds for the standard RSA key size of 1024 bits, and a horrendous 335 seconds for the 2048 bit keys that might be required in a couple of years from now. These execution times listed above are typical values for machine code automatically generated by a standard C compiler. By manually coding the critical parts of the multi-precision multiplication in assembly code, an acceleration of nearly a factor of two can be achieved.

The processing time for the 16 bit IPC@CHIP platform can be well approximated by the formula

$$t \approx \frac{702 \cdot (k/1024)^3 + 273 \cdot (k/1024)^2}{f_c\,[\mathrm{MHz}]} \quad [\mathrm{s}]$$

which is plotted together with our measurement data in Figure 9. For large key lengths k the processing time increases with the third power of k. This asymptotic behaviour can be easily explained by the fact that the execution time for a multi-precision multiplication of two k bit numbers is proportional to k^2 and that a modular exponentiation with a k bit exponent requires on the average about $k/2$ multi-precision multiplications. Besides the clock frequency, the processor's word size is very decisive for the execution speed. Halving the word size from 16 bits to 8 bits increases the time for a multi-precision multiplication by a factor of four whereas doubling it to 32 bits decreases it by the same factor.

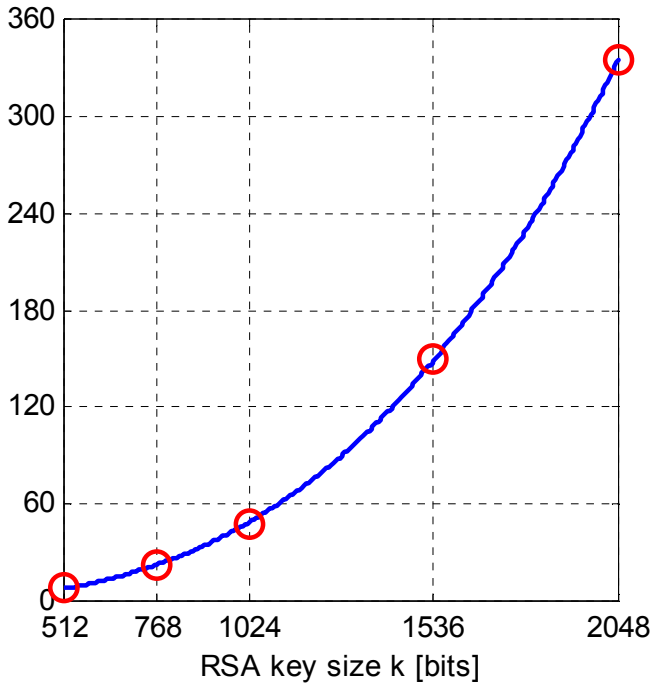

Figure 9. Processing time in [s] for RSA decryption

With the typical system clock of f_c = 20 MHz for a low-end embedded processor, an RSA decryption takes 48 seconds to execute whereas with today's 1000 MHz 32 bit Pentium processors the same operation can be done well below one second. Such an initial delay of nearly one minute can be prohibitive in many real-time application, although as a workaround in the particular case of the SSL/TLS protocol, a session can be resumed after a break without the need to encrypt, transmit and decrypt a new pre-master secret.

Time to break in MIPS years	RSA key size [bits]	ECC key size [bits]	RSA/ECC key size ratio
10^4	512	106	5:1
10^8	768	132	6:1
10^{11}	1024	160	7:1
10^{20}	2048	210	10:1

Table 2. RSA/ECC Key Size Equivalent Strength Comparison (data from [23])

The class of Elliptic Curve Public Key Cryptosystem (ECC) public key algorithms [22] can offer a significant improvement in processing delay, thus making many time-critical applications feasible. The main reason for this speed gain are the much shorter key lengths required (compared to the RSA cryptosystem) to achieve the same cryptographic strength. Table 2 shows that a 160 bit ECC key is equivalent in strength to a 1024 bit RSA key. In the years to come 2048 bit RSA keys might be required to maintain adequate security which doubles the current RSA key lengths whereas the ECC key length will have to grow by 50 bits only.

Due to the much shorter key lengths, ECC promises acceleration factors of 10..100 for a 160 bit ECC key, compared to a 1024 bit RSA key [23], especially if the algorithm is realized in special hardware or on a smartcard crypto-coprocessor. ECC software implementations are still not widely deployed, with Certicom being the market leader. Fortunately, thanks to a software donation by Sun Microsystems, the [snapshot] version of the OpenSSL package [16] includes now full ECC support.

4 Example of an SSL/TLS-enabled Embedded Platform

In October 2000 the Security Group of the Zurich University of Applied Sciences in Winterthur started its activities in the field of lightweight security protocols by porting the OpenSSL library [16] to the tiny 16 bit 80186-based IPC@CHIP embedded Web server platform shown in Figure 10.

The IPC@CHIP is based on a 16 bit little-endian Intel-compatible 80186 processor running with a modest system clock of f_c = 20 MHz. The platform is equipped with 512 kbytes of RAM and 512 kbytes of non-volatile flash memory and possesses an integrated 10Base-T Ethernet interface. Since about half of the flash space is used to store the BIOS comprising a built-in TCP/IP stack, 256 kbytes of free memory are available for the SSL/TLS protocol implementation and the actual application running on the embedded system.

Figure 10. SSL/TLS-enabled mini Web server

As part of their diploma thesis the two ZHW students Bernhard Lenzlinger and Andreas Zingg succeeded in stripping the OpenSSL library from the original 1.6 Mbytes down to a mere 580 kbytes. Based on this shrunk 16 bit library they were able to build an SSL/TLS-enabled Web server application. With its small size of 100 kbytes (after compression, but including the statically linked OpenSSL library) the SSL/TLS server fitted without problems into the available flash memory.

