Implementing Internet Key Exchange (IKE)

Niklas Hallqvist
Applitron Datasystem AB
niklas@openbsd.org

Angelos D. Keromytis
Distributed Systems Lab, University of Pennsylvania
angelos@openbsd.org

Abstract

A key component of the IP Security architecture is the Internet Key Exchange protocol. IKE is invoked to establish session keys (and associated cryptographic and networking configuration) between two hosts across the network. IKE needs to authenticate and authorize the parties involved in an exchange, negotiate parameters to be used for the communication, and interact with the local IPsec stack. The number of tasks, along with the flexibility built into the protocol, as well as the need to allow future additions and modifications to the protocol, need to be taken into consideration when designing and implementing IKE.

Another complicating factor is the need for security policy management. Although IKE can establish security associations with remote hosts, some method for determining what kinds of traffic can and should be exchanged with a remote host is necessary. As there is no standard specification yet, we are using a trust-management based approach using the KeyNote system as a basis for specifying policy.

This paper discusses the design, architecture, and implementation details of the OpenBSD IKE daemon, with separate mention of the security policy mechanism.

1 Introduction

The IP Security architecture [14], as specified by the IETF (Internet Engineering Task Force), is comprised of a set of protocols that provide data integrity, confidentiality, replay protection, and authentication at the network layer. This positioning in the network stack offers considerable flexibility in transparently employing IPsec in different roles (*e.g.,* in building Virtual Private Networks, end-to-end security, remote access, *etc.*). Such flexibility is not possible in higher or lower levels of abstraction.

The overall IPsec architecture is very similar to previous work [12] and is composed of three modules:

- The data encryption/authentication protocols [1, 2]. These are the "wire protocols," used for encapsulating IP packets to be protected. Outgoing packets are authenticated, encrypted, and encapsulated just before being sent to the network, and incoming packets are decapsulated, verified, and decrypted immediately upon receipt. These protocols are typically implemented inside the kernel, for performance and security reasons. A brief overview of the OpenBSD kernel IPsec architecture is given in Section 2.
- The key exchange protocol (IKE) [11] is used to dynamically establish and maintain Security Associations (SAs). An SA is the set of parameters necessary for one-way secure communication between two hosts (*e.g.,* cryptographic keys, algorithm choice, ordering of transforms, *etc.*). Although the wire protocols can be used on their own using manual key management, wide deployment and use of IPsec in the Internet requires automated, on-demand SA establishment.

  Due to the large number and variety of configurations and options an IKE implementation must support, this part of the IPsec architecture tends to dominate the other two in terms of code size and complexity. The first part of this paper discusses the OpenBSD implementation of IKE.

- The policy module governs the handling of packets on their way into or out of an IPsec-compliant host. Even though the security protocols protect the data from tampering, they do not address the issue of which host is allowed to exchange what kind of traffic with what other host. While traditional packet filtering mechanisms, such as employed in modern firewalls, can be used (with minor modifications) in enforcing traffic policies, a higher-level mechanism for validating and configuring such filters is needed. The second part of this paper discusses the implementation of a security policy mechanism based on trust management [6] in the OpenBSD IPsec.

1.1 Paper Organization

The remainder of this paper is organized as follows. Section 2 outlines the OpenBSD IPsec architecture. Section 3 gives a brief overview of the IKE protocol, while Section 4 discusses the design and implementation of the OpenBSD IKE implementation, and Section 5 presents the security policy mechanism. Related and future work is presented in Section 6.

2 OpenBSD IPsec

IPsec in the OpenBSD kernel is implemented as just another pair of IP transport protocols (AH and ESP). Thus, incoming IPsec packets destined to the local host are submitted to the appropriate IPsec protocol for processing based on the protocol number in the IP header. The SA needed to process the packet is located in an in-kernel database using information retrieved from the packet itself. Once the packet has been correctly processed (decrypted, authenticity verified, *etc.*), it is re-queued for further processing by the IP module, accompanied by additional information (such as the fact that it was received securely) for use by higher protocols and the socket layer.

Outgoing packets require somewhat different processing. When a packet is handed to the IP module for transmission, a lookup is made in a modified version of the routing table (called Security Policy Database, or SPD, in the IPsec standards) to determine whether that packet needs to be processed by IPsec. If this is the case, the result of the lookup also specifies what SA(s) to use for IPsec-processing the packet. Once processed, the packet is then re-queued for transmission by IP. If no SA is currently established with the destination host, the packet is dropped and a message is sent to the key management daemon through the **PF_KEY** interface [16]. It is then the key management's task to negotiate the necessary SAs.

To manage the SA and SPD tables, we use the PF_KEY interface, which is similar in concept to the routing socket interface available in BSD. Both manual keying utilities and key management daemons (such as IKE or Photuris [13]) use this interface to communicate with the kernel.

A somewhat dated overview of the OpenBSD IPsec architecture is given in [15].

3 The IKE Protocol

IPsec provides a solution to the problem of securing communications. However, for large-scale deployment and use, an automated method for managing SAs and key setup is required. There are several issues in this problem domain: negotiation of SA attributes, authentication, secure key distribution, and key aging to name some. Manual management is complicated, tedious, error-prone, and does not scale. Standardized protocols addressing these issues are needed; IETF's recommended protocol is named IKE, the Internet Key Exchange. IKE is based on a framework protocol called ISAKMP and implements semantics from the Oakley key exchange, therefore IKE is also known as ISAKMP/Oakley.

