SECURE NETWORK CARD

Implementation of a Standard Network Stack in a Smart Card

Michael Montgomery, Asad Ali, and Karen Lu
Axalto, 8311 North FM 620 Road, Austin, Texas, 78726, USA

Abstract: This paper covers the philosophy and techniques used for implementation of a standard networking stack, including the hardware interface, PPP, TCP, IP, SSL/TLS, HTTP, and applications within the resource constraints of a smart card. This implementation enables a smart card to establish secure TCP/IP connections using SSL/TLS protocols to any client or server on the Internet, using only standard networking protocols, and requiring no host middleware to be installed. A standard (unmodified) client or server anywhere on the network can securely communicate directly with this card; as far as the remote computer can tell, the smart card is just another computer on the Internet. No smart card specific software is required on the host or any remote computer.

Key words: Internet; smartcard; network; SSL; TLS; TCP/IP; PPP; resource constraints.

1. INTRODUCTION

Smart cards have been in use for more than two decades now, but they have yet to achieve wide acceptance in mainstream computing. Smart card advantages such as security, portability, wallet compatible form factor, and tamper resistance make smart cards potentially useful for a wide variety of applications. However, these applications are hindered because of the mismatch between smart card communication standards and the communication standards for mainstream computing and networking.

For years, efforts have been underway to connect smart cards to the Internet. Early pioneers include the University of Michigan Webcard [1], Bull iSimplify [2], Guthery's GSM Web server [3], and the Gemplus prototype [4]. This paper expands on the years of work in this area, detailing some of the drawbacks of the approaches to date, and how many of these

drawbacks can be overcome by using a standard network stack. Finally, techniques are presented for implementing a standard network stack within the resource constraints of a smart card.

2. MOTIVATION

The current network smart cards have some key areas of weakness. By analyzing these weaknesses, we can better understand how to achieve our vision of widespread acceptance of smart cards into the network computing mainstream.

2.1 Security

Security is the most important aspect of most smart card applications. Much effort has gone into improving overall smart card tamper resistance and defenses against specific attacks. APDU communication can be protected by encryption. Challenge/response can verify trusted terminals.

Networking adds an extra security challenge [5]. There are currently two primary techniques for establishing network security with smart cards.

One technique is to write remote applications that encrypt data within the remote application as shown in Figure 1. The remote applications do not depend upon the network layer for security, but use an encryption scheme that is shared with the card applications. The drawback of this approach is that smart cards can only interact securely with remote applications that have been written or modified with smart cards in mind.

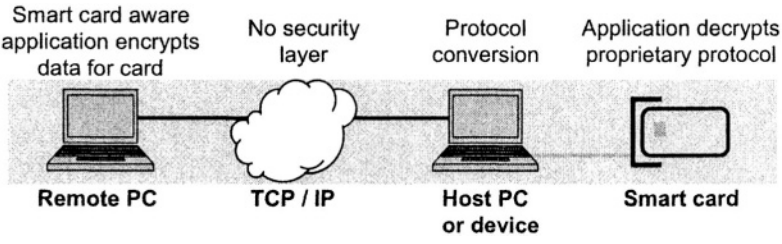

Figure 1. End-to-end security using card specific security protocols.

A second technique avoids this drawback, and allows the smart card to interact with applications that are not smart card aware. With this technique, the host computer establishes a secure network connection with the remote

application using standard SSL or TLS protocols, as illustrated in Figure 2. When the remote application sends data, the network layer automatically encrypts it before it is transmitted. The host then decrypts the data, packages it in APDUs, and can encrypt the APDUs before sending to the card if desired. The problem with this approach is that when the data in the host is decrypted, it is vulnerable to interception within the host computer. Since host computers are typically much more vulnerable to attack than smart cards, this becomes the weak link in the security chain.

