Secure Communications in Distributed Embedded Systems

Andreas Steffen (andreas.steffen@zhwin.ch)

Security Group, Zurich University of Applied Sciences in Winterthur, CH-8401 Winterthur, Switzerland

Security protocols can be implemented at different layers of the communications stack and are readily available as OpenSource implementations. The largest impediment against their widespread use in distributed embedded systems are the severely limited memory resources and processing power available with typical embedded system hardware platforms. The way out of this dilemma are lightweight security protocols. In this paper we present an SSL/TLS enabled embedded web server that we have implemented on a lightweight 16-bit 80186 platform running at 20 MHz with only 256 kbytes of available flash memory and 512 kbytes of RAM. A distributed embedded system can thus be configured and controlled using a standard web browser with authentication based on a X.509 certificate and a 1024 bit RSA public key and encryption based on the RC4 stream cipher using a 128 bit symmetric session key. Two key problems typical of embedded systems namely the generation of true random numbers on low-entropy embedded platforms and the speed up of public key operations using elliptic curve cryptosystems will also be discussed.

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 meagre processing power available from today's low-cost and mostly low-power micro controller platforms. This paper is going to show some promising approaches how security can be brought to embedded systems running time-critical applications.

Security can be implemented at various levels of the communications stack which is always founded on a physical layer and goes up via the data link, network, and transport layers up to the topmost application layer (see Figure 1). In the following introductory paragraphs we will shortly describe how security can be applied at each layer.

1.1 Application Layer Security

A typical example of an application layer security application is the popular secure shell (ssh [1]), which allows a secure login for administration and monitoring purposes - functions that were classically done by system administrators using the insecure and therefore now deprecated telnet protocol.

Another possible security approach at the application level would be to authenticate and encrypt the message bodies using the S/MIME [2] or PGP [3] packages known from secure email. But be aware that additional precautions must be taken to thwart replay attacks using past messages.

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

Figure 1. Security implemented at various layers

1.2 Physical Layer Security

At the lower 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 code 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.

1.3 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 [4] that make transmission highly insecure. The same is true for the GSM A5 encryption cipher that can be cracked with a simple PC in real time [5]. So if distributed embedded systems must rely either on WLAN or GSM links, no real security can be expected from layer 2 encryption and authentication.

1.4 Network Layer Security

The most flexible approach to security can be found at the network layer since this is the lowest communications layer that allows the implementation of end-to-end security in routed networks. If we restrict ourselves to the ubiquitous IP network layer protocol then the corresponding IPsec [6, 7] security protocol is the preferred choice. For distributed systems running an embedded Linux operating system a free but nevertheless mature solution is offered by the OpenSource FreeS/WAN IPsec stack [8]. Mutual authentication of the communicating nodes can either be based on preshared secrets or on raw RSA public/private key pairs. By additionally applying the X.509 certificate patch [9] developed by the Security Group of the Zurich University of Applied Sciences in Winterthur to the FreeS/WAN software package, the RSA public keys can be openly distributed in the form of validated X.509 certificates.

The most serious drawback of IPsec is its complexity, especially if the Internet Key Exchange protocol (IKE [10]) is used to automatically set up the encrypted and authenticated connections. Due to its large footprint IPsec is currently mainly used on high-end 32 bit embedded platforms possessing enough memory resources.

1.5 Transport Layer Security

A generic solution at the transport level is the well-known Secure Sockets Layer (SSL [11, 13]), its official IETF successor called Transport Layer Security (TLS [12, 13]) 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 can also be used for peer-to-peer communication.

A readily available OpenSource SSL/TLS implementation is the popular OpenSSL package [2]. 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, mostly embedded Linux based platforms. In the following section we want to demonstrate how a very lightweight SSL/TLS stack can be derived from the original OpenSSL library.

2 SSL/TLS-ENABLED MINI WEB SERVERS

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 with a diploma thesis [14] having the objective to port the OpenSSL library to the tiny 16 bit 80186-based IPC@CHIP [15] embedded web server platform shown in Figure 2.

Figure 2. SSL/TLS-enabled mini Web server

2.1 IPC@CHIP Platform

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.