The IKE protocol is unfortunately a rather complex one, with many modes and options. Furthermore, new extensions proposed result in a further increase in complexity. Interoperation has been a problem because of this, but we are beginning to see good interoperability in the mandatory parts of the protocol.

The IKE protocol has two phases: the first phase establishes a secure channel between the two key management daemons, while in the second phase IPsec SAs can be directly negotiated. The first phase negotiates at least an authentication method, an encryption algorithm, a hash algorithm, and a Diffie-Hellman [9] group. This set of parameters is called a "Phase 1 SA." Using this information, the peers authenticate each other and compute key material to use for protecting Phase 2. Depending on the protection suite specified during Phase 1, different modes can be used to establish a Phase 1 SA, the two most important ones being "main mode" and "aggressive mode." Main mode provides identity protection, by transmitting the identities of the peers encrypted. Aggressive mode provides somewhat weaker guarantees, but requires fewer messages and allows for "road warrior" [1] types of configuration using passphrase-based authentication.

The second phase is commonly called "quick mode" and results in a IPsec SA tuple (one incoming and one outgoing). As quick mode is protected by a Phase 1 SA, it does not need to provide its own authentication protection, allowing for a fast negotiation (hence the name). Optionally, a new Diffie-Hellman computation can be done, providing "Perfect Forward Secrecy". PFS is an attribute of encrypted communications allowing for a transient session key to get compromised without affecting the security of future keys negotiated under the same Phase 1 SA (in other words, all session keys are cryptographically independent).

4 OpenBSD IKE

During spring 1998, Ericsson Radio Systems was looking for technology that could secure general IP-traffic in networks of tens, maybe hundreds of thousands of participating hosts. Fairly soon it became evident that IPsec was the right approach, but it was not at all clear what IKE implementation to use. The IKE standard was still evolving, and available implementations were lacking in either functionality, portability, exportability, or scalability. After having been presented with the state of the IKE market, Ericsson agreed to fund the development of an IKE implementation written from scratch, *isakmpd.* The initial authors were Niklas Hallqvist and Niels Provos, both from the OpenBSD project.

4.1 Architecture

When reading the drafts (later RFCs) on IKE, it became clear the protocol was complex, with many degrees of freedom. It was also known that *isakmpd* would be ported to several platforms, each with different APIs to the IPsec stack. There were also a number of proposals for IKE extensions in varying stages of completion. All these facts pointed towards a very modular architecture with distinct APIs between the subsystems. To avoid development complexity, we also decided to map the concepts of the standards fairly directly onto internal data structures.

Given how *isakmpd* would work (accepting inbound packets, doing some processing in the packet-prescribed context, sending a reply), it felt natural to build a message-based event-driven application. Thus *isakmpd* looks like most Unix UDP servers, with a main loop consisting of a select call followed by a multiplexor calling the right handlers for the occurring events.

The most common event is packet arrival, handled by the message module which is also responsible for packet validation and context lookup. Another fairly common event is the timeout, dealt with by the timer module. There are also application events, which are upcalls from the controlled application, in our case the IPsec stack. The design of *isakmpd* allows for other such "applications" in the future. This is the reason why it is called *isakmpd,* instead of *iked.* IKE is just one possible instantiation of the ISAKMP framework. The upcalls are dealt with by the application module, which to a great extent consists of system-dependent code dealing with the IPsec stack at hand. Currently, there exist three application back-ends, PF_KEY, PF_ENCAP and FreeS/WAN's NetLink API.

For controlling *isakmpd* there are a couple of modules worth mentioning. The "user interface" (*UI*) module listens for asynchronous events that control different aspects of *isakmpd,* like debugging level, active connections *etc.* This is currently done through a FIFO, but the design allows use of sockets or some other IPC mechanism. There is also a configuration module dealing with configuration file parsing, as well as lookups and overrides (via UI) of configuration entries. Last but not least, there is a policy module controlling what kind of SAs are allowed to be negotiated and by whom (see Section 5).

[1] Remote mobile users that need to access the protected network behind a firewall, using IPsec.

As ISAKMP is a transport-neutral protocol, there is also a transport module, which is actually an abstract class in an object-oriented view. Since IKE only requires UDP as the transport mechanism, there is just one derived class, the *udp* class. Finally, there is also a low-level network interface module which provides interface-walking, *etc.*

As all ISAKMP packets belong to "exchanges," we chose to create an exchange abstraction which was mainly a script engine and a data structure accumulating context state to later be carried over into SAs. Therefore, there are exchange and SA modules. They deal with creation, lookup, maintenance, aging, and destruction of these structures. Each exchange has a "script," which is walked for every packet received or transmitted. This makes it easy to create a source file per exchange type, making the code well modularized.

Independent of what exchange is used, there are a lot of common operations that need to be carried out during a negotiation. For this purpose we created separate modules for authentication, encryption, hash computation, and Diffie-Hellman computation. These in turn need more basic modules, like random number generation, long integer math, group math of both *modP* and elliptic curve kinds, and *X.509* certificate management [7].

Lastly, there are miscellaneous modules dealing with things like dynamic loading of code, logging, *etc.*

4.2 Component Description

- The message module.

  This module provides an abstract data-type representing individual ISAKMP messages. Internally, the messages are subdivided and indexed by payload type. Exported functionality consists of creation/destruction, incremental payload addition, parsing, validation, and context lookup of incoming messages, registering of post-send functions, transport-independent send logic, and message debugging. There is also generic SA negotiation logic which is covered in the implementation details section below. The reason for this logic being here is because it is driven by the physical message layout.

