Implementing IPsec *

Angelos D. Keromytis
University of Pennsylvania – angelos@dsl.cis.upenn.edu
John Ioannidis†
AT&T Labs – Research – ji@research.att.com
Jonathan M. Smith
University of Pennsylvania – jms@central.cis.upenn.edu

August 1, 1997

Abstract

The IP Security protocols are sufficiently mature to benefit from multiple independent implementations and worldwide deployment. Towards that goal, we implemented the protocols for the BSD/OS, Linux, OpenBSD and NetBSD[1]. While some differences in the implementations exist due to the differences in underlying operating system structures, the design philosophy is common. A radix tree, namely the one used by the BSD code for routing purposes, is used to implement the policy engine; a transform table switch is used to make addition of security transformations an easy process; a lightweight kernel-user communication mechanism is used to pass key material and other configuration information from user space to kernel space, and to report asynchronous events such as requests for new keys from kernel space to a user-level keying daemon; and two distinct ways of intercepting outgoing packets and applying the IPsec transformations to them are employed. In this paper, the techniques used in our implementations are explained, differences in approaches are analysed, and hints are given to potential future implementors of new transforms.

1 IP security

Traditional approaches to system security have focused on high level, application dependent solutions. Regardless, network layer security has been accepted as a necessary element in a multi layer security architecture. Early attempts at this have been the SP3 [SP3] and NLSP [NLSP] protocols, and others [Anderson]. In November 1992, various members of the IETF (Internet Engineering Task Force) decided to design and deploy a protocol suitable for the large-scale Internet environment, which uses IP as the network protocol [IP]. The first such experimental protocol was swIPe [SWIPE], and although it never quite caught on, it proved that the concept was sound and the goal achievable. A few years later, the IETF IPSEC Working Group developed a set of specifications [Arch] and in December 1995 in Dallas held the first interoperating session between several vendors and individuals who had implementations of the proposed standard. The proposed standard was also adopted by the IPv6 [IPv6] Working Group as mandatory for the next generation Internet network protocol. A number of RFCs (Request For Comments) and Internet Drafts describe the IPsec architecture and specifics[AH] [ESP] [AHMD5] [AHSHA] [ESPDES] [ESP3DES] [IPIP]. Since the original set of documents, the Working Group has published new RFCs and Internet Drafts which define new transforms, modes of operation and APIs (Application Programmer Interfaces). [HMACMD5] [HMACSHA] [CAST128] [DESX] [SAPI] [PFKEY] Reader familiarity with some of those documents is assumed.

In this paper we describe our implementation of the IPsec architecture. John Ioannidis developed the first implementation in November 1995. The code was later ported to NetBSD and OpenBSD, and the new transforms were added by Angelos D. Keromytis. John Ioannidis wrote the Linux implementation as part of the SWAN project. Niels Provos continues integration of IPsec, key management[2]with the OpenBSD operating system.

2 Our implementation

Our implementation includes kernel patches for processing incoming and outgoing IP datagrams, maintaining the transforms database and the policy table, setting a virtual

 This code was originally developed in Greece. Further work on this topic is being made possible by DARPA under Contracts #DABT63-95-C-0073, #N66001-96-C-852, with additional support from Hewlett-Packard and Intel Corporations.

†This work was done before this author joined AT&T.

[1]The BSD/OS and Linux versions were developed in Greece by John Ioannidis between Fall of 1995 and Winter of 1997; the NetBSD port was done by Angelos Keromytis in December of 1996, also in Greece. Development on the NetBSD port continues at the University of Pennsylvania. The OpenBSD port was originally done by Angelos Keromytis and Niels Provos.

[2]The Photuris daemon.[PHOTURIS]

link's[3] security properties through **setsockopt()** calls, as well as a set of tools for configuring the policy table.

2.1 Transforms and policy databases

The transforms database, which holds information such as session keys, replay counters, SPIs and other transform specific information[4], is implemented as a hash table inside the kernel. The elements are referenced by SPI[5], remote IP address and security protocol (since that is a unique identifier of a security association). Entries that belong to the same security association are modified to point to each other, forming a doubly linked list. This allows fast lookup and packet processing, while minimizing routing table lookups.

The reason routing table lookups are performed is because we implemented our policy engine using the Radix tree code available in the BSD kernel[6], which is also used by the routing infrastructure. This code decides whether some IPsec transforms should be applied to an outgoing packet. Using the Radix tree code has some advantages:

- reuse of existing code, which in turn means:
  - smaller kernel size
  - faster development
  - less debugging need be done
- the Radix tree code has been optimized over the years
- it allows for policy decision with multiple fall backs, due to the way it works [Stevens].

