Cryptography in OpenBSD: An Overview

Theo de Raadt, Niklas Hallqvist, Artur Grabowski, Angelos D. Keromytis, Niels Provos
{deraadt,niklas,art,angelos,provos}@openbsd.org
The OpenBSD Project

Abstract

Cryptographic mechanisms are an important security component of an operating system in securing the system itself and its communication paths. Indeed, in many situations, cryptography is the only tool that can solve a particular problem, *e.g.,* network-level security. While cryptography by itself does not guarantee security, when applied correctly, it can significantly improve overall security. Since one of the main foci of the OpenBSD system is security, various cryptographic mechanisms are employed in a number of different roles.

This paper gives an overview of the cryptography employed in OpenBSD. We discuss the various components (IPsec, SSL libraries, stronger password encryption, Kerberos IV, random number generators, *etc.*), their role in system security, and their interactions with the rest of the system (and, where applicable, the network).

1 Introduction

An important aspect of security in a modern operating system is cryptographic services and mechanisms. While not a security panacea, cryptography is sometimes the right tool in solving certain problems. In particular, cryptography is extremely useful in solving a number of security issues in the following three areas:

- Network security.
- Secure storage facilities.
- (Pseudo-) Random number generators.

Since one of our goals in the OpenBSD project is to provide strong security, we have implemented a number of protocols and services in the base system. An OpenBSD distribution thus has full support for such mechanisms as IPsec, SSL, Kerberos, *etc,* being unaffected by export restriction laws.

Simply supporting these mechanisms, however, is not sufficient for wide-spread use. We are constantly trying to make their use as easy and, where possible, transparent to the end user. Thus, more work is done in those mechanisms that can be used to provide transparent security, *e.g.,* IPsec.

With this paper, we intend to give a good overview of the cryptography currently distributed and used in OpenBSD, and of our plans for future work. We hope this will be of interest both to end-users and administrators looking for better ways to protect their host and networks, and to developers in other systems (free or otherwise) that are considering supporting some of these mechanisms. We should again caution the readers, however, that cryptography does not solve all security problems in an operating system, and should not be considered as an end in itself, but rather as an important piece of the security puzzle.

1.1 Paper Organization

The remainder of this paper is organized as follows: section 2 describes the various network security facilities implemented and supported in OpenBSD, section 3 covers the extensive use of random number generators, and section 4 briefly outlines our future plans in this area. Section 5 concludes the paper.

2 Communications Security

In an increasingly networked environment, communications security support in an OS is extremely important. As there are different mechanisms and different layers where one may apply security, OpenBSD supports a number of security protocols and mechanisms, some of which were developed (or even designed) by our developers. In some cases, there is considerable overlap in functionality. One of our goals is to eventually make it transparent to the end user which such security mechanism is in use.

The following sections give a brief overview of these mechanisms, some detail of their implementation and integration in OpenBSD, and our plans for future work. As we already mentioned in section 1, we consider IPsec an extremely important tool in network security, both because of its potential for user-transparency and its flexibility. This is reflected by the more thorough coverage of IPsec in the text that follows.

Other popular mechanisms, such as SSH [38], are not covered because they are only part of our ports system. While virtually all the developers use SSH, there is no free implementation we can add to our standard distribution. Furthermore, the current version of SSH is restricted by the RSA patent in the US. We are waiting for a free implementation to become available as part of the IETF standardization process of SSH. Such an implementation would be linked with our `libssl`.

2.1 SSL

In OpenBSD `libssl` provides a toolkit for the Secure Socket Layer (SSL v2/v3) and Transport Layer Security (TLS v1) [6] which provide strong cryptographic protection for network communication such as server authentication and data encryption. The Secure Socket Layer is currently used by web servers, *e.g.,* Apache as shipped with OpenBSD, and browsers like Netscape Communicator. In the future, applications like *telnet* and *ftp* will be converted to use TLS, possibly even during our network installation process.

