21

IP Security (IPsec)

Of the new functionalities that IPv6 introduced to the TCP/IP stack, IPsec is best known, "backported" to IPv4 without loss of functionality and most widely implemented.

21.1 Basic Concepts

The ideas behind IPsec are quite straightforward: Additional option headers provide for the authentication and encryption of IP packets. In practice, IPsec itself is dauntingly complex. Implementations follow suit, ranging from "mostly undocumented" to "configuration nightmare" to "missing essential parts".

So we just take a look at the concepts and its operational implications but don't even attempt to set up a working installation.

21.1.1 Authentication and Encryption

IPsec supports two independent features: The authentication of a packet's source and the encryption of its contents. Both features can be used independently.

An authenticated packet contains an *authentication header* (*AH*) which certifies that the packet originated from the source address shown in the base header. Authentication uses a shared secret between the communication peers. The sender computes a checksum, or hash, over the shared secret, the relevant parts of the IP headers and the IP payload and stores it in the authentication header. The receiver recomputes the checksum and compares its result with the checksum found in the authentication header. If they match, then the receiver knows that the sender also holds a copy of the shared secret.

Encryption also uses an option header, called *encapsulating security payload* (*ESP*). It also uses a pre-installed shared secret between the peers. The

sender inserts an ESP header into an IP packet and encrypts all data following that header. The receiver finds the ESP header and then decrypts all the following data before it processes the packet.

21.1.2 Transport and Tunnel Mode

So far we have assumed that we want end-to-end encryption and authentication. But in section 21.2.1 we'll see that end-to-end encryption is sometimes a problem rather than a solution. To work around this and to use IPsec for *virtual private networks* (*VPN*s), IPsec also works together with IP-in-IP encapsulation.

End-to-end encryption is called *transport mode* while the combination of IPsec and encapsulation is called *tunnel mode*. Figure 21.1 shows the differ-

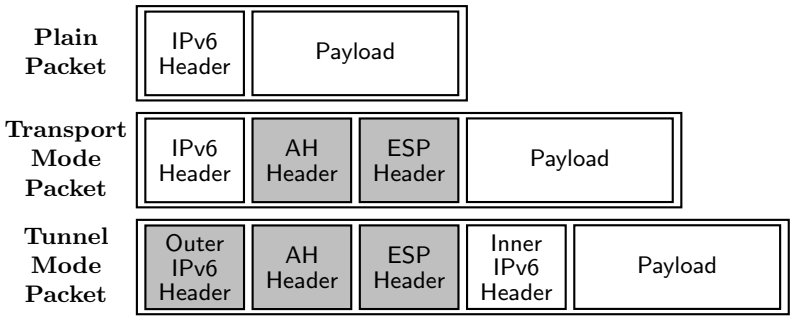

Fig. 21.1. Plain IPv6 packet and IPsec packets

ences between the packets. Things get slightly more complex if more option headers get involved; for our purposes it suffices to know that those option headers that are read by routers are placed before the IPsec headers and those that are only used by the final recipient are placed after. Everything that follows the ESP header is encrypted. Additionally, either the AH or ESP header may be missing.

21.1.3 Policy and Key Management Within the Kernel

When setting up IPsec, one of the first things we need to do is to decide which nodes will use which IPsec features between each other. The kernel maintains a *security policy database* (*SPD*) to decide which IPsec features to apply to outbound packets and which to require from inbound ones.

The SPD is quite similar to a packet filter configuration except that it doesn't filter packets that are forwarded. It is usually quite static and manually configured.

The kernel also stores all the shared secrets in a *security association database* (*SAD* or sometimes *SADB*). The SAD contains a set of *security associations* (*SAs*). Every SA consists of a source and destination IP address that it relates to, a key, an encryption or authentication algorithm, and a *security parameter index* (*SPI*). The SPI is a 32 bit integer that ESP and AH headers use to refer to an SA.

Consider an example: Node A wants to send a packet to node B. It first checks its SPD to see if it should apply IPsec, and if so, with what features and algorithms. Assume that the SPD on A says to use ESP with AES encryption and AH with an SHA1 hash. Next, A takes a look at the SAD to find the necessary SAs—one for ESP and another for AH. It takes the key and algorithm from the ESP SA, applies the algorithm to the data using the key and adds an ESP header behind the base header. The ESP header contains the SPI so the receiver can figure out which SA to use to decrypt the packet. Next, node A does almost the same for the AH header: It takes the key from the authentication SA, uses it to create a hash value from the key, the payload and those parts of the IP header that don't change in transit, and stores the hash value and the SPI in an AH header it puts behind the base header. Then it sends the packet to B.

