Implementing the TLS Protocol on a Bare PC

A. Emdadi, R. Karne, and A. L. Wijesinha
Department of Computer & Information Sciences
Towson University
Towson, MD 21252
{aemdadi, rkarne, awijesinha}@towson.edu

***Abstract*—A bare PC is an ordinary computer that runs applications without an operating system (OS). It thus provides immunity from attacks that target specific OS vulnerabilities while eliminating the OS overhead in a conventional system. We describe a novel implementation of TLS that runs on a bare PC. The TLS implementation differs from a conventional TLS implementation in that it is included within a bare PC application that manages its own CPU tasks, directly interfaces to the hardware, and communicates with network protocols without using a standard socket interface. Furthermore, the unique software architecture of a bare PC enables the TLS, TCP, and application protocols to be intertwined, thereby reducing the communication overhead compared to that of a conventional system. As an example, we give details of the internal architecture of a bare PC TLS Web server. We then illustrate intertwining of the TLS, TCP, and HTTP protocols. In particular, we show how a state transition model that represents events and actions in the TLS Web server is used to develop the intertwined protocol code. We also discuss the testing methodology, note some challenges with respect to implementing TLS on a bare PC, and outline the measures taken to address them.**

Keywords- Bare PC, Application Object (AO), protocol intertwining, Transport Layer Security(TLS), Web server.

I. INTRODUCTION

TLS (Transport Layer Security) is a widely used protocol for securing network communications on the Internet. Conventional implementations of TLS require the support of an Operating System (OS) or OS-kernel. This paper describes a novel implementation of the TLS protocol on a bare PC, which is an ordinary desktop or laptop computer that can run applications directly over the hardware (without using any form of an OS or OS kernel). In a bare PC, the minimal operating environment necessary for an application to execute on given hardware is included in the application itself.

The bare PC TLS implementation differs from a conventional TLS implementation in many ways. The main difference is that in a bare PC, the TLS, TCP, and application protocols are tightly coupled via protocol intertwining, which reduces communication overhead compared to that of a conventional system with an OS-controlled protocol stack. Other differences relate to the management of underlying CPU tasks by the bare PC application and the absence of a standard socket interface for network communication.

In particular, we consider the implementation of a bare PC TLS Web server and provide details of its software architecture and implementation. We also illustrate protocol intertwining with respect to the TLS, TCP, and HTTP protocols.

A discussion of the tradeoffs in using a bare PC versus an OS-based system to run applications is not within the scope of this paper. However, two advantages of a bare PC system may be noted. First, a bare PC, which does not use an OS, cannot be compromised by attacks that target vulnerabilities of a specific OS. Moreover, the self-contained bare PC code is smaller, simpler, and easier to analyze for security flaws than the code in an OS-based system. Second, bare PC applications perform better than their OS counterparts due to the elimination of OS overhead.

The rest of this paper is organized as follows. In Section II, we provide a brief overview of bare machine computing. In Section III, we review related work. In Section IV, we describe the implementation of TLS on a bare PC, using the example of a TLS Web server. We also discuss protocol intertwining, the testing methodology, and some challenges associated with implementing TLS on a bare PC. In Section V, we present the conclusion.

II. OVERVIEW OF A BARE PC SYSTEM

A bare PC runs a single Application Object (AO) [1] that contains a given small set of applications (usually one or two). Each AO, which resides on a portable storage medium such as a USB flash drive, is completely self-contained and has the necessary code to boot and execute on IA32 (Intel 32-bit) CPUs (it can also currently run in 32-bit mode on a 64-bit processor). The AO includes C++ interfaces to the hardware [2] and its own device drivers such as those for network interface cards/on-board chips and audio hardware if needed by the application. A simple menu-driven interface is available for configuring the AO.