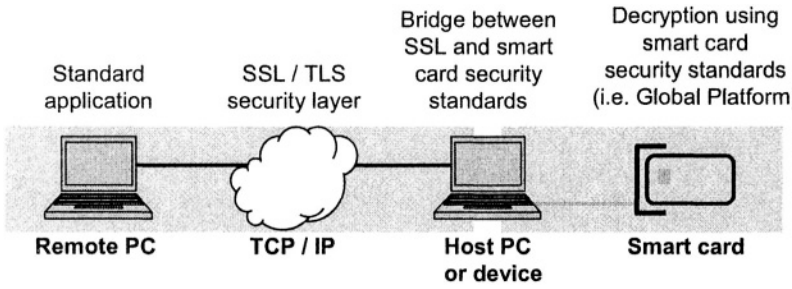

Figure 2. Bridging standard security layers in the host.

To avoid the problems associated with these two techniques, a first key requirement for a secure network card is to implement the TCP/IP network stack and SSL/TLS security layer inside the card, as shown in Figure 3. This allows the card to establish end-to-end security with a remote application that is not smart card aware, so that the communications are protected from all computers along the communications path, including the host computer. The security of the host is no longer an issue, allowing even an untrusted kiosk computer to be potentially used safely as a host.

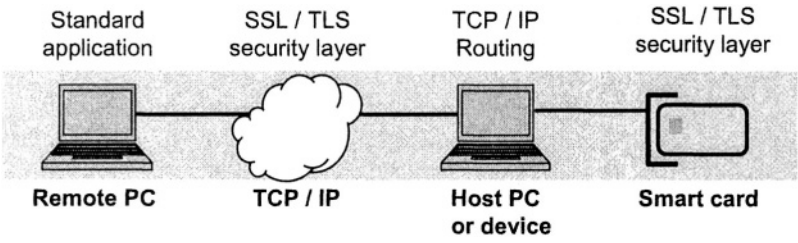

Figure 3. End-to-end security using SSL/TLS security within smart card.

2.2 Middleware

Thus far, all cards have required some middleware [5,6] to be installed on the host computer to facilitate the connection of the cards to the Internet. Typically, the middleware resides on the host computer, acting as a proxy for the card for establishing network communication, handling the network protocols, and repackaging the data to send APDUs to the smart card and receive the card responses. Sometimes the middleware handles the security layer as well, though this can lead to drawbacks as shown earlier. In addition, remote applications for accessing a smart card were sometimes built using a proxy/stub approach, requiring middleware on the remote computer as well.

Requiring middleware has several negative consequences. Middleware must be developed and tested for any platform to enable the use of the card. Middleware for various platforms such as Windows 98, Windows 2000, Windows NT, Windows XP, MacOS X, and Linux is expensive to develop and test, and even more expensive to maintain. Often platforms are excluded because of the high cost of middleware. Users dislike middleware; this is often one of the greatest barriers to user acceptance of a product. Users are now accustomed to plug-and-play operation, and reject products that do not use standard drivers built into the operating system.

Therefore, a second key requirement for a secure network smart card is to use standard interfaces and drivers that are built into most operating systems, so that no middleware is required. This allows a network smart card to freely move between computers, without having to install middleware every time a new host computer is used. This potentially enables host computers such as kiosks or corporate computers, where users may be forbidden to install middleware.

3. ARCHITECTURE

An architecture was selected based on our two key requirements: implement the TCP/IP network stack and SSL/TLS security layer inside the card; and use standard interfaces and drivers that are built into most operating systems, so that no middleware is required. This architecture is shown in Figure 4.