- The timer module.

  A fairly simple module accepting registration of functions to call at specific times together with their actual parameter. In order to get the functions called, the time module exports a function that calculates the timeout parameter to give the *select* call of the main loop, as well as the actual timer run function. Removal and reporting (for debugging) of timers is also supported.

- The application modules (*app, pf_encap, pf_key, etc.*)

  These modules deal with the communication with the application for which *isakmpd* is negotiating SAs. Currently, only one application is supported, IPsec. Communication with it occurs through various system-dependent APIs. Operations that need to be supported include getting a fresh SPI, creating an SA, updating a "larval" SA, grouping SA bundles, and, finally, removing SAs. Also needed is a means for telling the IPsec stack that ISAKMP traffic needs to be unencrypted. In OpenBSD, this is achieved by setting the appropriate *setsockopt(3)* options in the *isakmpd* socket.

- The network modules (*transport, udp* and *if*).

  The transport module exports an abstract data-type representing a specific transport. It has an associated function pointer table, just like the common *vtables* that C++ compilers create in order to implement polymorphism. Thus the transport structure is really a base class for the real transport classes. There is just one such class at the moment, the *udp* class. Exported functionality consists of creation/destruction (or rather reference/dereference as they are ref-counted) of transports, getting file descriptors ready for I/O to use in the select loop of *main(),* as well as checking them for I/O possibility afterwards. Message sending and reception methods are exported as well, along with endpoint address determination.

- The UI module.

  This module is really just a simple command line interpreter. It conveniently accepts commands asynchronously through a one-way FIFO (named pipe). The commands are rudimentary, one letter with a few parameters each. The existing controls deal with issues like debugging, SA management, and dynamic changes to the configuration database.

```
int (*ike_main_mode_initiator[]) (struct message *) = {
  ike_phase_1_initiator_send_SA,
  ike_phase_1_initiator_recv_SA,
  ike_phase_1_initiator_send_KE_NONCE,
  ike_phase_1_initiator_recv_KE_NONCE,
  initiator_send_ID_AUTH,
  ike_phase_1_recv_ID_AUTH
};
```

Figure 1: The Initiator Main Mode script

- The configuration module.

  isakmpd maintains a configuration database consisting of section/tag/value triplets, *i.e.* it maps closely to a well known format called ".INI". This configuration database is primed from the configuration file (.INI-style) at program start, and every time a HUP signal is sent to the *isakmpd* process. It is also possible to dynamically alter the database via the UI module. There is functionality to treat the value of a triplet as a comma-separated list, and easily "walk" that list. Otherwise, ordinary database operations like creation, lookup, and removal of entries are exported.

- The policy module.

  See section 5 for a description of this module. This module exports only one function, which is called to validate a combination of SA proposal, remote peer identity, and packet selectors (Phase 2 IDs).

- The exchange module.

  A key abstraction in *isakmpd* is the exchange. This is the engine that drives the negotiations towards SA establishment. Exchanges form the context of all negotiations, and closely map to the exchange concept of the RFCs. Every exchange is a well-defined, fixed-length sequence of messages between the two peers. Every individual message also has a well-defined minimum content of payloads. This structure of exchanges lends itself to implementation as a generic finite state machine driven by "scripts" supplied by each exchange type. These scripts provide the actions to execute at message reception as well as before/after message transmission. It is also easy to have a generic "syntax checker" inspecting each message, ensuring the required payloads are present. This module's exported API consists of functions for establishing exchanges when acting as initiator, as well as setting up exchanges for "incoming" negotiations. There are also several lookup functions, finding exchanges using different criteria.

- The SA module.

  Just like the IPsec kernel, *isakmpd* needs to maintain its own SA database. This database actually consists of both ISAKMP SAs, which are the results of Phase 1 negotiations, and application SAs from Phase 2. Every SA has attached DOI-dependent (Domain Of Interpretation) data, should we ever need to support other DOIs than IPsec. The SA structure contains both the on-the-wire representation of the SA, as well as internal per-SA data. SAs are created when the negotiation starts, but are inactive until an exchange finalization routine is run. The SA API is mostly a set of life maintenance functions, *i.e.* creation, ref-counting, expiration setup, and destruction operations. Similar to the exchange module, a fairly versatile set of lookup functions is available.

- The authentication module.

  IKE allows for several kinds of authentication. An authentication method needs to provide just three functions: generation of a shared secret the peers derive keys from, encoding of a keyed hash proving the authenticity of the peer, and decoding of such a hash thereby verifying the other peer's authenticity. Currently *isakmpd* supports the mandatory pre-shared key authentication method, as well as certificate based (X.509) RSA signature authentication. We plan to support public key encryption-based authentication in the near future.

- Cryptography and math.

  Isakmpd builds upon some basic cryptographic and mathematic components.

  - Ciphers.

There is a collection of ciphers which can be used interchangeably to protect the data that goes on the wire. It is natural to implement these ciphers as subclasses to a "crypto" base class, which provides hooks for initialization, cloning, and updating of key state, as well as encryption and decryption of data. The separation of key state management from the actual algorithm applications is important for maintaining cryptographic synchronization between the peers. *isakmpd* implements the following algorithms: DES, 3-DES, CAST, and Blowfish.

- Hashes.