Bare PC and OS-based systems differ significantly with respect to system call handling, API design, networking and user interfaces, memory, file, and process/task management, exception handling, and program loading, initiation and termination. In essence, data and control flow in a bare PC application are tightly coupled since the AO encapsulates the necessary protocols, operating environment (including interfaces to the hardware), and application logic. Thus, the AO programmer manages CPU and memory as there is no OS. Examples of bare PC applications include an HTTP-only

(i.e., non-TLS) Web server [3], email server [4], and VoIP softphone [5].

III. RELATED WORK

The latest version of the TLS specification is TLS Version 1.2 [6]. The main differences between this version and TLS Version 1.1 [7], which we have implemented, relate to the replacement of the SHA-1 and MD5 protocols with SHA256 due to the security weaknesses of the former. The current TLS version also allows a new cryptographic operation known as authenticated encryption with additional data encryption (AEAD). In [8] and [9] respectively, use of the HTTP/1.1 upgrade mechanism to initiate a secure connection on port 80, and HTTP over TLS with port 443 as the default are documented.

Many implementations of TLS are based on OpenSSL [10], which is a comprehensive TLS toolkit that runs on OS-based systems. In contrast, a small functional implementation of TLS whose goal is cryptographic verification is described in [11]. While their TLS server, like ours, is shown to interoperate with popular browsers, it implements an earlier version of TLS (1.0), and executes on .NET, which is an OS-based environment.

Most of the previous work on TLS is primarily concerned with improving performance. In [12], performance of a TLS Web server is improved by an operating system extension implemented in OpenBSD that reduces system calls, context switching, and data copying. The bare PC implementation also eliminates system calls and OS-related overhead, and allows all protocol layers to manipulate a single copy of the data (i.e., with zero data copying), but there is no OS present. The detailed study of TLS performance in [13] shows that RSA operations dominate cost, and that optimization efforts should focus on the TLS handshake phase as opposed to the data transfer phase. In [14], a technique to reduce server load by assigning expensive handshake operations to the client is described. Such techniques are designed and tested in OS-based systems and require many facilities provided by the OS. For example, hardware accelerators that are frequently used to improve TLS performance [13] require OS support. TLS on a bare PC system can be likewise optimized, but optimizations must be handled by the application since there is no OS. Since our current TLS implementation does not include such performance enhancements, we do not discuss bare PC TLS performance in this paper.

IV. TLS IMPLEMENTATION

A. Adapting the TLS Protocol

TLS on a bare PC is based on TLS Version 1.1 and currently supports AES encryption (with CBC block cipher mode), RSA key exchange, and a few essential alert messages including close notify. Since TLS on a bare PC always runs as part of an application that is encapsulated within an AO, it is easier and more convenient to describe TLS in the context of a specific application. In this paper, we only focus on the TLS implementation for a bare PC Web server. We do not consider a bare PC TLS client implementation.

We first give a brief overview of TLS in the context of the bare PC Web server. A detailed specification of TLS is given in [6, 7]. TLS consists of four protocols: Handshake Protocol, Alert Protocol, Change Cipher Spec Protocol, and the Record Protocol. TLS consists of two layers, where the first three protocols run on the Record Protocol, which itself runs on TCP. Application protocols such as HTTP, if protected by TLS, also run on the Record Protocol. TLS provides encryption, data integrity, and authentication of peer identity. The Alert Protocol comprises several messages that are classified as fatal or warning, and serve to handle errors or provide notifications. Alert messages are encrypted. The Change Cipher Spec Protocol uses a single encrypted 1-byte message with value 1 before using the newly negotiated security parameters.

TLS Version 1.1 also defines a secure pseudo-random function (PRF) that XORs a pair of data expansion functions based on HMAC with SHA-1 and MD5 as the underlying hash functions. The PRF produces any desired number of bytes and is used in generating the master secret from the exchanged premaster secret; encryption keys and MAC secrets for read and write operations; and the finished message. The Handshake Protocol is used to negotiate security parameters and set up a shared master secret. It establishes a TLS session that enables the application to securely exchange data over a TLS connection. Since session establishment is expensive, new connections may be set up within an existing session.