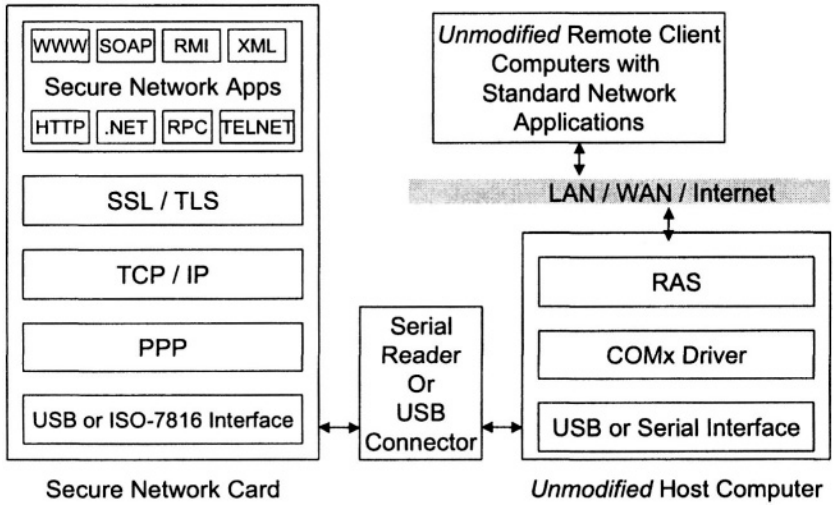

Figure 4. Secure network card architecture.

The secure network card contains a USB or ISO 7816 physical layer, a complete network stack consisting of PPP, TCP/IP, and SSL/TLS, and various network applications. If a USB interface is used, then a USB connector is used to connect to the host computer. If an ISO 7816 interface is used, a specialized smart card reader is used to convert this to full duplex serial or USB, and is connected to a serial or USB interface on the host computer.

The host computer can be any platform that is configured to permit network access from a serial or USB port. This includes most workstation, desktop, and laptop platforms including Windows, MacOS X, Linux, and Unix platforms, as well as some mobile palmtop and handset devices. In the case of Windows platforms, configuration is a simple task requiring less than 10 seconds using the New Connection Wizard (a standard utility that comes with all Windows operating systems) to specify a direct connection to another computer. The host is unaware that the computer being connected is a smart card; it treats the smart card as any other computer requesting a connection. Middleware or other smart card specific software is not required for any platform.

The host computer functions simply as a router to connect the smart card to the network, where it may be accessed by various remote computers. These remote computers are also unmodified, with no middleware or added software. A remote client or server anywhere on the network can securely

communicate directly with this card using standard network applications; as far as the remote computer can tell, the smart card is just another standard computer on the Internet. No smart card specific applications are required.

4. HOST OPTIONS

The main issue with the host was how to get from a physical layer compatible with a smart card to the TCP/IP layer. Once the TCP/IP layer was reached, standard network services were available.

The host could potentially use any physical interface. In practice, a physical interface using standard serial or USB protocols was needed, since these are the interfaces currently available on smart cards and readers.

There are several possible ways to get from serial or USB physical interfaces to the networking stack, while adhering to the requirement of no middleware or smart card specific software. Here are some options that were considered:

1. Serial → PPP (Point-to-point protocol) → RAS (Remote Access Server)
2. USB (Universal serial Bus) → encapsulated Serial → PPP → RAS
3. USB → RNDIS (Remote Network Device Interface Specification)
4. USB → Ethernet over USB → Ethernet drivers

Option 1 is standard for dial-up networking. Option 2 is the same as option 1, except that it uses an encapsulation protocol to create a virtual serial port. There are many drivers that support serial-over-USB encapsulation[1]. Option 3 using RNDIS suffers from two major drawbacks: there are no standard non-Windows implementations, and it has a heavy device footprint. Option 4 would likely be ideal, but at this time, the standard is still under development by the USB communications working group.

Options 1 and 2 were used. This required no software for the host computer. The computer need only be set up to accept an incoming connection over the appropriate serial port, a very simple task for Windows, Macintosh, Linux, and most other computing platforms.

[1] The FTDI driver (www.ftdichip.com), which is built into Windows, is one of many drivers that support serial encapsulation over USB.

5. PROTOCOL STACK: PPP, TCP/IP

The secure network card implements a TCP/IP protocol [7,8,9,10] stack in order to be a standalone Internet node. Depending on the I/O characteristics of the physical connection with the host device, the card may have various link layer protocols.

As listed in section 4, Option 1, one approach to connect the card to the Internet is to connect a smart card reader to the serial port of a PC with Internet connectivity. All Windows platforms define a direct serial cable connection as one of the modem types. RAS, which is a standard part of Windows platforms, provides services that enable a dial-up device, in this case the smart card, to connect to the Internet. RAS uses PPP [11,12] to communicate with the dial-in device that acts as a client and initiates the PPP negotiation. RAS can acquire an IP address on behalf of this client via DHCP. After the PPP connection is established with the client, RAS merely acts as a router between the Internet and the client. With this approach, the smart card has its own IP address, and can act as an autonomous node on the Internet.