  As was the case with ciphers, it is also a design requirement that hash algorithms be easy to alter. Thus, hash algorithms are also implemented as subclasses of a generic hash class, providing a simple API for incremental hash computation of concatenated data.

- Diffie-Hellman.

  The Diffie-Hellman algorithm is a means of establishing a shared secret between two peers without exposing sufficient data for wire-tappers to compute that secret. The API is simple, since only two functions are needed: creation of a local random big-integer, and computation of the actual secret based on the local big-integer and a similar-type value received from the peer.

- Group mathematics.

  The mathematical basis for Diffie-Hellman is called group math. Groups are big-integer arithmetic systems with a few parameters. It turns out that groups are also suitable to implement in an object-oriented fashion, as there are different algorithms that comply with the group math requirements. In *isakmpd,* there is support for two kind of groups, elliptic curves and *modP* groups.

- Big integer mathematics.

  Both group mathematics and the public key cryptography used in the authentication and policy modules, need big-integer math. We currently use OpenSSL's BN functions as well as a few supplementary routines written by us. We have however made the underlying math library exchangeable so other math libraries can be used if needed. We currently support FSF's GMP but we also intend to take advantage of hardware support for big-integer operations, since such products have begun to make their appearance in the market.

- The dynamic loader module.

  Perhaps a less obvious component to have in a daemon like *isakmpd* is a module for dynamic loading and linking of code. The reason for this module is mainly due to the RSA patent; we cannot ship RSA code in OpenBSD as the license-free implementation cannot be imported to the United States. Therefore, we dynamically load that support if it is available (the supporting libraries can be fetched separately, different versions for different countries). This module exports a function that takes a dynamic load script, written in a very simple language we designed, that describes what files should be loaded and what symbols should be resolved.

- The log module.

  Logging is crucial in security applications. It is also important that developers of security software are presented with debugging tools that help them find bugs faster. We consider logging to be such a tool, if it can be controlled in a fine-grained way. This module exports functions to change the levels per logging class, to control where logging information goes and, naturally to actually log both binary and textual buffers.

- The system-dependent module.

  In order to maintain portability, every function that may need differing implementations depending on the platform, needs to be placed in a central, exchangeable, system-dependent module. Most often, functions placed here are glue or proxies.

4.3 Implementation Details

4.3.1 The Exchange Script Machine

An IKE exchange normally consists of a fixed number of well-defined messages, which each peer sends every other turn. Recognizing this simple fact, we chose to build the state machine around an engine which ran "scripts" unique for each exchange type. An example of a script is shown in figure 1.

This is the script an initiator runs when doing a "main mode". The elements of the script are functions, alternately constructing a message to be sent, or dealing with a message that has been received. Along with this semantics description there is also a syntactic "script", which may look like figure 2. This syntax description describes what payloads are mandatory in each message of the exchange. It also marks when the exchange ends.

4.3.2 Configuration

Configuring IKE is an involved process, due to IKE being a complex protocol. When we were faced with the problem of how to design the configuration language we tried a few simplistic approaches, but they soon turned out to be too inflexible. Thus we decided to use a rather generic configuration syntax which we could fit in everything we wanted. The syntax would also allow for easy dynamic modification of the internal configuration information without reloading a full file. The caveat is that our configuration syntax maps much better to the machine and protocols than to a human being administering *isakmpd*. Our plan was to get someone else write a "real" configuration file format that could be translated into our style. So far no one has taken the bait. Note that ideally, very little configuration should be needed for *isakmpd*; most of the information should be provided on-the-fly by the kernel (at least in the end-to-end case), or through some security policy discovery mechanism.

The file format is commonly known as .INI-format, and a snippet is shown in figure 3. Internally, everything is treated as (section, tag, value) triplets, where the values can optionally be lists of scalar values. The values themselves are often section names thereby giving a tree (or rather a forest) structure to the data.

As we have already mentioned, the internal configuration is dynamically alterable. We saw a need for several "users" altering the configuration concurrently, so we made the API transactional. Each transaction can contain several modifications to the configuration, and they are atomically introduced.

Internally there is also an API to get the actual configuration values. Because of this, it is considered very easy to move the configuration database into other internal formats or even externalize it.

4.3.3 Portability Considerations

From its conception, there was a portability requirement in *isakmpd*. It should run on various platforms, and with different IPsec stacks. Because of this demand, the "sysdep" module was introduced. Each platform we support needs to provide its own version of this module. In principle, all of the IPsec API could be dealt with here, but as APIs can be shared among several platforms (and there even exist standards now), most often the sysdep module only has stub code to call the right API module, like PF_KEY.

PF_KEY may become a standard, but it is only an API for maintaining SAs, and IPsec also needs policy maintenance. All PF_KEY systems we support have chosen to add policy extensions to PF_KEY because of the fact that the API is flexible enough to pass such data as well, and it is easier to extend something working than to invent something entirely new. However, extensions tend to be platform specific, so the PF_KEY support code in *isakmpd* has to deal with several different variants of the protocol. This problem is recognized, and there actually is some consensus between OpenBSD, KAME, and FreeS/WAN that this needs to change, and that the extensions need to converge, if not even be standardized.

With respect to differences in the build environment, we have seen a need to support both main "make" dialects, BSD and GNU. This is of course less than optimal, but given the alternatives it is currently our best option. Furthermore, every supported platform has to provide a makefile fragment wherein constraints on what *isakmpd* should support on that particular platform can be expressed, as well as instructions on how to build system-dependent code.