This last item is the most important. It means that we can have a general policy for communicating with a specific host, but then can override it for specific packet flows. Currently, decisions can be made based on source/destination IP addresses, protocol number and source and destination ports (if the transport protocol in use is TCP [TCP] or UDP [UDP]). An example of a policy that can be expressed easily in this implementation is:

- all packets from *IP1* to *IP2* should be processed using the SPI chain starting with *SPI1*
- all packets from *IP1* to *IP2* with transport protocol *ICMP* [ICMP] should be processed using the SPI chain starting with *SPI2*
- all packets from *IP1* to *IP2* with transport protocol *TCP* should be processed using the SPI chain starting with *SPI3*
- all packets from *IP1* to *IP2* with transport protocol *TCP* and source port *P1* and destination port *P2* should be processed using the SPI chain starting with *SPI4*

New test fields can be added in this scheme, without modifying the lookup code.

2.2 Virtual interfaces

Both the BSD and the Linux implementations implement a virtual interface (**enc0** for BSD, **ipsec**n, where n is 0, 1, ... for Linux), which has several uses:

- assist packet filtering: when used in a packet filtering firewall, the interface can be used as a means of selecting packets that have gone through IPsec processing. This allows a firewall to only permit authenticated/encrypted traffic, without having to change the internals of the packet filter itself.
- (for Linux only): provide a means to send packets to the IPsec code; in the next subsection we shall explain why we had to use a virtual interface for this, and what our approach was in the BSD code.
- (for Linux only): allow us to implement the transport-layer mode of IPsec. The lack of an **ip_output()** routine in Linux made this a non-trivial exercise.

In BSD, the **enc0** interface is implemented like the loopback interface, except that no input routine is defined, as it is not needed. See the next sections for more details on the Linux implementation.

2.3 Incoming packets

Incoming IP packets that have a *Next Protocol* field indicating ESP, AH, or IP-in-IP are delivered by **ip_intr()** to **ah_input()**, **esp_input()** or **ipe4_input**[7] respectively. These routines lookup the outer SPI of the IP packet (the one corresponding to the last IPsec transform applied) in the transforms database. If an entry is found for an SA (Destination Address, SPI, Security Protocol), the packet and the relevant entry is passed to the appropriate routine which performs the necessary (cryptographic) transforms. If more than one SPI applies (recursively) to the packet, the appropriate routines are called in turn. When the transforms are applied to the received packet, flags indicating what security services were used are set in the mbuf[8] chain header. This lets the transport layer's input routines compare the expected level of security to the actual security services used for the packet and drop it if the security level was insufficient.

[3] A connection or, more generally, a packet flow.

[4] It is imperative that this information is not accessible by a normal user.

[5] An SPI (Security Parameters Index) is a 32-bit value which can be interpreted, along with the peer's IP address and the security protocol in use (ESP or AH), as an identifier of the specific parameters of a security association.

[6] For Linux, we had to port the Radix tree code. We named it Radij, but the functionality is the same.

[7] A slightly modified **ipip_input()** is used if the kernel is compiled with multicasting support.

[8] **mbuf**s are a data structure used to keep networking data, such as packets, in the BSD kernel. A large packet is kept in a linked list of such structures, called an mbuf chain.

If these routines fail, the packet is discarded and the failure is logged. Additional measures might be taken in future releases. Various statistics are kept, and can be seen via the **netstat** command or the kernfs/procfs interface on some platforms. A **sysctl** interface has also been provided, to turn debugging messages on or off if the kernel has been compiled with the appropriate options.

2.4 Outgoing packet processing

The BSD and Linux implementations differ in the way they handle outgoing packets, due to the differences of their network stacks.

2.4.1 Outgoing packet processing in BSD

In this implementation, the **ip_output()** routine is "tapped"; shortly after it is called, a lookup in the policy database is performed. The result of that query is either *no action*, in which case the packet is sent out as is, or the beginning of an SPI chain which defines the set of transforms that should be applied to the packet before it is sent on the network. In the latter case, the appropriate routines are called successively. If no failure is reported, statistical counters and the internal state are updated. Finally, the modified packet is sent to the appropriate virtual interface's output routine, along with a flag indicating that IPsec processed the packet. It will then be re-processed by **ip_output()**, just for fragmentation and next-hop route discovery (since the destination address might have changed as a result of tunneling).