With a standard full-duplex serial I/O, a device can connect to the serial port (COM port) of a PC and establish connection with RAS to gain Internet access without loading any additional software on the PC. Since RAS speaks PPP, for a smart card to connect to PC via serial connection, the card needs to implement PPP in addition to TCP/IP. However, smart card communication standards (ISO 7816, or ISO 14443) specify half-duplex I/O, while PPP presumes a full duplex channel. (Current USB smart cards are also based on ISO 7816 APDUs and use a half-duplex logical protocol.)

To bridge the current gap between available half-duplex smart cards, and the required full-duplex behavior for supporting PPP, a token passing protocol was devised. This APDU-based protocol allows peer-to-peer I/O in a logical full-duplex mode, where either the card or host can initiate I/O.

A smart card reader was developed which implemented this token passing protocol with both serial and USB host interfaces. Thus, the encapsulated PPP serial data was recovered from the APDUs and presented to the host over a standard serial interface (option 1) or encapsulated over a USB interface (option 2). Ideally, the USB interface should be built into the card, so that neither the reader nor PPP nor APDUs are required (option 4). Once the Ethernet over USB standard is established, this would become the preferred standard to implement in a USB smart card.

6. PROTOCOL STACK OPTIMIZATIONS

Smart cards have limited RAM and non-volatile memory compared to most computers running network stacks. Current cards may have up to 8K bytes of RAM, and up to 512K bytes of non-volatile memory. To fit the standard protocol stack of PPP and TCP/IP in the limited resources of a smart card, the following optimizations were done. These optimizations affect both the design and implementation of these protocols in a secure network card.

6.1 Protocol Feature Subset

To conserve memory resources, only those features of PPP and TCP/IP were implemented that are essential for making a smart card an independent node on the Internet. These features are listed below:

- The PPP layer supports dynamic IP addressing and AHDLC processing. It has LCP and IPCP finite state machines for link/network layer negotiation. However, it does not support all PPP options.

- The IP layer processes basic IP datagrams, but currently does not support fragmentation.

- The TCP layer provides reliable transmission for multiple connections. It supports PUSH and delayed ACK for interactive data flow, timeout, round trip time (RTT) measurement, and retransmission time out (RTO) computation using Jacobsen's algorithm.

6.2 Buffer Management

Prudent buffer management is an integral part of the design and implementation of standard protocol stacks like PPP and TCP/IP. Since I/O buffers pose heavy RAM requirements, it is even more critical to have an optimized design for resource-constrained devices like smart cards. Some of the key techniques used in the secure network card to optimize use of the limited RAM resources are listed below:

Chained Buffer: To allow flexibility in use of small as well as large data, a chained-buffer mechanism is used to store and process data. Similar mechanisms have been used in various BSD-style TCP/IP implementations and some embedded developments [13,14,15]. The details of chained buffer data structures vary from one approach to another, but the basic mechanisms

are similar. The design used in the secure network card is based on the Packet Buffers *(pbufs)* defined in lwIP [14]. Figure 5 shows the chaining of *pbufs.*

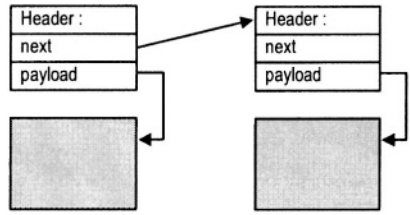

Figure 5. A pbuf chain with two pbufs.

The advantage of a chained buffer approach over fixed length buffer is conservation of memory. With a fixed buffer, memory is wasted by upfront allocation of the largest possible buffer. Most of this allocated memory lies unused during normal processing. With a chained buffer, a new buffer is allocated from a pool on an as-needed basis and then "chained" to existing buffer(s) to give a logically contiguous data array.