4.3.4 Debugging Support

Being a security critical application, it is vital *isakmpd* be as bug-free as possible. All software contains bugs, and all development creates new ones. Recognizing that, we have chosen to make debugging a more pleasant task than it usually is. Normally *isakmpd* detaches from the controlling terminal and logs only exceptional conditions to the syslog facility. However, in order to be able to run under a normal debugger, it is possible to run in

```
int16_t script_identity_protection[] = {
  ISAKMP_PAYLOAD_SA,/* Initiator -> responder.  */
  EXCHANGE_SCRIPT_SWITCH,
  ISAKMP_PAYLOAD_SA,/* Responder -> initiator.  */
  EXCHANGE_SCRIPT_SWITCH,
  ISAKMP_PAYLOAD_KEY_EXCH,/* Initiator -> responder.  */
  ISAKMP_PAYLOAD_NONCE,
  EXCHANGE_SCRIPT_SWITCH,
  ISAKMP_PAYLOAD_KEY_EXCH,/* Responder -> initiator.  */
  ISAKMP_PAYLOAD_NONCE,
  EXCHANGE_SCRIPT_SWITCH,
  ISAKMP_PAYLOAD_ID,/* Initiator -> responder.  */
  EXCHANGE_SCRIPT_AUTH,
  EXCHANGE_SCRIPT_SWITCH,
  ISAKMP_PAYLOAD_ID,/* Responder -> initiator.  */
  EXCHANGE_SCRIPT_AUTH,
  EXCHANGE_SCRIPT_END
};
```

Figure 2: The syntax of an ID_PROT exchange

```
# Incoming phase 1 negotiations are multiplexed on the source IP address.

[Phase 1]
192.168.0.1= ISAKMP-peer-node-0

[ISAKMP-peer-node-0]
Phase= 1
Transport= udp
Address= 192.168.0.1
Configuration= Default-main-mode
Authentication= yoursharedsecretwith0

[Default-main-mode]
DOI= IPSEC
EXCHANGE_TYPE= ID_PROT
Transforms= 3DES-SHA,3DES-MD5

[3DES-SHA]
ENCRYPTION_ALGORITHM= 3DES_CBC
HASH_ALGORITHM= SHA
AUTHENTICATION_METHOD= PRE_SHARED
GROUP_DESCRIPTION= MODP_1024
Life= LIFE_600_SECS

[LIFE_600_SECS]
LIFE_TYPE= SECONDS
LIFE_DURATION= 600,450:720
```

Figure 3: Configuration entry samples

the foreground, sending logging messages to *stderr* instead. As we have already mentioned, the logging module has a fine-grained control mechanism making it easy to chose detailed information on certain topics. In order to ease problem pinpointing, almost every intermediary computation can be logged.

The build environment also contains instructions on how to build *isakmpd* with two different memory allocation debugging tools: ElectricFence, for finding buffer overflows and use after deallocation, and Boehm's garbage collector to find memory leaks. We periodically run with these tools to test for such problems.

4.3.5 Addressing Denial of Service Attacks

IKE is subject to DoS (Denial of Service) attacks since state has to be kept in the responder after the first message has been received. If a malicious peer starts flooding *isakmpd* with exchange initiations, a lot of state will accumulate in the responder. Worse yet, in aggressive mode, the responder will have to do expensive computational work [2] before the peer has been authenticated. These issues are actually protocol problems and could have been moot, if only the "cookie" mechanism adopted from the Photuris protocol had been understood and used correctly [13, 17]. Since the protocol has been standardized, we need to address the potential attacks. Our approach is twofold: first off, we always check memory allocation for failure, and back out, cleaning up all resources tied in with the message we are re dealing with. Second, we use a maximum, configurable, exchange lifetime. If the exchange times out, all resources are given back to the system.

We have considered additional measures, like aggressive random tail drop of exchanges stuck in the state after the first reply. This would be somewhat analogous to the normal response to TCP SYN-floods.

4.3.6 Solving the RSA "problem"

At the time we started implementing *isakmpd,* exporting a US RSA implementation in source form to the world at large was illegal. Another problem was that it is not legal to use the RSA algorithm within the US unless one has a license from RSA Inc. or use the US-originated non-commercial RSAREF library. Thus, there was no way to make a distribution that would be free to use both in the US and in the rest of the world, because the only implementation that is free in the US was not exportable. OpenBSD has solved this problem in other places of the source tree in an elegant way: we chose to use all RSA functionality via a dynamically linked shared library, *libcrypto,* which is part of OpenSSL. This library exists in three variants: one RSA-crippled, with no RSA support at all, one with internationally written RSA code and one with RSAREF. We ship the RSA-crippled version as that one has no patent or exportability issues at all. Then we tell international users to fetch the international libcrypto version, and US users to get the one based on RSAREF (if they meet criteria to legally use it).

This could work for *isakmpd* too, if it were not for the fact that we want *isakmpd* to be statically linked, so we can get IKE negotiation capabilities really early in the boot process.

The solution was to use dynamic linking via the *dlopen* API. Every RSA-related symbol of libcrypto needs to be accessed indirectly through a pointer. This pointer is initialized with the address of the statically linked RSA-crippled stubs. After a successful dynamic link the pointers get reset to the newly loaded libcrypto equivalents. It is not considered a fatal error if the dynamic linking fails. Not all operating systems allow statically linked binaries to use *dlopen* though, but those who do can benefit from this.