The TLS Handshake Protocol begins with an exchange of respective hello messages between the client and server. These hello messages are used to negotiate the key exchange, encryption and MAC algorithms to be used. The bare PC Web server supports only specific algorithms as noted above and requests for alternates are rejected. Next, the server sends its RSA certificate and the hello done message. The client then sends an RSA-encrypted premaster secret, which is processed by the server to compute the master secret and encryption and MAC keys. The handshake terminates with the exchange of the respective change cipher spec and finished messages by client and server. The finished messages, which are encrypted and authenticated, enable the client and server to verify that the security parameters established during the handshake are correct. Following the handshake, HTTP messages are fragmented into records of at most 2^{14} bytes that are AES-encrypted along with the MAC. An explicit per-record IV, CBC mode, and padding are used to protect the application data as specified in TLS Version 1.1. Application data is not compressed.

B. TLS Web Server Architecture

Fig. 1 shows the internals of the bare PC HTTP/TLS architecture for a Web server AO including the relevant handlers, objects, and CPU tasks. For ease of reference, each block in this figure is assigned a numeric label. Since our focus in this paper is on the TLS implementation on a bare PC, we do not provide a detailed description of other protocols or the Ethernet NIC driver that are needed by the Web server. We also omit details of upper-level processing that is different depending on whether a request is for a static

or dynamic Web page. For details concerning the design, implementation, and performance of a HTTP-only bare PC Web server (without TLS), see [3].

The single Web server AO residing on a bootable USB mass storage device, encapsulates the Web server application, the necessary protocols including TLS, the network interface card (NIC) driver, and the load and boot code used to start the bare PC (1) and launch the application. Since there is no OS in a bare PC, the NIC Driver (20) and its interfaces are directly accessible to the application.

When the AO is loaded, it starts the Main Task (2). This task, which runs whenever no other tasks are running in the machine, selects the next task to be run (3) through its multiplexor logic (4). For example, the Main Task runs the Receive Task (RCV) when an incoming packet (that has arrived on the link) has been placed in memory by the network interface hardware. Other tasks in this system include TLS Handshake tasks and HTTP tasks.

When the Receive Task runs, it reads the packet from the Ethernet buffer (19). If the packet contains a HTTP/TLS request, the respective Ethernet, IP, TCP, and TLS Handlers (17) process the packet in a single thread of execution and update the state in the TCP Table (TCB) (23). The TCB acts as a centralized state controller for all requests in the system. When the Web server AO is designed, the AO programmer does not include any delay in the above single thread of execution to process the packet. During this thread, a request may also send a response such as an ACK to the client. Once the received packet is processed and the appropriate response is sent, the Receive Task returns control back to the Main Task.

However, as described previously, before the application data can be protected by TLS, the TLS handshake protocol must be used to establish the security parameters for a TLS session between a client and the server. Thus, after the TCP connection to port 443 is established (requesting a TLS connection as specified in [11]), a TLS Task is popped from the TLS Task Stack (7) and inserted into a circular list (5). This TLS Task, which runs when it is scheduled by the Main Task, is only used for handshake messages; when the handshake is completed, the task will be returned to its stack pool (7). The entire TLS handshake process is modeled as a State Transition Diagram (STD) (12), which is part of the TLS Object (10). This object contains the necessary logic to perform all TLS operations.

After the TLS handshake is completed, a TLS flag will be set for a given TLS connection. This connection is terminated as usual when a TLS close_notify alert sent by the client is received (for a specification of server behavior related to incomplete closures, see [11]). Once a TLS connection is established, if a message containing data sent by the client is received, it will be processed by the respective handlers (17). The TCP handler invokes the services of the TLS handler to decrypt (15) and authenticate the packet. If the message contains a GET command, an HTTP Task will be popped from the HTTP Task Stack (6) and inserted into the circular list (5). This HTTP Task will run when it is scheduled by the Main Task. The HTTP Task processes the GET request, encrypts the HTTP packet containing the response data (14), attaches the MAC, and sends the packet. This is also done using a single thread of execution (17). The HTTP handler (18) contains the code that processes the GET request and gives it to the TCP handler. The latter requests the TLS handler to perform the necessary cryptographic operations. After this, the packet is returned to the TCP handler, which invokes the lower layer handlers (17) to send the packet.