In a secure network card, all upper layer protocol modules, including PPP, IP, and TCP, share the *pbuf* chain allocated by the AHDLC module for input processing. During PPP negotiation, the PPP module uses *pbufs* for input and output. *A pbuf* is dynamically allocated from a pool of buffers; it is released after the input packet is processed or the output packet is sent.

AHDLC Processing: To optimize AHDLC processing, a technique of in-place handling of incoming frames is used. Escape characters in each frame are handled without allocating a separate buffer. Similarly, data can flow from the smart card hardware interface directly into application buffers without any additional copy. Figure 6 shows a typical processing of an input AHDLC frame containing a PPP LCP configuration request with no options.

Initial Input buffer

```
7E EF 7D 23 C0 21 7D 21 7D 21 7D 20 7D 24 D1 B5 7E
```

Input buffer after in-place AHDLC processing

```
EF 03 C0 21 01 01 00 04 D1 B5
```

Figure 6. Typical in-place processing of an AHDLC frame data.

Socket Interface: A traditional BSD socket interface has copy semantics. This is because the application and the system usually reside in different protection domains. Since data must be copied during a call across such domains, a socket APIs effectively doubles the memory requirement per packet. Several options have been proposed for zero-copy I/O mechanism [16,17,18,19]. We follow a similar approach where the data buffer, instead of the data itself, is transferred between the communication layer and the application layer. This design has the advantage of reduced memory usage and increased I/O performance.

7. SSL/TLS LAYER

Secure Sockets Layer (SSL) and its successor Transport Layer Security (TLS) are the de facto standards for securing Internet web communication. Implementing SSL in a smart card can improve the card to support end-to-end network security with any unmodified client on the Internet. However, current smart cards do not support SSL implementations. This is partly due to the absence of an underlying reliable bi-directional transport layer, and partly due to the heavy memory demands imposed by the SSL protocol stack and the cryptographic computations that are required by the protocol. Description of the SSL/TLS protocol stack is outside the scope of this paper. SSL/TLS are open specifications and can be found in various books and RFCs [20,21,22,23].

The secure network card overcomes these challenges by various optimization techniques that reduce the RAM utilization of the SSL layer to less than 1.5 Kbytes. These design optimizations can be broadly divided into four categories.

7.1 SSL Feature Subset

Due to the limited resources of smart cards, the first challenge was to select a minimal feature subset from the SSL/TLS protocol specification without compromising either the specification or compatibility with existing standard clients, the mainstream web browsers. Three browsers were considered for gauging this compatibility: IE 6.0, Netscape 7.0, and Mozilla 1.5. A close examination of the SSL/TLS protocol and the selected browsers [24] led to the following decisions regarding feature subset:

- There was little value in mixing multiple protocol versions - SSL 2.0, SSL 3.0, and TLS 1.0 - in the same implementation. Instead two separate

implementations were completed: one using TLS 1.0 that can be used for all future work, and one using SSL 2.0 that can be used for extremely low end devices without cryptographic accelerators.

- Instead of supporting multiple cipher suites, the design focused on a single one, TLS_RSA_WITH_DES_CBC_SHA that was available on all mainstream browsers. It uses RSA for authentication and key exchange, DES for encryption, and SHA-1 for digest. This design allowed a fast streamlined implementation of a single cipher suite on the smart card while still providing hooks to add additional cipher suites the in future.

7.2 Stack vs. Heap

To better manage the limited RAM resources, stack size and depth were kept to a minimum. Instead, all memory required for maintaining TLS context state and for performing cryptographic operations was allocated on the heap. This was done by a customized heap management subsystem so that buffers could be dynamically allocated and de-allocated as needed. The TLS implementation requested buffers from this heap and then freed them once the task was complete. The same RAM space could then be used for other operations. Since the TLS state machine knew exactly when it was safe to free a buffer, premature and accidental buffer release was not an issue.