4.3.7 Performance and Code Size

The SA negotiation is very CPU-intensive. More specifically, in main and aggressive mode there is always a Diffie-Hellman exponentiation and sometimes, depending on authentication method, RSA or DSS signature operations that are fairly expensive in terms of CPU processing. In quick mode, the DH exponentiation is optional but recommended. That exponentiation is what provides "Perfect Forward Secrecy." Some sample timings can be found in figure 4.

In its current state, *isakmpd* consists of roughly 36,000 lines of code, almost all of it in C. This in-

[2] Even hardware accelerators for big number computation cannot handle the high volume of operations that would be involved in such a DOS attack.

Exchange	Seconds
Main mode, 3DES, SHA, DH group 2, pre-shared key	1.44
Quick mode, 3DES, SHA, PFS (DH group 2)	1.40
Main mode, DES, MD5, DH group 1, pre-shared key	0.95
Quick mode, DES, MD5, PFS (DH group 1)	0.60
Aggressive mode, 3DES, SHA, DH group 2, RSA signature (X.509)	1.50
Quick mode, 3DES, SHA, no PFS	0.35

Figure 4: A Pentium 200MHz running two instances of isakmpd negotiating over the loopback interface (an exchange between two distinct machines may actually finish faster as some computations can be carried out in parallel).

cludes commentary, which we have at least tried to be fairly generous with. Security protocol implementations need to be auditable, and readability is therefore an important aspect. 4,000 of these are the platform-dependent parts, and 2,500 are regression testing. The static memory footprint for i386 is approximately 950KB for a full-blown version and 300KB for a trimmed down version with support only for mandatory ciphers, exchanges, groups, and authentication methods (no debugging or refined policy handling is included in the trimmed-down version).

5 Security Policy

When discussing security policy, it is often useful to define the term in the appropriate context. For our purposes, security policy in the network layer is the information needed to decide whether a packet should be accepted/forwarded or dropped. Further restricting the definition in the IPsec context, security policy dictates what classes of packets are acceptable over a specific SA. This is all the more important for IPsec, since the encapsulation mechanism used literally allows establishment of arbitrary virtual topologies over the network fabric.

Since there exists no standard mechanism for specifying, disseminating, and processing security policy for IPsec, we have adopted some ongoing research work based on a compliance-checking architecture. The concept behind this architecture is that, at SA establishment time, we utilize some mechanism that validates the suitability of an SA for a particular class of packets and a remote principal at IKE exchange time; all the characteristics of the SA (cryptographic algorithms, key sizes, transform ordering, *etc.*), along with the packet classes (in effect, a set of packet filter rules) and the remote principal's identity (public key, X.509 certificates, passphrase, *etc.*) are available at that stage. It is important to realize that this operation is performed only infrequently compared to the number of packets that will use the established SAs. Thus, it is possible to use a mechanism that is more general, powerful, and extensible than a simple packet filter specification language. We would also like to be able to utilize credentials delegating authority, as we have found these to allow easier and more scalable administration.

The higher-level mechanism for security policy compliance-checking we use is a trust-management system. Trust-management systems [5, 4] provide a unified approach to specifying security policies, credentials, and relationships between principals in the system. Unlike traditional certification schemes, trust-management credentials bind keys directly to the authorization to perform some task. A trust-management system provides a highly-adaptable general-purpose mechanism for specifying security policies and credentials. A principle of trust management is "monotonicity." This means that policies and credentials can only have a positive effect on the privileges of a principal; it is not possible to revoke privilege by issuing a credential. This may only be done by expiring credentials, or by modifying the relevant policies and credentials. For an extensive overview of trust-management, see [3].

KeyNote is an instantiation of a trust-management system, designed to be simple yet flexible. It provides a single language for both policies and credentials, based on predicates that describe the trusted actions permitted by holders of specific public keys (or other cryptographic identifiers). For more details on KeyNote syntax and processing, see [4]. For more details on the policy architecture itself, see [6]. The following subsection discusses some implementation specifics.

5.1 Implementation Details

Modifying *isakmpd* to make use of the compliance-checking architecture for policy resolution proved straightforward. *isakmpd* was initially designed with a rudimentary mechanism for verifying security associations proposed by the remote peer. The set of acceptable security associations was read from the configuration file, and then consulted when examining the proposed SA. However, this scheme lacked flexibility and extensibility. In particular, it was not possible to delegate authority, allow for very fine-grained SA specification without an explosion in the size of the configuration file, take into consideration information not directly relevant to the SA (such as time of day, or system security level), nor allow for flexible packet selectors (an exact match was required).

Since this verification mechanism was implemented as a procedure call, we only had to modify the invoking code to call another procedure that ultimately invoked KeyNote. This change occurred in two places:

1. When the Responder of an IKE exchange examines the list of IPsec (Phase 2) SAs to determine which one is acceptable.
2. When the Initiator receives (during Phase 2) the response containing the acceptable SA.

