A "Bump in the Stack" Encryptor for MS-DOS Systems

David A. Wagner*
daw@cs.berkeley.edu
U. of California at Berkeley
Berkeley, CA

Steven M. Bellovin
smb@research.att.com
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

Most implementations of IP security are deeply entwined in the source of of the protocol stack. However, such source code is not readily available for MS-DOS *systems. We implemented a version using the packet driver interface. Our module sits between the generic Ethernet driver and the hardware driver; it emulates each to the other. Most of the code is straightforward; in a few places, though, we were forced to compensate for inadequate interface definitions.*

1 Introduction

The Internet Engineering Task Force (IETF) is in the process of adopting standards for IP-layer encryption and authentication (IPSEC) [3, 1, 2]. Not surprisingly, most of the original implementations are being developed for various flavors of the UNIX[1] operating system. While this is good—indeed, we are primarily UNIX system users ourselves—much of the world feels differently, and prefers MS-DOS[2] systems. For many reasons, it seemed desirable to develop an IPSEC implementation for a MS-DOS system.

Our immediate need was for an "E.T. call home" version of IPSEC. We did not need general functionality, and most of the conversations would be between a laptop running MS-DOS and our firewall. This allowed us to make a number of simplifications, especially in the area of key management (Section 5).

The immediate problem we faced, of course, was that we did not have source code to the TCP/IP stack we were running (PC/TCP 3.10[3] from FTP Software). Accordingly, we chose to implement our IPSEC module as a device driver, entirely below IP (Figure 1). This also gave us a considerable degree of independence from the precise version of the protocol stack; so long as the interface remains reasonably constant, our implementation should continue to work.

PC/TCP supports three different device driver interfaces: NDIS, ODI, and the packet driver. NDIS is from Microsoft, and will probably be the dominant standard in the future. ODI was developed by Novell for their network products. John Romkey (then of FTP Software) invented the packet driver [4]. Despite their internal differences, all three have the same

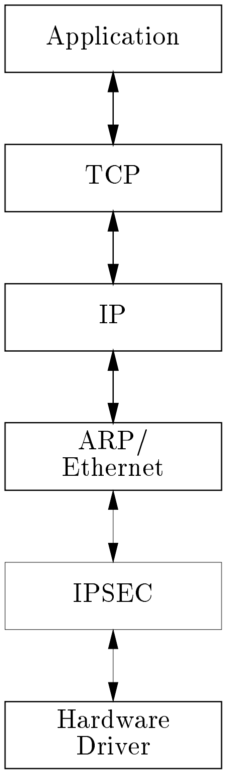

Figure 1: The protocol stack with IPSEC included. This is an implementation diagram; the IPSEC module includes some IP functionality.

*Work was done while at AT&T Bell Laboratories.
[1] UNIX is a registered trademark of X/Open.
[2] MS-DOS is a registered trademark of Microsoft.
[3] PC/TCP is a trademark of FTP Software.

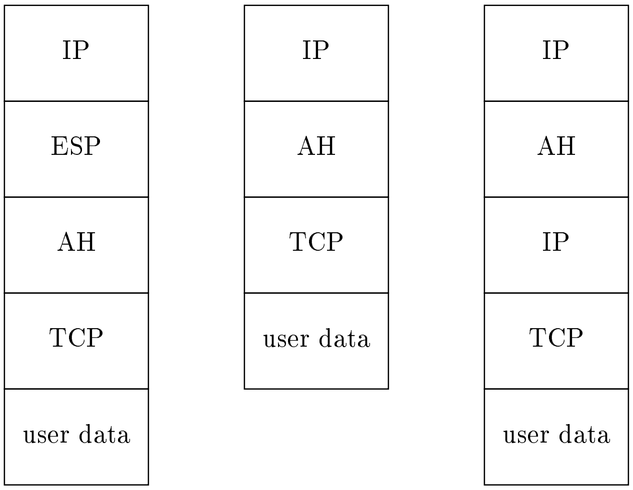

Figure 2: The layout of different types of secure IP packets.

purposes—to permit network drivers to be developed independently of the the protocol stacks, and to support multiple stacks—and we could have used any of the three. We opted to use the packet driver interface because source code was available for a number of drivers, and such samples are always valuable, even with the clearest of specifications. But the techniques we used are applicable to almost any such driver, even for radically different operating systems.