7.3 Buffer Reuse

As an additional optimization of dynamic heap management, an allocated buffer is used in more than one context within the same allocation-release cycle. This eliminates the overhead of releasing a buffer and then allocating another one from the heap. The TLS implementation in the secure network card carefully uses this technique. Some examples are listed below:

- During the full-handshake phase of TLS negotiation [21], the pre-master secret and master secret values are stored in a single common buffer. Although both values are critical during the handshake, they are not used concurrently.

- While processing the Client-key-exchange message during the TLS handshake, the value of the encrypted pre-master secret is not copied to a separate buffer. Instead, it is kept in the same I/O buffer used for processing all incoming TLS records.

- When performing DES encryption and decryption, the same buffer is used for input as well as output.

While the buffer reuse technique reduces the RAM footprint in most cases, it is not viable in all scenarios. For example, during the TLS handshake process a lot more information needs to be kept in memory than the allocated RAM pool will allow. In these situations, we follow an approach that is unique to smart card development. The unused data is swapped to non- volatile memory (NVM), which is much more abundant in the smart card. The RAM buffer is reassigned to hold some other data and perform a different computation. Once this computation is complete, the saved data is reloaded from NVM and the RAM context is restored to its original state. One example of this approach occurs when performing RSA decryption during TLS handshake.

7.4 Application Interface

Following the completion of the TLS handshake, the smart card has established a set of session keys that can be used to encrypt and decrypt application data. It can now begin to securely exchange data with an unmodified client on the Internet. For a resource-constrained device, this secure application data exchange presents a unique challenge. This is due to the limited size of the receive buffer into which TLS records are written after being read from an underlying socket layer. Figure 7 explains this application level call to the TLS layer.

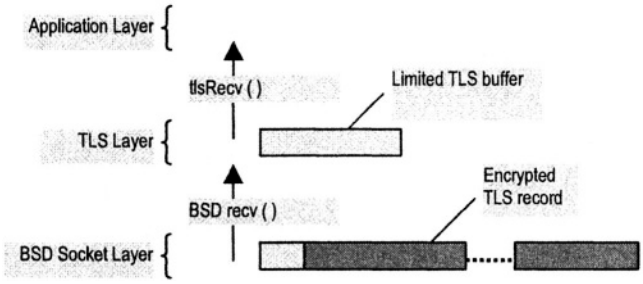

Figure 7. Reading a larger TLS record using a small TLS buffer

In a secure network card, the TLS receive buffer is set to 200 bytes. However, browsers can send TLS records of much larger sizes. Even the simplest of HTTP GET requests from a browser can be a single TLS record of 500 bytes or more. Since symmetric encryption and MAC are applied

over the complete TLS record, the challenge is to use a 200-byte TLS receive buffer to process TLS records of much larger sizes. This processing involves decryption as well as verification of the MAC.

We solve this problem by using two distinct approaches, each with its own advantages.

The first approach is a *performance-critical* approach. In this approach, the TLS record is read in blocks of 200 bytes or less. As each block is read, incoming data is decrypted, and the corresponding plain text data is passed to the application layer. In addition, the MAC context and initialization vector are also updated. When the final block of data in the TLS record is read, we perform an additional step of verifying the MAC over the complete TLS record. If this MAC fails, an error is flagged. With this approach, the application layer gets data as soon as it is read, without having to pay the penalty of larger RAM buffers. However, since MAC verification is not possible until the entire TLS record has been processed, any errors in secure transmission are not flagged until we read the entire record. In several applications, this slight delay in receiving a MAC error is acceptable, particularly if this behavior is requested to improve performance.

The second approach is the *error-critical* approach and is illustrated in Figure 8. In this approach, an application can mandate that no application level data be passed to it unless the MAC has been verified. To achieve this behavior we successively read the TLS record, in chunks of 200 bytes, and write it to NVM. Once the complete TLS record is read, we use the NVM buffer to decrypt data and then verify the MAC. If the MAC succeeds, the requested amount of data is passed to the application. On subsequent read calls, the remaining data in NVM is passed directly to the application without any need for decryption or MAC verification.

In this approach, the first application level read call is slow, but subsequent calls for data in the same TLS record are much faster. Overall, this provides a more secure application interface.