Since outgoing data may require many packets with associated round-trip delays for each packet, this process will suspend for a period of time and return control to the Main Task after its single thread of execution. If a round-trip timer expires, or an ACK arrives, this process will be resumed. The process ends when all the data has been sent successfully to the client and its task will be returned to the HTTP Task Stack (6) upon completion. There can be a pool of TLS tasks (8) and HTTP tasks (9) running in the system at any given time. The HTTP Task uses the HTTP Object (11) and the associated STD (13) to implement its logic.

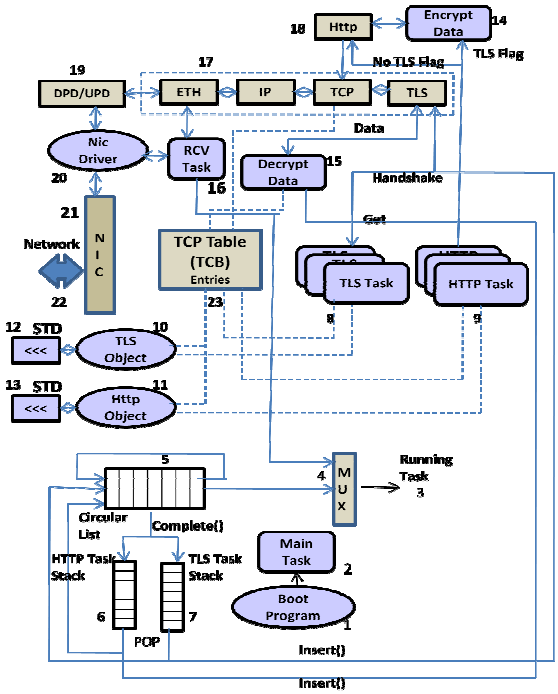

Figure 1. Bare PC TLS architecture for a Web server.

C. *Protocol Intertwining*

The bare PC TLS architecture in Fig. 1 exhibits tight coupling between the data and control flow of an application, which enables protocols, operating environment, and application logic to be encapsulated within a single monolithic AO. In particular, the TCP handshake is intertwined with the TLS handshake, while the TCP, TLS, and HTTP protocols are intertwined during the processing and transfer of encrypted data, and again during the closing. This intertwining is represented in the integrated message exchange for these protocols shown in Fig. 2. In the absence of an OS, the application itself controls the underlying CPU

tasks, while protocols that are implemented as part of the application communicate directly via their associated handlers.

During the design phase, the protocol intertwining shown in Fig. 2 and the corresponding events/actions were modeled using a state transition diagram. In case of the TLS handshake messages, there are three states TLS_START, SR_CLNT_HELLO, and SR_KEY_EXCH. Fig. 3 shows these states and the associated events/actions. The state transition diagram was used to develop the code for the TLS handshake. As an example, the C++ implementation of the SR_CLNT_HELLO state in Fig. 4 contains TLS code that illustrates intertwining with TCP.

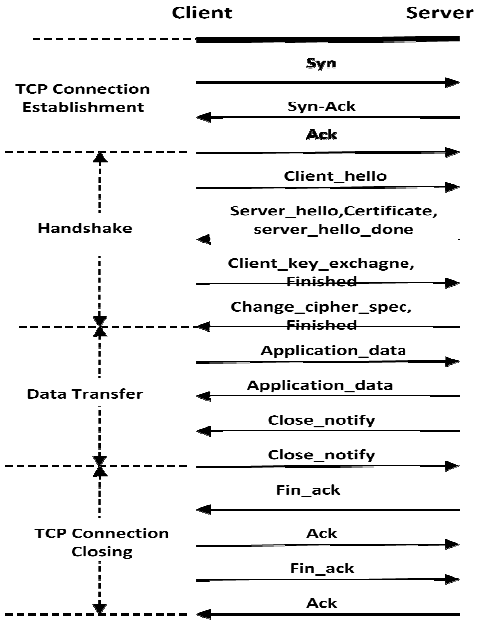

Figure 2. Intertwined protocol message exchange.