When B receives the packet, it first takes a look at its own SPD to decide which IPsec features it requires from the packet. If any are missing or use an insufficiently secure algorithm, then B will silently discard the packet. Otherwise it will search its SAD for the matching SAs; if multiple SAs for the given source and destination address exist, then B will use the SPIs from the headers to find the correct ones in its SAD. It can then check the authenticity of the packet by verifying the hash value from the AH header and decrypt the payload. Then it processes the packet as usual.

While this example ignores a number of fine points, it explains how IPsec works within the kernel.

21.1.4 The Internet Key Exchange Protocol (IKE)

At this point, a major issue remains: How do we distribute those shared keys between nodes?

We can configure the SAs manually. This approach is simple and avoids a number of notorious interoperability problems. It is also tedious and error-prone. And what's worse, in a way it is less secure than dynamically exchanged keys: The longer a key is used, the more ciphertext an attacker can gather for cryptographic analysis for that key. Once the attacker has broken the key, he can decrypt a huge amount of traffic. What's worse, even if the attacker couldn't break the key, he could still use captured traffic for a replay attack; with captured NTP packets he could for example turn back a node's system clock to prepare for an attack on Kerberos, which relies heavily on synchronized clocks.

With dynamically distributed keys IPsec can use sequence numbers, much like TCP, to prevent these replay attacks. For this purpose the kernel asks a userland program to update the SAD if necessary. This userland program then uses a public-key cryptography protocol to set up the needed SAs and sequence numbers on both sides.

The protocol is called *Internet key exchange* (*IKE*). It uses parts of the *Internet security association and key management protocol* (*ISAKMP*), which is why some implementations refer to it as ISAKMP.

IKE operates in two phases. The first phase establishes a secure channel between the IKE daemons, possibly using public-key cryptography to authenticate each other. This secure channel is then used during the second phase to exchange shared secrets for the SAs.

In the most simple setup, IKE uses pre-shared keys between nodes to authenticate and encrypt the exchange of the IPsec keys; this is safer than using manually configured IPsec keys but still requires a separate key for every pair of nodes that communicate and a secure means to distribute the keys.

In a more advanced setup IKE uses public key cryptography. With unauthenticated, or self-signed, X.509 certificates we only need to distribute an X.509 certificate for every node to all other nodes; the distribution doesn't need to be secret as with shared keys anymore.

Finally, IKE can use X.509 certificates that are signed by a *certificate authority* (*CA*). With these, we only need to distribute the public key of the CA to all machines, like we do with HTTPS web servers. In fact, since both HTTPS and its underlying SSL/TLS protocol use X.509 certificates, we can use the same CA for both SSL/TLS and IKE. If we want to communicate securely with peers outside our administrative control, then we can use the services of well-known CAs; their certificates are distributed with virtually all HTTPS-capable web browsers.

Unfortunately, this scenario isn't widely supported by implementations. Taking a look at the documentation available today, IPsec is most commonly used either between individual hosts or in tunnel mode between VPN end points. Some implementations don't actually bother to compare the source address of an IKE packet with the address stored in the certificate.

The original IKE specification as of RFC 2409 [57] showed two problems: The key exchange was unnecessarily complicated and a cryptographic weakness in the IKE exchange was discovered. In December 2005, RFC 4306 [31] specified a successor protocol *Internet Key Exchange Version 2* (*IKEv2*) to remedy these problems. Unfortunately, IKEv2 implementations aren't generally available yet.

21.1.5 References

IPsec is standardized in an entire cluster of RFCs; these are the most important ones:

RFC 4301 [81] defines the fundamental IPsec architecture, RFC 4302 [78] the authentication header and RFC 4303 [79] the encapsulating security payload. RFC 3280 [68] specifies a X.509 public key infrastructure for the Internet in general, RFC 2409 [57] IKEv1 and RFC 4306 [31] IKEv2.

A number of additional RFCs specify the cryptographic algorithms that IPsec supports; they are referenced in the main RFCs.

21.2 Open Problems

Before we consider setting up IPsec in an environment we should take a closer look at its security implications. Some of these are direct effects of the fact that IPsec operates within the network layer; others are consequences of the IPsec standards and their implementations; yet others are related to authentication and encryption in general.

In short, IPsec solves a limited range of security concerns at a very high price. It is a long way from the catch-all security solution that some people claim it to be.

21.2.1 Inherent Limitations

The specifications for IPsec are excessively long. The most relevant and current RFCs exceed 450 pages (see section 21.1.5), not including the now outdated IKEv1. The consequences are notorious interoperability problems and a high chance of faulty implementations that might be exploitable.