Other TCP/IP stacks support the packet driver interface as well, though sometimes only through an NDIS-to-packet driver shim layer. Our implementation will probably work with these as well, though perhaps with minor changes; there are a large variety of calls to the interface, and we have only implemented the ones we actually needed. (Indeed, as is discussed in Section 4, we wish that PC/TCP used more of the interface.)

2 The IP Security Architecture

IP security is implemented as a set of protocols that sit above IP. These protocols are used for authentication only (AH), encryption (ESP), or IP within IP (IP-IP). Each of these headers contains a field denoting the type of the next protocol, as shown in Figure 2.

Security services can be provided from host to host, host to gateway, or gateway to gateway. In the latter two cases, the nested IP header is used to denote forwarding to the ultimate destination.

Each security header contains a *Security Parameter Index* (SPI), sometimes known as a *Security Association Identifier*. The SPI is an arbitrary integer; it is used as an index to a table with keying information, cryptographic algorithm selections, etc. The receiver specifies the SPI to be used for packets sent to it; a typical conversation will have two SPIs, one for each direction.

It is up to each host to define its security policy. That is, it must specify the necessary grade of secrecy and/or authentication that is required when talking to any of its peers. No security at all is perfectly valid, and will long be the default case for most machines. In general, applications—and in particular servers—need to know the security properties of their connections; for our setup—manually controlled, and effectively client-only—this is not an issue.

A variety of cryptographic algorithms are defined [9, 8]; for our prototype, we have implemented just one, an authentication-only mechanism using MD5 [12] and a shared secret key. Future versions will likely include DES [11] and triple DES [7].

3 Implementation Details

We implemented our IPSEC module as a *Terminate and Stay Resident* (TSR) module. Such programs are not deleted from main memory when they exit; instead, the memory area remains allocated and the contents undisturbed.

The module is purely a packet driver from above. When talking to the real hardware packet driver below, it appears to be the generic Ethernet driver. Calls that are not relevant to IPSEC, such as ARP transmissions or receptions, are passed straight through. Although in principle any sort of device driver could live below IPSEC, our current implementation assumes an Ethernet driver. An obvious extension would be PPP compatability, though encryption interacts poorly with modem-based compression and the authentication header would interfere with TCP header compression [5].

3.1 Output Processing

When the IPSEC module receives a packet from above, it must look at the destination address, build an IPSEC header, perform the necessary security operations (authentication and/or encryption), determine the IP address of the endpoint of the cryptographic association, and re-invoke IP to route the packet. A device driver does not have that luxury; it cannot hand IP a packet as if it were a transport protocol. Accordingly, our module emulates IP and builds its own new IP header.

With one exception, this emulation is not particularly difficult; IP is a simple protocol, especially for transmission. The one difficult matter is routing: how do we find the IP address of the next hop. A very general IPSEC implementation might have to solve this problem; in reality, though, there is a simple answer: we assume that the new endpoint is along the same path as the ultimate destination, and hence shares the same next hop. If we are not using tunnel mode, the situation is even simpler: the new endpoint will be the same as the ultimate destination. We therefore make the simplifying assumption that the original next-hop address is still valid, and do not perform another routing operation or ARP query. An alternative would be to add that information to the keying table; this would handle the oddball cases, albeit at the expense of more configuration overhead.

The biggest problem in output processing comes when the new packet is larger than the MTU of the underlying device driver. This is discussed in Section 4.

3.2 Input Packet Processing

When an input packet arrives, the source address is examined to see if security is required when talking to that peer. If such filtering were not done, an enemy could inject unauthenticated packets into what should be a secure connection. Valid packets, or packets from a source for which security is not configured, are passed up the stack.

If the IP layer will forward packets—that is, if it will act as a router—there is a subtle danger to be wary of. Assume that an enemy sends some host `H` a packet claiming to be from `H`, but addressed to a different host. Upon forwarding, the packet could be treated as if it were generated on `H`, and will be given `H`'s cryptographic fingerprint. This allows the attacker to forge cryptographically authenticated packets from `H` to any of its peers. To defend against this attack, an IPSEC module must examine the source and destination addresses of all packets it receives. This is especially difficult if nested tunnel headers are used.