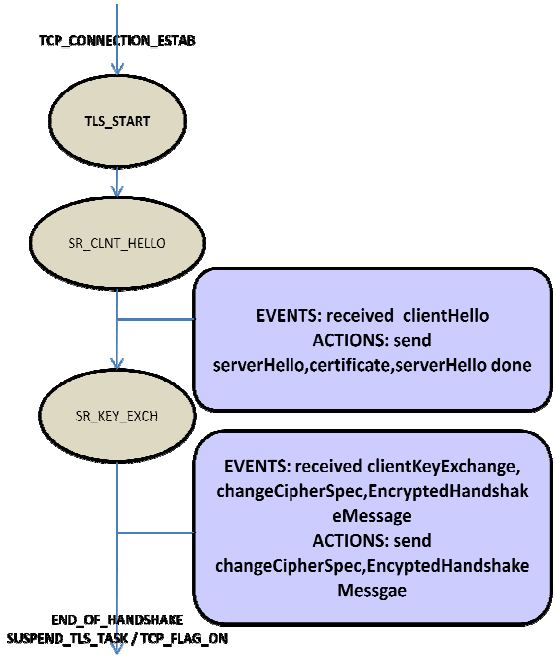

Figure 3. Bare PC TLS handshake states and events.

D. Testing

The bare PC TLS Web server implementation was tested with different file sizes and for a variety of client/server interactions using both the Internet Explorer and Firefox Web browsers. The exchanged messages were captured using Wireshark during the TLS handshake and the subsequent TLS-protected data transfer. The relevant data was then compared against data obtained for identical browser transactions with Web servers running on Linux and Windows to identify possible problems. We also analyzed and validated the intertwined protocol states to ensure desired behavior during the various phases of TLS operation. These tests verified interoperability of the bare PC TLS Web server with both browsers.

```
case TLS_START:     //2001
if  ((tcp.getTCBState(tcbrno) == ESTAB) &
(tcp.getTCBPacketArrived(tcbrno) == 1)) //client hello
{
tcp.resetTCBPacketArrived(tcbrno);
m1 = tcp.getTCBMsgPtr(tcbrno, length); //get pointer
flag = processMessage(m1);  //process message
for(int i=0; i <32; i++) //client_hello random bytes 11-43
clientHello[i] = m1[11+i];
for(i=0; i < 135; i++)
clientHelloNoRecord[i] = m1[5+i]; //capture client hello
if(flag==0)
{
formatTLSPacket(m2, tls1);//format outgoing packet
assignIntToChar(randTime, tls1->sh.rand.GMTTime, 0);//time
for (j =0; j < 28; j++)
randBytes[j] = tls1->sh.rand.random_bytes[j];
for (j =0; j < 868; j++)
serverMessagesNoRecord[j] = m2[j + 5];
//server hello, certificate, and server hello done
messageSize = 873;
retcode = tcp.updateTCBEntry
(tcbrno, (unsigned long)m2, messageSize, 0);
//update file size and pointer in the TCB
retcode = tcp.setTCBTempFlag
(tcbrno, TPUSH|TACK); //flags
retcode = tcp.setTCBTempSeqNum
(tcbrno, tcp.getTCBSNDNXT(tcbrno));//set seqnum
retcode = tcp.TLSSendN(tcbrno); //send message
retcode = tcp.setTCBSNDNXT(tcbrno, messageSize);
statec = SR_KEY_EXCH;  //Go to next state
retcode = tcp.setTCBTLSState(tcbrno, statec); //save the state
statec = tcp.getTCBTLSState(tcbrno);
} //end of flag==0
} //end of flag arrived and estab
break;
```

Figure 4. TLS code with protocol intertwining.