IPsec operates within the network layer, so it only provides authentication and encryption between nodes. It is *not* a substitute for SSL/TLS, or SSH, or other application-layer mechanisms that provide cryptographic protection for individual users or processes. Neither is it a full substitute for missing encryption and authentication at the application level—telnet over IPsec is still inferior to SSH because it doesn't authenticate a user by a public key. But then, SSL/TLS and such don't provide proper protection between nodes: Running neighbor discovery through SSL/TLS won't work, because first SSL/TLS needs neighbor discovery to have finished before it can set up its TCP connection.

SSL/TLS provides authentication based on DNS names while IPsec authenticates IP addresses only. Since few people bother to remember the IP address of `online-banking.example.com` and even fewer will bother to track the occasional renumberings, authentication should really be based on DNS names. IPsec can't offer this: while it *may* be possible to do a DNS lookup on a DNS name stored in an X.509 certificate, doing so relies on the security of the DNS system; even though attempts have been made to retrofit DNS with up-to-date security features, these can't compete with the way that SSL/TLS authenticates the DNS name directly.

Originally, IPsec only supported unicast packets. Multicast groups were considered "publicly accessible" and therefore couldn't be encrypted. Authentication support was also missing; in theory it is possible to sign every packet using public key cryptography, but with today's public-key algorithms doing so is prohibitively CPU-expensive. Since IPv6 uses multicasts for a number of internal purposes, from neighbor solicitations and router discovery to various dynamic routing protocols, all of these are still unprotected even when IPsec is in use. RFC 3740 [56] and the latest IPsec RFC suite add multicast support to the IPsec framework but still don't specify any cryptographic algorithms that could actually be used for multicast authentication or encryption.

According to RFC 2406 [80], IPsec implementations were only required to provide the "NULL" encryption, which doesn't actually encrypt the payload at all, and *DES* (*data encryption standard*) in *CBC* (*cipher block chaining*) mode, which has been broken using brute force as far back as 1998. Cryptographically strong algorithms were optional. Microsoft's Windows XP doesn't even implement DES, so its IPsec implementation not only violates RFC 2406 but provides no encryption at all. RFC 4305 [29] recently changed these requirements but hasn't been widely implemented yet.

Surprisingly enough, encryption can actually reduce security. If we allow end-to-end encryption between machines inside our network and others on the outside, people can send data in and out without our control: Business secrets, malware or whatever. With IPsec, a packet filter can't even determine the transport layer protocol and port number used; it just notices some kind of traffic from workstation X to `dns0.example.net` but can't tell if it is genuine DNS traffic or anything else.

Finally, encryption and authentication tend to provide people with an exaggerated feeling of security. The huge increase of phishing attacks using HTTPS with invalid certificates demonstrates that a large fraction of people believe online banking is safe "because it is encrypted and everything".

21.2.2 Implementation Issues

Beyond the fundamental concerns we've addressed in the previous section, real-world implementations suffer from a number of additional problems.

Most implementations provide some strong encryption algorithms, but in a heterogeneous environment it may happen that no universally supported algorithm exists, even without Microsoft Windows XP boxes that only support "NULL" encryption.

Debian Sarge exhibits some erratic behaviour concerning IPv6 entries in the security policy database: Requiring encryption for UDP packets blocks neighbor discovery.

Setting up a certificate authority requires OpenSSL 0.9.8 or later; older versions don't support IPv6 addresses in the `altSubjectName` field.

IKEv2 support is mostly missing; so far only FreeBSD 6.1 offers an implementation `racoon2`, which comes without proper documentation. Using

signed X.509 certificates is apparently possible, but not documented. Debian Sarge and Solaris 10 don't support IKEv2 at all.

Virtually all implementations focus on setting up a VPN using tunnel mode. Using IPsec throughout a local network based on signed X.509 certificates is tedious at best.

Solaris 10 ignores the IP address stored in an X.509 certificate; if a node holds a certificate signed by a trusted CA, then it can use this certificate with arbitrary addresses.

21.3 Packet Filter Considerations

Setting up a packet filter for IPsec is fairly simple: We need to let IKE traffic at port 500/UDP through. In some cases we can filter by AH and ESP headers:

Debian Sarge *The* `ip6tables` *packet filter supports a matching extension* `esp` *that lets us filter by SPIs.*

FreeBSD 6.1 *There is no documented feature in* `pf` *to filter by IPsec headers, but apparently the* `proto` *keyword can be used.* 134

If we want to ensure that packets are authenticated, we may need to do so on the destination node using an appropriate policy.

If we use encryption across the packet filter, then we can't filter by port numbers or even protocols anymore. Neither can we filter by packet contents using "deep inspection" or application level gateways.