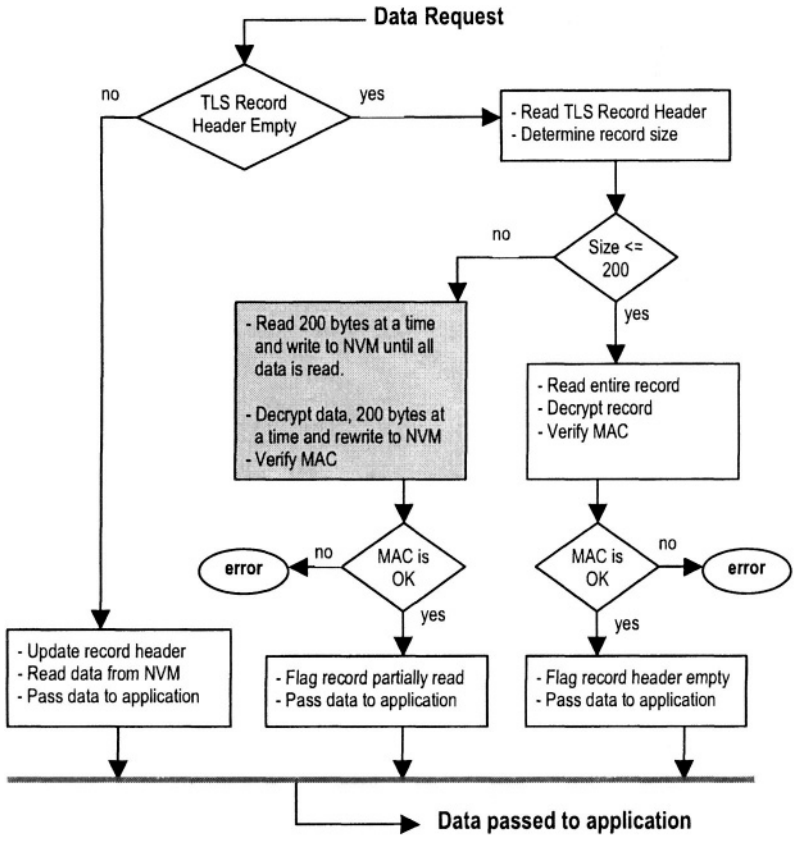

Figure 8. Error-critical design of reading TLS Record with limited I/O buffer.

8. PROTOTYPE AND APPLICATIONS

This project was originally implemented as a Windows simulation and an ARM smart card simulation. SSL/TLS and several applications were developed and simulated. For the prototype card, Samsung Jumbo was chosen to avoid masking. This had the drawback of having no crypto processor; establishing a TLS connection on the prototype took about 15 seconds. So SSL 2.0 with a shorter key length was implemented, to reduce the connection time to less than 1 second. With either TLS or SSL, once the connection was established, the performance was acceptable: pages would load across the network without significant lag time.

Performance varied significantly between the various browsers tested: Internet Explorer, Netscape, and Mozilla. Mozilla offered the best overall performance. After a production card is completed, detailed performance benchmarks will be run and published.

Applications loaded in the prototype cards included a web server/agent, Telnet server, and a mini-shell. Many services were loaded into the card that were accessible via HTTP, including a secure stock trading service, a secure email/encryption service, a secure e-commerce service, and a secure ticket service. Total NVM footprint was 76K Code and 34K Data. This left 412K available for other applications and services to be loaded. The RAM footprint was carefully optimized to just under the 6K available. These applications/services were demonstrated at Cartes 2003 and CT/ST 2004.

The smart card SSL/TLS library provides a simple application level interface that can be used by on-card applications to establish secure end-to-end network connections with any remote unmodified clients on the Internet. New application frameworks such as .NET, SOAP, and RMI can be easily added to enrich the application versatility.

The secure web server implemented in the prototype card can serve both static and dynamic HTML content. Static content is supported by reading the requested file from the card file system. Dynamic content is supported by invoking the requested application through a CGI interface and redirecting the results to the browser. The mini-shell is accessible through Telnet or HTTP, providing a very powerful way of interacting with the card.