A better solution would be to ensure that cryptographic processing takes place only on packets originating on the local machine. However, this cannot be done with any sort of outboard encryptor, whether implemented as a separate box or via our techniques. It would seem, then, that routers—which we define as *any* IP stack that will forward packets, regardless of the stack's primary function—should not use outboard cryptographic modules for protection, unless the address-checking described above is implemented.

3.3 Policy Management

The behavior of IPSEC is controlled by two tables, the policy table and the security association table. The two concepts are separate, and should not be confused. Ultimately, the security association table will be controlled by a key management module; the policy module, though, is an administrative concept, and will always be specified manually. Both tables are read from files at system startup time; either can be changed at any time via a utility program.

The policy table specifies what classes of transforms should be applied to outgoing packets, and what is expected on incoming packets. Authentication can be configured for individual hosts or for groups of hosts. Separate policies can be set for input and output; it is thus possible to send unauthenticated packets to some destinations, but require that the other end send only authenticated packets. The default behavior for both transmission and reception can be specified independently.

The current prototype has a rather simplistic implementation. A production version will need more clearly defined semantics for the search order. It will also need port number matching as well, to permit key management exchanges in the absence of an existing key.

The security association table is much more straightforward. For each destination, a security transform, SPI, and key are specified. For each SPI, the transform, key, and IP address are given. The latter field must match what is in the received packet. Currently, we do not attempt to support multicast packets.

4 Fragmentation and MTU

Security has its costs; apart from the obvious issue of CPU time, the security headers take up space in the packet. This is problematic, as TCP will often try to send maximum-size packets, leaving us no headroom. An ideal solution would be to tell IP and TCP that the device has a smaller MTU than the 1500 bytes prescribed for Ethernet. Indeed, the packet driver specification includes such a call. Unfortunately, PC/TCP doesn't use it.

The solution[4] we implemented was to use the *Path MTU* mechanism [10]. When a too-large packet is received from IP, the IPSEC module crafts an ICMP error message indicating the proper maximum message size. TCP—which has no way of knowing that the message was generated locally, rather than from a distant router—will then reduce its idea of the maximum packet size and retransmit. This exchange will take place for each new connection (and can even happen at intervals during the lifetime of an existing connection), but the amount of overhead isn't high. Unfortunately, as of this writing the code isn't working properly yet; it is unclear if it is a bug in our code or in PC/TCP.

The alternative would be to have IPSEC fragment large outgoing packets. While this isn't difficult per se, it creates two other problems: the general difficulties of fragmented packets [6], and the need to reassemble such packets in the receiving IPSEC module.

The latter is more of a nuisance, but one that a general implementation cannot avoid. Fragmentation can occur anywhere along the path, not just on the sending side; this means that a receiver must be prepared to reassemble fragments, since it is not possible to authenticate a partial packet. Ideally, we could let IP do the reassembly—it already has the ability to do so—and pass the complete datagram on to IPSEC for decryption; unfortunately, this cannot be done in a clean fashion (Figure 3). While a request for IPSEC services is denoted by a special protocol number in the IP header, and IP is generally able to support an arbitrary number of higher-level protocols, this interface is *not* open on PC/TCP. There is no standard comparable to the packet driver for this connection, and we did not wish to reverse-engineer the existing programs that used it (i.e., `ping`).

If we really wanted to avoid the need for fragment reassembly code, we could try to send Path MTU messages to the remote host. But this is probably fruitless; at this time, very few hosts would honor such a request.

A third possibility would be to ignore the issue; most connections are off-net, which generally implies an MTU of 576 bytes or less, and most modern links support larger packet sizes. But such assumptions tend to bite you when you least expect it, and we still need to authenticate local connections. Furthermore,

[4] This idea was suggested by Frank Kastenholz of FTP Software.