2.2 OpenSSL ported to IPC@CHIP

As part of their diploma thesis [14] 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 2:

- Support of the SSL 3.0 and TLS 1.0 protocols.
- File-based initialization of 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 premaster 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.

2.3 New SSL/TLS Stack from Scratch

Unfortunately the OpenSSL-derived SSL/TLS stack suffered from some untrackable dynamic memory allocation problems that led to frequent crashes of the IPC@CHIP platform. Therefore in August 2001, in preparation for the Orbit/Comdex Europe 2001 trade fair in Basel where the SSL-enabled web server was to be exhibited, the whole SSL/TLS protocol was rewritten from scratch in the language C by the author of this paper, retaining only the cryptographic functions from the original OpenSSL distribution.

As a result the new SSL/TLS stack, including a skeleton HTTPS server has shrunk even more in size and uses now only 40 kbytes of FLASH memory. Some features like multithreading, full TLS support and the cryptographically sound initial seeding of the pseudo-random generator (see section 3.1) are still missing, but the final full-fledged SSL/TLS stack is expected to fit within a 50-60 kbyte footprint.

2.4 Modular Lightweight SSL/TLS Protocol

Encouraged by the success of the lightweight SSL/TLS protocol, the Security Group of the Zurich University of Applied Sciences in Winterthur (ZHW [16]) decided to launch the R&D project "*Secure Communications in Distributed Embedded Systems*" jointly with the Microelectronics Laboratory of the Scuola Universitaria Professionale della Svizzera Italiana (SUPSI [17]). The project has been started in July 2002 under the grant SNS 6113.1 financed to a large extent by soft[net] [18] and is part of an initiative undertaken by the Embedded and Distributed Solutions Network (EDiSoN [19]) to further the creation and dissemination of know-how in the area of distributed embedded systems in close cooperation with Swiss industry.

The "EDiSoN_Sec" project has the following 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- and 32-bit embedded platforms possessing limited memory resources and processing power.
- Support of the fast Advanced Encryption Standard (AES [20]) with 128 bit symmetric keys.
- Support of the fast Elliptic Curve Cryptosystem (ECC [21]) in addition to the traditional RSA Public Key Cryptosystem.
- Hardware acceleration of AES and ECC/RSA using a low-power crypto coprocessor.
- Generation of true random numbers on embedded platforms.

3 KEY PROBLEMS

In this section we will examine to two of the key problems that are typical for low-end embedded system platforms. One 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.

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 random challenge strings will never occur twice during the whole lifetime of the system. As a consequence a lot of random or "nearly-random" material must be generated in order to satisfy the demands from 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,. where as 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 [22] with a built-in random generator producing 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 in transferring an initialization file containing about 1024 true random bytes to the target platform during system configuration where this pool of random data is then used as a seed for a pseudo-random generator. After each access to the random 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.

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.

3.2 Public Key Operations

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

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

constitutes the corner stone of 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 server running on an embedded system must compute a singular modular exponentiation in order to decrypt the preshared secret that had been previously encrypted by the SSL 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 specified in section 2.1. 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 RSA decryption

The selected RSA key size k is equivalent to the size of the modulus n used in the modular exponentiation. The observerd 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 bits that might be required in a copuple of years from now. The processing time for the 16 bit IPC@CHIP platform can by approximated by the formula

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

that is plotted together with our measurement data in Figure 3.