The mbuf chain flags can also be set by the transport[9] layer's output routine so that the packet is not sent to the network unless certain security services are used (authentication and/or confidentiality).

2.4.2 Outgoing packet processing in Linux

Since Linux doesn't have an **ip_output()** routine or its equivalent, we decided to change the way the policy engine works: for packets that need to go through the IPsec code, routing entries are created that point to one of the virtual interfaces. These virtual interfaces are matched on a one-on-one basis with real network interfaces. This allowed us to push security processing to these pseudo-device drivers, without radically modifying the routing code.

However, since Linux does not use the Radix tree for its routing table, we were not able to route based on arbitrary fields in the packet, as we did in BSD. This means that if there exists a security association with a remote host for a specific packet flow (for example, a TCP connection), all packets to that host will be delivered to an **ipsec** interface, which then has to decide which of these packets should be further (cryptographically) processed. This processing is done by performing a lookup in a Radij tree that contains the more detailed policy information. Otherwise, the output routine of the **ipsec** interface behaves like the BSD **ip_output()**[10].

Using virtual interfaces in this way has the advantage of presenting a more realistic MTU to the TCP protocol. It is not clear however what the MTU that should be reported in the interface structure is. Small MTUs will cause no fragmentation due to IPsec processing, but will decrease throughput by reducing the actual data size in the packets. Large MTUs (close to the real network MTU) maximize the effective bandwidth, but may cause fragmentation. However, since the greatest performance cost in software IPsec implementations is the cryptographic algorithms, it is unclear whether fragmentation will further degrade network performance. Determining the MTU dynamically (for each packet sent) would require extensive modifications of the TCP code and impose additional delays in outgoing packet processing for all such packets.

2.5 Kernel interfaces

For communication between user processes and the kernel, we implemented two mechanisms:

- the PF_ENCAP[PFENCAP] protocol family, which works similarly to the PF_ROUTE. It is used for manipulating the transforms database (setting up new security associations, modifying and deleting them). It is used by the manual keying utilities, as well as potential key management daemons. In Linux, we used Netlink instead, which is a generic kernel communication mechanism. This mechanism allows the kernel to ask the key management daemons to establish a security association with a remote host.

- a **setsockopt()/getsockopt()** interface. This allows processes to set the security properties of their packet flow(s) and get information about system defaults. The options are set in the protocol control block of the socket. Subsequent calls to **connect()** or **send()** will cause either use of existing SPIs or a notification to the key management daemon (through PF_ENCAP or PF_KEY) to negotiate a security association with the remote host.

2.6 Management utilities

A number of utilities that take advantage of the PF_ENCAP interface were written to allow for manual key setup, which is a requirement for all IPsec conforming implementations. These utilities initialize, modify and delete the state kept in the kernel transform and policy databases, the latter using the PF_ROUTE socket mechanism.

[9] This includes UDP and TCP, but also ICMP and raw sockets.

[10] Although some details of the BSD implementation, such as the service flags, have not yet been included.

2.7 Future work

We plan on adding soft state to the tunnel endpoints, so that Path MTU [PMTU] discovery information can find its way back to the sender, in the presence of multiple encapsulation in the network. An additional possible optimization for Path MTU discovery would be to check what the final size of a packet about to be processed is; if it is larger than the MTU of the network interface and the DF (Don't Fragment) flag is set, there's no need to actually do the cryptographic processing. Instead, the appropriate ICMP message can be sent back. We also plan to modify TCP's initial MSS resolution to improve performance in the presence of fragmentation caused by IPsec imposed headers. Finally, there is thought of implementing the PF_KEY kernel communication draft, as an alternative to PF_ENCAP.