E. Challenges

The main challenges in implementing TLS on a bare PC stem from the inability to used OS-based components, libraries, and tools in software development on a bare PC. Application development on a bare PC is more tedious since standard support tools, debuggers, and environments are OS-dependent and cannot be used. Also, each application program inherits execution knowledge, which requires the AO programmer to be involved in the systems aspects of programming including resource and task management. Some challenges and the measures taken to address them are as follows:

1. testing and verifying operational correctness of cryptographic algorithms and protocols: we developed a Windows-based TLS tool for validating algorithms and protocol states.
2. generating certificates and verifying signatures: we developed a certificate tool for generating RSA keys, and constructing and signing X.509 certificates.
3. managing and allocating memory dynamically from within the AO, including the memory required for RSA calculations: we developed a memory management module for dynamic memory allocation.

V. CONCLUSION

The design and implementation of TLS on a bare PC differs significantly from the usual OS-based implementation that relies on a conventional TCP/IP stack. The streamlined software architecture enables the direct self-management of tasks by the TLS Web server application and facilitates efficient communication between the TLS, TCP and HTTP protocols by intertwining. A state transition diagram served as the basis to develop the intertwined protocol code. The TLS Web server implementation was tested and verified, and found to interoperate correctly with popular browsers.

REFERENCES

[1] R. K. Karne, K. V. Jaganathan, T. Ahmed, and N. Rosa, “DOSC: Dispersed Operating System Computing”, OOPSLA ’05, 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, Onward Track, ACM, San Diego, CA, pp. 55-62, Oct. 2005.

[2] R. K. Karne, K. V. Jaganathan, and T. Ahmed, “How to Run C++ Applications on a Bare PC” Proceedings of the 6th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, IEEE Computer Society, Washington DC, pp. 50-55, 2005.

[3] L. He, R. Karne, and A. Wijesinha, "The Design and Performance of a Bare PC Web Server", International Journal of Computers and Their Applications, vol. 15, pp. 100 - 112, June 2008.

[4] G. Ford, R. Karne, A. Wijesinha, and P. Appiah-Kubi, “The Design and Implementation of a bare PC Email Server,” 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC), pp. 480-485, Seattle, WA, July 2009.

[5] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, “A Peer-to-Peer bare PC VoIP Application,” Proceedings of the IEEE Consumer and Communications and Networking Conference (CCNC), pp. 803-807, IEEE Press, Las Vegas, NV, 2007.

[6] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," August 2008, RFC 5246.

[7] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," April 2006, RFC 4346.

[8] R. Khare and S. Lawrence, “Upgrading to TLS within HTTP/1.1,” May 2000, RFC 2817.

[9] E. Rescorla, “HTTP over TLS,” May 2000, RFC 2818.

[10] OpenSSL: The open source toolkit for SSL/TLS http://www.openssl.org.

[11] K. Bhargavan and R. Corin, “Cryptographically Verified Implementations for TLS,” 15th ACM conference on Computer and Communications Security, Alexandria, VA, USA, pp. 459-468, Oct. 2008.

[12] M. Burnside and A. D. Keromytis, **“**The case for crypto protocol awareness inside the OS kernel,” ACM SIGARCH Computer Architecture News 58. Vol. 33, pp. 58-64, March 2005.

[13] C. Coarfa, P. Druschel, and D. S. Wallach, “Performance Analysis of TLS Web Servers,” *ACM Transactions on Computer Systems* Vol. 24, pp. 39-69, Feb. 2006.

[14] C. Castelluccia, E. Mykletun, and G. Tsudik, “Improving Secure Server Performance by Re-balancing SSL/TLS Handshakes,”ASIAN ACM Symposium on Information, Computer and Communications Security, Taipei, Taiwan, pp. 26-34, 2006.