When invoked, the procedure converts information taken from the *exchange* and *sa* structures to a format suitable for use by KeyNote. Such information contains the IPsec protocols to be used, the cryptographic algorithms to be used, the packet selectors requested (Phase 2 User IDs), the cryptographic identifier used in Phase 1 by the remote peer, *etc.*

This cryptographic identifier is used by the compliance checker to determine which part of the security policy is relevant to a specific request. If public key authentication was used, then our security policy may directly refer to said public key, and the same applies for passphrase authentication. For X.509-based authentication, we have a number of options as to who policy may refer to:

- The public key of the remote principal as it appears in the Subject field of the X.509 certificate, or the X.509 certificate itself. This form of delegation is the most direct and limited in scope.
- The public key or X.509 certificate of some certification authority (CA) that ultimately "speaks for" the remote principal. This may be the CA immediately validating said principal, or some other CA further up in a CA hierarchy. The higher up the CA we delegate to, the broader the scope of the delegation (and thus, more users share the same rights). Note that it is possible to delegate a set of rights to some CA that "speaks for" some user, and simultaneously give more rights to that specific user. Reducing a user's privileges through the same mechanism is not feasible under KeyNote, however (because of monotonicity, as previously described).
- Since public keys and X.509 certificates can be cumbersome to manipulate even in a text form, it is possible to use the Distinguished Name as it appears in an X.509 certificate. This makes policies much more concise and readable. An added benefit is that certificates (and even keys) may change without affecting the policy (although in some cases this may turn into a liability). We can use the DN of the remote principal directly, or that of some CA that "speaks for" the principal.

The assembled information is passed on to KeyNote, and the response indicates whether the SA should be accepted or dropped. In effect, KeyNote is verifying that the combination of remote peer, IPsec protocols (and algorithms, lifetimes, *etc.* used by those protocols), and packet selectors are acceptable by policy. This policy may be expressed solely in terms of local policy or as a combination of local policy and (signed) credentials. These credentials may be acquired during the Phase 1 exchange (provided by the remote peer) or at any point in time afterwards (*e.g.,* fetched on-demand through some out-of-band protocol[3]). As soon as an SA is accepted, the search is concluded.

The procedure is called once for each distinct SA proposal received from the peer (since there is no

[3] We have experimented with fetching credentials from a web server, using a primitive cgi-script and a database keyed on public keys and X.509 Distinguished Names.

way to efficiently encode all the SA proposals in one action attribute set and have KeyNote make a decision on which one to select – this is a drawback of using KeyNote instead of a more complex policy language). Note however that each such invocation is very "lightweight" in processing terms: converting the relevant information is straightforward, and any cryptographic operations are only performed once and their results cached for future use. The policy assertions are loaded once at startup time (and reloaded if *isakmpd* is asked to re-initialize). Some simple experiments show that the cost of invoking KeyNote increases linearly with the number of assertions in use, and that for a simple setup of 3-4 assertions/credentials the cost is in the order of 150μsec.

Here, we wish to make two additional observations:

- KeyNote is invoked during Phase 2 only. While it is trivial to allow policy control over establishment of Phase 1 SAs, we believe that this is both unnecessary and potentially confusing to users. Since Phase 1 SAs are used only by *isakmpd* and have no direct effect on the system or on network traffic, this approach does not compromise safety.
- Currently, compliance checking on the initiator is performed when the accepted SA is received from the responder (message 2 in Quick Mode). Ideally, this check should be done before transmission of the first message in Quick Mode, to avoid transmitting SA proposals that in the end will not be accepted by us. Processing after receipt of message 2 should be limited to verifying that the returned SA is among those offered in the first message. We elected not to do this because of code complexity: because KeyNote support was added after most of *isakmpd* was written, the code that constructs the list of SAs in message 1 was already intricately tied to message construction, configuration file parsing, and attribute syntax verification. Rewriting the relevant code just to accommodate KeyNote would involve serious restructuring. We intend to rewrite that piece of *isakmpd* in the near future to retrieve SA information from the kernel (as opposed to a configuration file). At that time, an interface better suited to policy compliance checking will be introduced. We should note that this issue is not an artifact of our use of KeyNote; using any security policy system on the initiator side would require the same code restructuring.

In terms of code size, the "glue" code between *isakmpd* and KeyNote was about 1200 lines, almost exclusively dealing with the conversion of information from *isakmpd*'s internal structures to KeyNote action attributes. We also had to add about 50 lines of code in different parts of KeyNote, dealing with initialization and record keeping. The code displaced by KeyNote was approximately 500 lines long. The KeyNote library itself is about 5000 lines (not including the cryptographic functions, where *libcrypto* is used).

6 Conclusion

6.1 Current State

We believe that *isakmpd* currently addresses all mandatory features in the RFCs. We also implement most optional features. *isakmpd* currently runs on OpenBSD's old IPsec stack with PF_ENCAP, OpenBSD's current stack with PF_KEY, FreeS/WAN with Linux NetLink API and FreeBSD/NetBSD with KAME's IPsec stack via PF_KEY. We have also made it possible to shave off much of the extras at compile time, thus making *isakmpd* a candidate for being used in small embedded systems. *isakmpd* is in production used in numerous sites.

6.2 Future Directions

There seems to be an increasing number of proposed new IKE extensions after every IETF. We are, however, reluctant to incorporate them all as code bloat is a problem we should fight to maintain any kind of security. Something we definitely are going to add is IPv6 support, as we recently have started shipping OpenBSD with an IPsec-aware IPv6 stack. Other likely enhancements are support for PKCS#11 (an API to talk to cryptographic tokens, like smartcards, for authentication), challenge-response authentication for Phase 1 exchanges and PKIX compliance. A major short-term project is support for cryptographic hardware for RSA and Diffie-Hellman computation, since OpenBSD has began to support a cryptographic services framework in the kernel. Other minor projects involve integration with DNSSEC [10] infrastructure once we see further deployment and use, and "New group mode" support to dynamically negotiate new groups to compute DH secrets in. There are plans to support some new platforms, for example FreeS/WAN over PF_KEY and Solaris 8. There are other commercial Unices with IPsec stacks which we may port *isakmpd* to. Closer integration with the kernel and userland applications (possibly through the *setsockopt(3)/getsockopt(3)* API), and various projects involving policy discovery/negotiation (in particular, direct exchanging of KeyNote credentials) and automatic configuration are also part of our plans for future work.