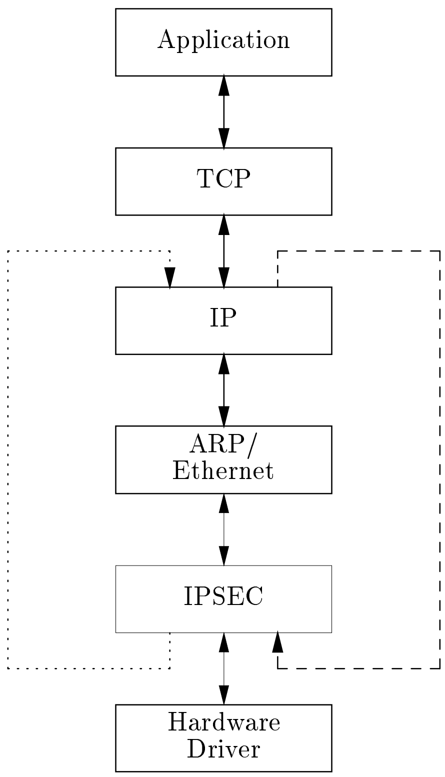

Figure 3: An alternative (but difficult) implementation that lives above and below IP. The dashed line denotes the processing path for fragmented packets; the dotted line shows re-entry to IP for re-routing.

there is one very important case where small MTUs are used: dial-up machines.

For TCP, an ugly implementation could detect the `SYN` packets and fiddle with the maximum segment size option. Apart from esthetic questions, it leave unsolved the problem of UDP.

We could try to specify the proper MTU during the key management process. While probably a good idea, it doesn't solve the problem of informing the higher layers of the protocol stack.

The only solution, then, is to reimplement fragment reassembly in the IPSEC module. While not tremendously clean, it works, and we were able to steal much of the code from 4.4BSD. The `mbuf` code had to be stripped out, though, since our module uses flat buffers.

Fragmentation poses another difficulty. Suppose you can induce a host to send a packet large enough that the host itself fragments it before transmission. Put a fake transport header at the fragmentation boundary. The security layer will calculate the appropriate authentication data for this fragment. Note that it cannot include the fragmentation data in the calculations, since this may change en route. This fragment can now be retransmitted, with a new IP header that doesn't indicate fragmentation. The authentication check will succeed and the fake transport header will be believed, thus permitting injection of phony data into a connection. (Other security issues relating to fragmentation are discussed in [13].)

The only remaining problem for the attacker is how to induce the host to send such a packet. MS-DOS computers are not multi-user machines, which eliminates one threat, but services such as `ping` will do the trick.

Our solution to this security threat is to use encapsulation mode on all fragments passed to IPSEC. That is, we transmit the original IP header inside the authentication header, and create a new IP header that has no mention of fragmentation. Upon receipt, the authentication header will be stripped off, and the enclosed IP fragment passed up the stack for reassembly by IP itself.

5 Key Management

At development time, no key management protocol had been defined for IPSEC, though there were a number of candidates. Accordingly, we did not try to implement anything more sophisticated than manual keying. Our intent, though, is to make key management a completely manual process, even when a protocol is available. That is, before trying to communicate via IPSEC, the user must explicit invoke a command to negotiate and load session keys. Though a bit awkward for a general-purpose IPSEC module, it works well for our "call home" model. Additionally, keying information can be read at boot time from a configuration file.

Trying to do automated key management would require that the IPSEC module send and receive TCP and/or UDP messages. Apart from the problem of recursion, TSR routines are not allowed to issue normal MS-DOS system calls. Another way to handle this

Table 1: IPSEC performance measurements doing FTP.

no IPSEC (vanilla PC/TCP)	188 KB/sec
IPSEC module, no authentication	135 KB/sec
IPSEC with MD5 MAC	65 KB/sec

would be to pass the keying request to a user-level program via an up-call; unfortunately, such things are difficult to implement, since the MS-DOS system does not support multitasking.

A new key table is downloaded via a special call to IPSEC packet driver. Rather than passing the new table to the driver via this call, we have the driver pass back its address, and let the key management program overwrite it manually: storage protection isn't an issue for MS-DOS computers. An additional benefit of this scheme is that the same call can be used by a table dump utility.

6 Performance

Not surprisingly, there is a performance cost associated with IPSEC (Table 1). On FTP tests run between two 90 megahertz Pentium systems[5] with 3C509 Ethernet cards, the mere presence of the IPSEC module caused a performance hit of about 30%. This can be attributed to data copying and the extra context switch. The latter will be especially serious in the future, since packet drivers run in 16-bit mode, and newer stacks will run in 32-bit mode.