Here are the features offered by the mini Web server depicted in Figure 10:

- Support of the SSL 3.0 and TLS 1.0 protocols.
- File-based initialisation of the random seed.
- MD5 and SHA-1 hash used for message authentication.
- RC4 stream cipher with 128 bit symmetric key used for encryption.
- X.509 certificate used for server authentication.

- 1024 bit RSA key embedded in the X.509 server certificate used for the encrypted transmission of the pre-master secret from client to server.
- Secure client-side authentication based on HTTP passwords after SSL/TLS link has been successfully established.
- Single HTTP session supported, no multithreading.

Multiple sessions could be cached making session resuming possible, but only a single concurrent TCP/IP socket was supported.

Encouraged by the success of this lightweight SSL/TLS rapid prototype, the Security Group of the Zurich University of Applied Sciences in Winterthur decided to launch the project "*Secure Communications in Distributed Embedded Systems"* in July 2002*,* with the following extended objectives:

- Implementation of a modular and lightweight SSL 3.0 / TLS 1.0 protocol stack that can be quickly ported with little or no modifications to a large variety of 16 bit and 32 bit embedded platforms possessing limited memory resources and processing power.
- Support of the fast Advanced Encryption Standard (AES) [24] with 128 bit symmetric keys which is much more secure than the current default RC4 stream cipher.
- Support of the fast Elliptic Curve Cryptosystem (ECC) [22] in addition to the traditional RSA Public Key Cryptosystem.
- Multi-threading capability allowing multiple concurrent TCP/IP sockets.

The design of this extended and modular SSL/TLS stack will be tailored to the needs of a large percentage of the existing embedded system platforms, spanning the whole range from the low-end 16 bit micro controllers with proprietary operating systems up to the powerful 32 bit processors running under embedded Linux.

5 Conclusions

In this chapter we have shown that secure communications in distributed embedded systems can be implemented in nearly all layers of the communications stack, but we express a pronounced preference for the network and transport layers. A practical implementation of a SSL/TLS stack on a low-end 16 bit embedded platform has clearly demonstrated that lightweight security protocols are feasible. For high-end, embedded Linux based platforms, IPsec might be the optimum choice due to the maximum flexibility that can achieved. With the advent of the fast AES and ECC algorithms, the speed handicap incurred by low-cost and low-power processors will be alleviated and time-critical applications will become feasible. In special cases hardware acceleration using a crypto co-processor might be required. Such devices are available e.g. in the form of smart card crypto co-processors. For pure console-based applications the secure shell (ssh) is certainly the right choice. And if your security requirements are not so stringent and encryption of the payload is not mandatory then HTTP Digest Access Authentication might be the appropriate option.

References

[1] N. Borisov, I. Goldberg, D. Wagner, *Intercepting Mobile Communications: The Insecurity of 802.11*, Proceedings of the Seventh Annual International Conference on Mobile Computing And Networking, July 16-21, 2001.

[2] S. Fluhrer, I. Mantin, and A. Shamir, *Weaknesses in the Key Scheduling Algorithm of RC4*, August 2001.

[3] A. Biryukow, A. Shamir, D. Wagner, *Real Time Cryptanalysis of A5/1 on a PC*, Fast Software Encryption Workshop, April 10-12, 2000.

[4] G. Meyer, The PPP Encryption Control Protocol (ECP), RFC 1968, June 1996.

[5] S. Kent, R. Atkinson, Security Architecture for the Internet Protocol, RFC 2401, Nov. 1998.

[6] S. Frankel, *Demystifying the IPsec Puzzle*, Artech House, 2001.

[7] S. Kent, IP Authentication Header (AH), RFC 2402, Nov. 1998.

[8] S. Kent, IP Encapsulating Security Payload (ESP), RFC 2406, Nov. 1998.

[9] μClinux Project, www.uclinux.org.

[10] Linux FreeS/WAN Project, www.freeswan.org.

[11] D. Harkins, D. Carrel, *The Internet Key Exchange (IKE)*, RFC 2409, Nov. 1998.

[12] X.509 Certificate support for Linux FreeS/WAN, www.strongsec.com/freeswan.

[13] A. Freier, P. Karlton, P. Kocher, *The SSL Protocol Version 3.0*, Internet draft , Nov. 1996, http://wp.netscape.com/eng/ssl3/draft302.txt.

[14] T. Dierks, C. Allen, *The TLS Protocol Version 1.0*, RFC 2246, Jan. 1999.

[15] E. Rescorla, *SSL and TLS: Designing and Building Secure Systems*, Addison Wesley, 2001.

[16] OpenSSL Project, www.openssl.org.

[17] OpenSSH Project, www.openssh.org

[18] International PGP Home Page, www.pgpi.org.

[19] J. Franks et al., *HTTP Authentication: Basic and Digest Access Authentication*, RFC 2617, June 1999.

[20] Intel Random Number Generator, www.intel.com/design/security/rng/rngbrf.htm.

[21] D. Eastlake, S. Crocker, J. Schiller, *Randomness Recommendations for Security*, RFC 1750, Dec. 1994.

[22] A. Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, 1993.

[23] *The Elliptic Curve Cryptosystem for Smart Cards*, A Certicom White Paper, May 1998, www.certicom.com/resources/download/ECC_SC.pdf.

[24] Advanced Encryption Standard (AES), www.nist.gov/aes.