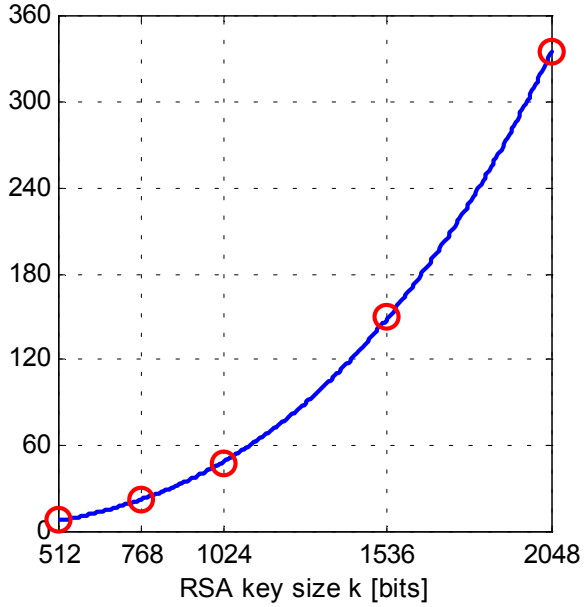

Figure 3. Processing time in [s] for RSA decryption

With the typical system clock of f_c = 20 MHz for a low-end embedded processor, an RSA decryption takes 48 seconds to execute where as 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 fatal in many real-time applications although in the case of the SSL/TLS protocol a session can be resumed after a break without the need to encrypt, transmit and decrypt a new premaster 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 Cryptosystem (ECC [21]) public key algorithms 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 where as the ECC key length will have to grow by 50 bits only.

Due to the much shorter key lengths ECC promises acceleration factors of 20..100 for a 160 bit ECC key compared to a 1024 bit RSA key [23]. The main obstacle impeding a fast breakthrough of this new technology has been the fact that most efficient ECC algorithms are patented by Certicom [24] and must be licensed from that company. This might also be the reason that OpenSource implementations of ECC algorithms are still not available.

4 CONCLUSIONS

In this paper we have shown that secure communications in distributed embedded systems can be implemented in various places of the communications stack but we express a clear 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. 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 hardware crypto processor might be required. Such devices will be available e.g. on smart cards.

REFERENCES

1. OpenSSH Project <www.openssh.org>
2. OpenSSL Project <www.openssl.org>
3. International PGP Home Page <www.pgpi.org>
4. 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. <www.isaac.cs.berkeley.edu/isaac/mobicom.pdf>
5. A. Biryukow, A. Shamir, D. Wagner, *Real Time Cryptanalysis of A5/1 on a PC*, Fast Software Encryption Workshop, April 10-12, 2000.

<www.wisdom.weizmann.ac.il/~albi/fse00a5revised.ps>

6. S. Kent, R. Atkinson, Security Architecture for the Internet Protocol, RFC 2401, Nov. 1998. <www.ietf.org/rfc/rfc2401.txt>
7. S. Frankel, *Demystifying the IPsec Puzzle*, Artech House, 2001, ISBN 1-58053-079-6.
8. FreeS/WAN Project <www.freeswan.org>
9. X.509 Certificate Support for Linux FreeS/WAN <www.strongsec.com/freeswan>
10. D. Harkins, D. Carrel, *The Internet Key Exchange (IKE)*, RFC 2409, Nov. 1998. <www.ietf.org/rfc/rfc2409.txt>
11. A. Freier, P. Karlton, P. Kocher, *The SSL Protocol Version 3.0*, Internet draft <wp.netscape.com/eng/ssl3/draft302.txt>
12. T. Dierks, C. Allen, *The TLS Protocol Version 1.0*, RFC 2246, Jan. 1999. <www.ietf.org/rfc/rfc2401.txt>
13. E. Rescorla, *SSL and TLS: Designing and Building Secure Systems*, Addison Wesley, 2001, ISBN 0-201-61598-3.
14. B. Lenzlinger, A. Zingg, *Mini Web Server Supporting SSL*, Diploma Thesis, Zurich University of Applied Sciences Winterthur, October 2000. <www.strongsec.com/zhw/DA/Sna3_2000.pdf>
15. IPC@CHIP Online, Beck IPC GmbH, Wetzlar <www.bcl-online.de>
16. Security Group of the Zurich University of Applied Sciences in Winterthur (ZHW) <www.security.zhwin.ch>
17. Microelectronics Laboratory of the Scuola Universitaria Professionale della Svizzera Italiana (SUPSI) <www.lme.die.supsi.ch>
18. soft[net] Home Page <www.softnet.ch>
19. Embedded and Distributed Solutions Network (EDiSoN) <www.edison.ch>
20. Advanced Encryption Standard (AES) <www.nist.gov/aes>
21. Elliptic Curve Cryptosystem <www.certicom.com/resources/ecc_tutorial/ecc_tutorial.htm>
22. Intel Random Number Generator (RNG) <www.intel.com/design/security/rng/rng.htm>
23. *The Elliptic Curve Cryptosystem for Smart Cards*, A Certicom White Paper, May 1998 <www.certicom.com/resources/download/ECC_SC.pdf>
24. Certicom Home Page <www.certicom.com>