Due to patent restrictions, `libssl` in the OpenBSD distribution supports only digital signatures with DSA [27], but an additional package is provided for users outside the USA to add back RSA-signature [19] support. This is implemented by providing two shared libraries: `libssl.so.1.0` has only function stubs for RSA support, while `libssl.so.1.1` contains full RSA support. Notice that shared library minor-version number changes typically indicate interface-transparent bug fixes.

2.2 IP Security (IPsec)

2.2.1 Background

While IP has proven to be an efficient and robust protocol when it comes to actually getting data across the Internet, it does not inherently provide any protection of that data. There are no facilities to provide confidentiality, or to ensure the integrity or authenticity of IP [31] datagrams. In order to remedy the security weaknesses of IP, a pair of protocols collectively called IP Security, or IPsec [3, 16] for short, has been standardized by the IETF. The protocols are ESP (Encapsulating Security Payload) [2, 15] and AH (Authentication Header) [1, 14]. Both provide integrity, authenticity, and replay protection, while ESP adds confidentiality to the picture. IPsec can also be made to protect IP datagrams for other hosts. The IPsec endpoints in this arrangement thereby become security gateways and take part in a virtual private network (VPN) where ordinary IP packets are tunneled inside IPsec [36].

Network-layer security has a number of very important advantages over security at other layers of the protocol stack. Network-layer protocols are generally hidden from applications, which can therefore automatically and transparently take advantage of whatever network-layer encryption services that host provides. Most importantly, network-layer protocols offer a remarkable flexibility not available at higher or lower layers. They can provide security on an end-to-end (securing the data between two hosts), route-to-route (securing data passing over a particular set of links), edge-to-edge (securing data as it passes from a "secure" network to an "insecure" one), or a combination of these.

2.2.2 Operation

Central to both ESP and AH are an abstraction called security association, or SA. In each SA there is information (algorithm IDs, keys, *etc.*) stored describing how the wanted protection should be setup. For two peers to be able to communicate they need matching SAs at each end. When deciding what SA should be used for outbound traffic, some kind of security policy database needs to be consulted. In OpenBSD, this is currently implemented as an extension to the routing table, where source/destination addresses, protocol, and ports serve as selectors.

Looking at the wire format, IPsec works by inserting an extra header between the IP header and the payload. This header holds IPsec-specific data, such as an anti-replay sequence number, cryptographic synchronization data, and integrity check values. If the security protocol in use is ESP, a cryptographic transform is applied to the payload in-place, effectively hiding the data. As an example, an UDP datagram protected by ESP is shown in figure 1.

This mode of operation is called transport mode, as opposed to tunnel mode which is typically used when a security gateway is protecting datagrams for other hosts. Tunnel mode differs from transport mode by the addition of a new, outer, IP header consisting of the security gateways' addresses instead of the actual source and destination, as shown in figure

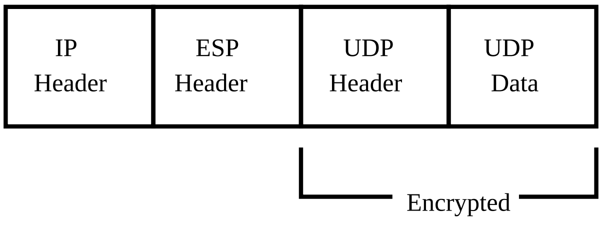

Figure 1: **IPsec Transport Mode**

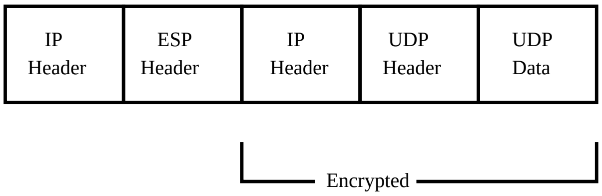

Figure 2: **IPsec Tunnel Mode**

2.

As was mentioned earlier, this mode is ideal for implementing VPNs.