6.3 Interoperability

We have attended a couple of interoperability workshops as well as carried out our own tests and have succeeded remarkably well, given the complexity of the IKE specifications. A lot may be attributed to our flexible configuration which, however, cannot be said to be user-friendly. We have been known to interoperate with the 3com Pathbuilder 500, Ashley Laurent VPCom, Axent Raptor, Cendio Fuego Firewall, CheckPoint FireWall-1, Cisco IOS, Cisco PIX, F-secure VPN+, FreeBSD/NetBSD KAME, Intel LanRover, Linux FreeS/WAN, Nortel Contivity, PGP VPN, Radguard cIPro, Teamware TWISS, Windows 2K, and Timestep Permit.

Most of this interoperation has been with pre-shared keys. Unfortunately we have not yet had a chance to do extensive certificate-based interoperability testing.

6.4 Security Considerations

As might have become clear by now, IKE is a complex protocol, perhaps overly so. As we are implementing security, complexity is not something well looked upon. Complex protocols are implemented with complex programs which tend to have more bugs, and some bugs might just happen to be security breaches. Modular design with clear APIs internally helps reduce complexity and allows for easier auditing, but there is still a lot more risk with complex programs than with simple ones. There are simpler alternatives to IKE, more limited in functionality, but likely more secure [13].

6.5 Related Work

There are of course other Open Source projects that implement IKE, the two most widely known being the Linux FreeS/WAN project's *Pluto,* and *Racoon,* of the KAME project whose IPsec stacks exist for both NetBSD and FreeBSD. Both of these are only meant for their respective platforms, unlike *isakmpd,* which is meant to be a portable implementation. As a matter of fact, *isakmpd* runs on top of both the FreeS/WAN and KAME stacks. Racoon is, to our knowledge, the only IKE implementation with IPv6 support. There are also other key-management protocol implementations available, an example is *photurisd,* OpenBSD's Photuris implementation. An extensive overview of the employment of cryptography in OpenBSD may be found in [8].

7 Acknowledgments

We would like to thank Matt Blaze, Theo de Raadt, Martin Fredriksson, Markus Friedl, Hugh Graham, John Ioannidis, Håkan Olsson, Niels Provos, and Jonathan Smith for their support, comments, suggestions, and work in various aspects of this project and paper. Most of the development of *isakmpd* was funded by Ericsson Radio Systems. The security policy work mentioned in this paper was supported by DARPA under grant F39502-99-1-0512-MOD P0001.

8 Availability

All the software described in the paper is available through the OpenBSD web page at:

`http://www.openbsd.org/`

OpenBSD is based in Calgary, Canada. All individuals doing cryptography-related work do so outside countries that have limiting laws.

References

[1] R. Atkinson. IP Authentication Header. RFC 1826, August 1995.

[2] R. Atkinson. IP Encapsulating Security Payload. RFC 1827, August 1995.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer-Verlag Inc., New York, NY, USA, 1999.

[4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The keynote trust management system version 2. Internet RFC 2704, September 1999.

[5] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, Los Alamitos, 1996.

[6] M. Blaze, J. Ioannidis, and A. Keromytis. Trust Management and Network Layer Security Protocols. In *Proceedings of the 1999 Cambridge Security Protocols International Workshop*. Springer, 1999.

[7] Consultation Committee. *X.509: The Directory Authentication Framework*. International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.

[8] T. de Raadt, N. Hallqvist, A. Grabowski, A. D. Keromytis, and N. Provos. Cryptography in OpenBSD: An Overview. In *Proc. of the 1999 USENIX Annual Technical Conference, Freenix Track*, pages 93 – 101, June 1999.

[9] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, Nov 1976.

[10] D. Eastlake and C. Kaufman. Dynamic Name Service and Security. Internet RFC 2065, January 1997.

[11] D. Harkins and D. Carrel. The internet key exchange (IKE). Request for Comments (Proposed Standard) 2409, Internet Engineering Task Force, November 1998.

[12] John Ioannidis and Matt Blaze. The Architecture and Implementation of Network-Layer Security Under Unix. In *Fourth Usenix Security Symposium Proceedings*. USENIX, October 1993.

[13] P. Karn and W. Simpson. Photuris: Session-key management protocol. Request for Comments (Experimental) 2522, Internet Engineering Task Force, March 1999.

[14] S. Kent and R. Atkinson. Security architecture for the internet protocol. Request for Comments (Proposed Standard) 2401, Internet Engineering Task Force, November 1998.

[15] A. D. Keromytis, J. Ioannidis, and J. M. Smith. Implementing IPsec. In *Proceedings of Global Internet (GlobeCom) '97*, pages 1948 – 1952, November 1997.

[16] D. McDonald, C. Metz, and B. Phan. PF_KEY Key Management API, Version 2. Request for Comments (Informational) 2367, Internet Engineering Task Force, July 1998.

[17] W. A. Simpson. IKE/ISAKMP Considered Harmful. *USENIX ;login:*, December 1999.