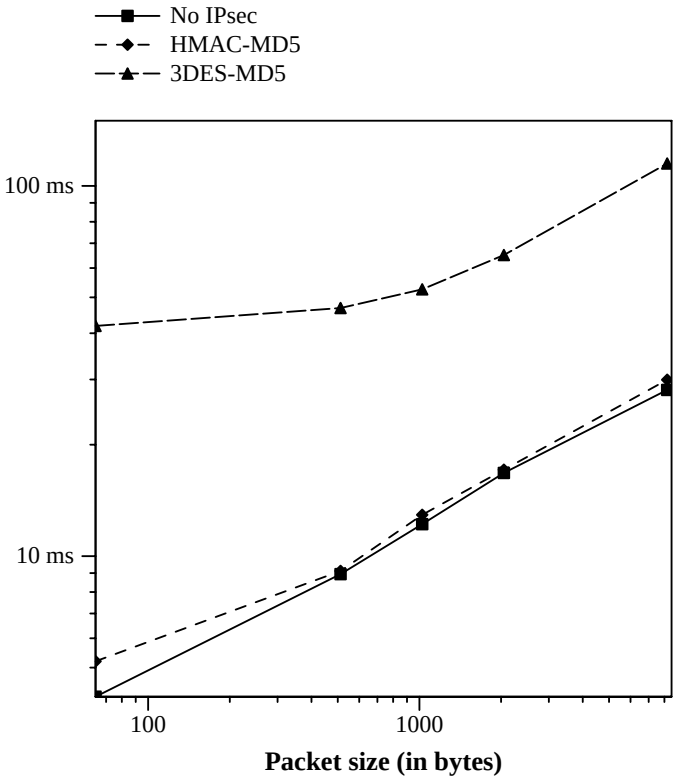

Figure 1. ping performance with IPsec

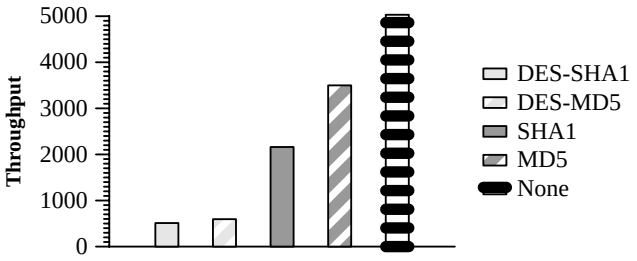

Figure 2. blasting over UDP (cpu time)

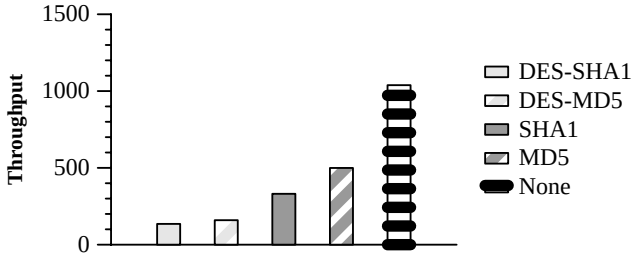

Figure 3. TCP transfer (real time)

3 Performance measurements

Performance parameters of interest include latency and throughput. We measured latency using `ping`. The measurement configuration consisted of two machines running our software. The first was a 166MHz Pentium equipped with a 100Mbit/sec ethernet card, the second a 120MHz Pentium with a 10Mbit/sec ethernet card. We did the test for different packet sizes (512, 1024, 4096 and 8192 bytes of payload) and different IPsec transforms, pinging from the P166 to the P120. The results can be seen in Figure 1. Notice that the scale is logarithmic. The graph shows that the cost of authenticating packets does not really downgrade response time, but that encryption (especially triple-DES) is a major bottleneck.

The results from the Linux implementation are similar, which is not surprising since the computationally intensive part (encryption) remains the same.

In the second test, we measured how fast the P166 machine could "push" 15MB of data to the network, using UDP, while applying different transforms on the packets. The size of the packets was 1KB. The results are shown in Figure 2.

In a third test, we transfered 15MB of randomly chosen data from the P166 to a 50MHz SPARC LX with a 10Mbit/sec ethernet card. We used `ttcp` to measure throughput, with TCP as the transport protocol. The results are in Figure 3.

It would be interesting to do the same tests using some hardware cryptographic device (a DES chip), but the current regulations (ie. ITAR) make this difficult.

4 Conclusion

We described the IPsec architecture and some of the important parts of our implementation(s) for the BSD and Linux kernels. Since most of the existing implementations are proprietary, we believe this paper will help potential developers in designing and implementing the standards in the future.

Our BSD IPsec implementation is freely available to all US citizens. A somewhat older version of it is available to everyone, and can be found at ftp://ftp.funet.fi/pub/unix/security/net/ip/, file

BSDipsec-pl1.tar.gz. The code found in that archive was written while the first two authors were in Greece. The Linux implementation is available at the same directory, file ipsec-0.5.tar.gz. Most recently, work is being done in the OpenBSD project, which has incorporated our BSD implementation in their base distribution[11]. There is still work to be done on our implementation, as new transforms and interfaces are defined and standardized.