9. CONCLUSIONS

The technology presented in this paper enables smart cards to participate as first-class citizens on the Internet within the established infrastructure, with no host or remote application changes required to accommodate smart cards. With this technology, smart cards are like any other computer on the Internet, while providing portability, enhanced security, and tamper resistance. Internet applications or services can be migrated to a smart card, increasing the security of critical information such as certificates, private keys, and passwords, while maintaining compatibility with existing clients. With smart card deployment unshackled from the infrastructure, the barrier to entry for smart cards in mainstream applications is removed. We foresee this technology triggering an unprecedented growth in deployment of smart card applications for the Internet.

REFERENCES

1. Rees, J., and Honeyman, P. "Webcard: a Java Card web server," Proc. IFIP CARDIS 2000, Bristol, UK, September 2000.
2. Urien, P. "Internet Card, a smart card as a true Internet node," Computer Communication, volume 23, issue 17, October 2000.
3. Guthery, S., Kehr, R., and Posegga, J. "How to turn a GSM SIM into a web server," Proc. IFIP CARDIS 2000, Bristol, UK, September 2000.
4. Muller, C. and Deschamps, E. "Smart cards as first-class network citizens," 4th Gemplus Developer Conference, Singapore, November 2002.
5. Itoi, N., Fukuzawa, T., and Honeyman, P. "Secure Internet Smartcards," Proc. Java on Smart Cards: Programming and Security, Cannes, France, September 2000.
6. Urien, P. "Internet smartcard benefits for Internet security issues," Campus-Wide Information Systems, Volume 20, Number 3, 2003, pp. 105-114.
7. Postel, J. "Internet Protocol," RFC 791, September 1981.
8. Postel, J. "Transmission Control Protocol," RFC 793, September 1981.
9. Socolofsky, T. "A TCP/IP Tutorial," RFC 1180, January 1991.
10. Almquist, P. "Type of Service in the Internet Protocol Suite," RFC 1349, July 1992.
11. Simpson, W. "The Point-to-Point" Protocol (PPP)," RFC 1661, July 1994.
12. Carlson, J. "PPP Design, Implementation, and Debugging," second edition, Addison-Wesley, 2000.
13. Wright, G.R. and Stevens, W.R. "TCP/IP Illustrated, Volume 2," Addison-Wesley professional Computing Series, 1995.
14. Dunkels, A. "IwIP – A Lightweight TCP/IP Stack." More details are available at http://www.sics.se/~adam/lwip/.
15. Lancaster, G., et al. uC/IP (pronounced as meu-kip) is an open source project to develop TCP/IP protocol stack for microcontroller. It is based on BSD code. For details, see http://ucip.sourceforge.net/.
16. Chihaia, I. "Message Passing for Gigabite/s Networks with Zero-Copy under Linux," Diploma Thesis Summer 1999, ETH Zurich.
17. Pai, V.S. and Druschel, P. and Zwaenepoel, W. "IO-Lite: A Unified I/O Buffering and Caching System," Rice University.
18. Thadani, M. N. and Khalidi, Y.A. "An Efficient Zero-Copy I/O Framework for Unix," SMLI TR-95-39.
19. Abbott, M., and Peterson, L. "Increasing network throughput by integrating protocol layers," IEEE/ACM Transactions on Networking, 1(5):600-610, October 1993.
20. Freier, Alan O., et al. "The SSL Protocol, Version 3.0," Internet Draft, November 18, 1996. Also see the following Netscape URL: http://wp.netscape.com/eng/ssl3/.
21. Dierks, T., Allen, C., "The TLS Protocol, Version 1.0," IETF Network Working Group. RFC 2246. See http://www.ietf.org/rfc/rfc2246.txt .
22. Elgamal, et al. August 12, 1997, "Secure socket layer application program apparatus and method." United States Patent 5,657,390.
23. Rescorla, E., SSL and TLS, "Designing and Building Secure Systems," 2001 Addison-Wesley. ISBN 0-201-61598-3.
24. Goldberg, I., and Wagner D., "Randomness and the Netscape Browser," Dr. Dobbs Journal, January 1996.