Adding MD5 authentication caused a further 50% drop in throughput. However, these tests were run with the default small window size; this undoubtedly hurt performance considerably. Using a larger window would have run afoul of the fragmentation issue.

The import of these numbers for long-haul connections is unclear. Even the slowest speed is about a third the capacity of a DS1 line, and few real connections approach even that figure.

7 Conclusions

The usual approach to implementing IP security is to modify the source code. By exploiting open interfaces, we were able to implement the necessary functionality without having source code to the protocol stack. This is an old technique in some parts of our profession, but much Internet work has been done by UNIX system programmers who have been spoiled by easy access to source code.

Our implementation isn't perfect. There is some duplication of functions and some awkward techniques. These were needed where the interface definition wasn't quite sufficient. While it's impossible for system designers to anticipate all contingencies, it's a good idea to remember two rules: first, eschew magic numbers, such as a 1500 byte MTU; second, where two or three objects of some type can exist, make it easy for outsiders to add a fourth or a fifth.

It isn't yet clear to us how a computer using authenticated IP should interact with our firewall. Should the user be constrained to our current mode of operation: `telnet` to a gateway, followed by token-based authentication? Should the user be allowed to tunnel through to an end system, using host-to-router mode authentication? Can we use the current challenge/response devices for key distribution? What if the user's machine has been penetrated? Are we leaving ourselves open to attack? More and more MS-DOS systems run servers of various types, leaving them more susceptible to remote penetration. (To those who claim that good cryptography obviates the need for firewalls, we note that a private, well-authenticated connection to a hacker is of little benefit if the software on our end of the call is buggy enough to grant shell access to our systems.)

Another way to view our code is as a host-based emulation of a true "bump in the cord" security unit. The thorny issues we faced, such as MTU, fragment reassembly, and spoofing, would apply equally well to any front-end box. This suggests the need for architectural improvements to IP hosts to make them "security-ready".

8 Acknowledgments

This work would have been much more difficult were it not for the copious amount of freely usable source code available. Some of it we studied; other sections were included verbatim. We used sections of code from 4.4BSD, the Crynwyr/Clarkson packet driver collection, NCSA `telnet`, and MD5. Additionally, much was learned by studying Robert Glenn's IPSEC code for BSD/OS.

References

[1] R. Atkinson. IP authentication header. Request for Comments (Proposed Standard) RFC 1826, Internet Engineering Task Force, August 1995.

[2] R. Atkinson. IP encapsulating security payload (ESP). Request for Comments (Proposed Standard) RFC 1827, Internet Engineering Task Force, August 1995.

[3] R. Atkinson. Security architecture for the internet protocol. Request for Comments (Proposed Standard) RFC 1825, Internet Engineering Task Force, August 1995.

[4] FTP. PC/TCP packet driver specification, June 1994. Available from ftp.ftp.com.

[5] V. Jacobson. Compressing TCP/IP headers for low-speed serial links. Request for Comments (Proposed Standard) RFC 1144, Internet Engineering Task Force, February 1990.

[6] C. Kent and J. Mogul. Fragmentation considered harmful. In *Proceedings of SIGCOMM '87*, pages 390–401, August 1987.

[7] Ralph C. Merkle and Martin Hellman. On the security of multiple encryption. *Communications of the ACM*, 24(7):465–467, July 1981.

[5]Pentium is a registered trademark of Intel

[8] P. Metzger, P. Karn, and W. Simpson. The ESP DES-CBC transform. Request for Comments (Proposed Standard) RFC 1829, Internet Engineering Task Force, August 1995.

[9] P. Metzger and W. Simpson. IP authentication using keyed MD5. Request for Comments (Proposed Standard) RFC 1828, Internet Engineering Task Force, August 1995.

[10] J. Mogul and S. Deering. Path MTU discovery. Request for Comments (Draft Standard) RFC 1191, Internet Engineering Task Force, November 1990.

[11] NBS. Data encryption standard, January 1977. Federal Information Processing Standards Publication 46.

[12] R. Rivest. The MD5 message-digest algorithm. Request for Comments (Informational) RFC 1321, Internet Engineering Task Force, April 1992.

[13] P. Ziemba, D. Reed, and P. Traina. Security considerations for IP fragment filtering. Request for Comments RFC 1858, Internet Engineering Task Force, October 1995.