5 Acknowledgments

Most of the code was written in Athens, Greece, and in Distributed Systems Lab, University of Pennsylvania. Thanks to Niels Provos for pointing out problems in our code and working hard to both fix them and integrate the code in the OpenBSD kernel. Bill Arbaugh provided many useful comments on earlier versions of this paper.

References

[SWIPE] "The Architecture and Implementation of Network-Layer Security Under Unix", *Ioannidis, J. and Blaze, M., Fourth Usenix Security Symposium Proceedings, October 1993*

[PFENCAP] "The ENCAP Key Management Protocol, Version 1", *Ioannidis, J., Keromytis, A. D. and Provos, N., Work in Progress*

[AH] "IP Authentication Header", *Atkinson, R., RFC 1826, August 1995*

[AHSHA] "IP Authentication using Keyed SHA", *Metzger, P. and Simpson, W., RFC 1852, October 1995*

[ESP3DES] "The ESP Triple DES-CBC Transform", *Metzger, P., Karn, P. and Simpson, W., RFC 1851, October 1995*

[ESP] "IP Encapsulating Security Payload", *Atkinson, R., RFC 1827, August 1995*

[AHMD5] "IP Authentication using Keyed MD5", *Metzger, P. and Simpson, W., RFC 1828, August 1995*

[ESPDES] "The ESP DES-CBC Transform", *Metzger, P., Karn, P. and Simpson, W., RFC 1829, August 1995*

[Arch] "Security Architecture for the Internet Protocol", *Atkinson, R., RFC 1825, August 1995*

[UDP] "User Datagram Protocol", *Postel, J.B., RFC 768, August 1980*

[TCP] "Transmission Control Protocol", *Postel, J.B., RFC 793, September 1981*

[IP] "Internet Protocol", *Postel, J.B., RFC 791, September 1981*

[IPv6] "Internet Protocol, Version 6 (IPv6) Specification", *Deering, S. and Hinden, R., RFC 1883, January 1996*

[HMACMD5] "HMAC-MD5 IP Authentication with Replay Prevention", *Oehler, M. and Glenn, R., RFC 2085, February 1997*

[HMACSHA] "HMAC-SHA IP Authentication with Replay Prevention", *Madson, C. and Glenn, R., draft-ietf-ipsec-auth-hmac-sha196-00.txt, Work in Progress*

[DESX] "The ESP DES-XEX3-CBC Transform", *Simpson, W. A. and Baldwin, R., draft-ietf-ipsec-ciph-desx-00.txt, Work in Progress*

[CAST128] "The ESP CAST128-CBC Algorithm", *Pereira, R. and Carter, G., draft-ietf-ipsec-ciph-cast128-cbc-00.txt, Work in Progress*

[IPIP] "IP in IP Tunneling", *Simpson, W., RFC 1853, October 1995*

[PMTU] "Path MTU Discovery", *Mogul, J. and Deering, S., RFC 1191, November 1990*

[ICMP] "Internet Control Message Protocol", *Postel, J., RFC 792, September 1981*

[MD5] "The MD5 Message-Digest Algorithm", *Rivest, R., RFC 1321, April 1992*

[DES] "NBS FIPS PUB 46 - Data Encryption Standard", *National Bureau of Standards, U.S. Department of Commerce, January 1977*

[Stevens] "TCP/IP Illustrated Volume 2 - The Implementation", *Wright, Gary R. and Stevens, W. Richard, Addison-Wesley, 1995*

[SAPI] "A Simple IP Security API Extension to BSD Sockets", *McDonald, D. L., draft-mcdonald-simple-ipsec-api-01.txt, Work in Progress*

[Anderson] "A Protocol for Secure Communication in Large Distributed Systems", *Anderson, D. P. et al, Technical Report UCB/UCSD 87/342, University of California, Berkeley, February 1987*

[SP3] "NISTIR 90-4250: Secure Data Network Systems (SDNS) Network, Transport and Message Security Protocols", *National Institute of Standards and Technology, February 1990*

[NLSP] "ISO-IEC DIS 11577 - Information Technology - Telecommunications and Information Exchange Between Systems - Network Layer Security Protocol", *ISO/IEC JTC1/SC6, November 1992*

[PFKEY] "PF_KEY Key Management API, Version 2", *McDonald, D. L., Metz, C. W. and Phan B. G., draft-mcdonald-pf-key-v2-03.txt, Work in Progress*

[PHOTURIS] "Photuris: Session Key Management Protocol", *Karn, P. and Simpson, W. A., draft-simpson-photuris-14.txt, Work in Progress*

[11] For more information, see http://www.openbsd.org/