The last, but not least, part of the picture is a key management infrastructure. IPsec can only work if the keys in the SAs are synchronized and updated in a secure fashion. To automate this task, different protocols have been devised that allow two peers to compute identical keys without actually sending all the data needed for it over the wire [7, 8]. The Internet Key Exchange, IKE, is one such, and Photuris is another. The main difference between these two lies in the complexity level. IKE is a very complex protocol which, however, offers considerable flexibility in negotiating and establishing SAs. IKE is the official IETF standard. Both protocols work in a similar vein, by first building an encrypted application-level "tunnel" where further key exchanges take place. The Diffie-Hellman algorithm [7] is used to make it computationally hard to crack the key computation. Every SA is assigned a lifetime, either in wall-clock time or in volume, and when such a lifetime expires, the key management daemon renegotiates with the peer, creating new SAs with fresh keys.

2.2.3 OpenBSD IPsec

OpenBSD's IPsec stack was written by John Ioannidis and Angelos Keromytis [18] and later enhancements and fixes have been provided by Niels Provos and Niklas Hallqvist. The core is stable and in production use securing data in many places all over the world, as it does not suffer from US export regulations. A number of companies, agencies, institutions, and individuals are using the code, a fact that has helped us significantly in finding and fixing bugs, and in motivating further development.

Recently, the API used to setup and maintain the SA database was switched to the standard PF_KEY [23]. This API is much more flexible than the old PF_ENCAP interface. Available algorithms for encryption are DES [26], 3DES, Cast-128, Blowfish [35], and Skipjack (support for the latter, despite its known weaknesses, was added after requests by US Government agencies using our IPsec stack). One-way hash algorithms are MD5, SHA1 and RIPEMD160 [20, 21, 17]. For key management, two daemons are available, *isakmpd* implementing IKE [29, 22, 12] and *photurisd* implementing Photuris [13].

2.2.4 Future Work in IPsec

Our IPsec implementation is under constant development and improvement, as there remain a number of unresolved issues.

- Our IPv6 stack is not yet integrated with our IPsec implementation.
- We want a more flexible, possibly unified policy mechanism. In particular, we are looking into merging routing, security policy, and protocol block lookups.
- Develop or borrow a policy API, rather than use private extensions to PF_KEY and PF_ROUTE.
- *isakmpd* has not yet covered all mandatory requirements in the RFCs.
- A DNSSEC [9] implementation, and integration in *isakmpd* and *photurisd*, will be needed for opportunistic encryption.
- *isakmpd* and *photurisd* are not linked with *libssl* so they will not automatically support RSA when an RSA-supporting *libssl* is installed.
- We do not currently do on-demand keying (a facility available in the past through the PF_ENCAP API).
- Finally, we intend to support some application API for requesting security and possibly other services. With that in place, we intend to have all networking applications take advantage of IPsec.

All of these are improvements that we want to address in the time-frame for the next release.

2.3 Kerberos

In a networked environment, it is very important to be able to authenticate users in a secure way over insecure networks. Kerberos is a network authentication protocol using a trusted third-party to provide authentication and basic session-key exchange.

Kerberos is built around a central key distribution center (KDC) which keeps a database of clients and servers (called principals) and their private keys. Encryption in Kerberos is based on DES [26]. When the client wants to use some service it issues a request to the KDC for a ticket for that service. The server returns a message encrypted with the client's private key, containing three parts: a session key that can be used for encryption between the client and the server, a timestamp, and a ticket. The ticket is encrypted with the private key of the server and contains the name of the client, a timestamp, the clients network address, lifetime of the ticket, and the same session key that the client obtained. The ticket can be passed to the server for authentication.

Kerberos [24] was originally developed by project Athena at MIT, but was not exportable from the US due to legal restrictions. The cryptographic functionality was removed and a "Bones" distribution was created and exported. The cryptographic interfaces were added back by Eric Young, and KTH (The Royal Institute of Technology in Stockholm, Sweden) maintained the code outside the USA. The Kerberos implementation in OpenBSD is "kth-krb", protocol version 4, and is used in a number of utilities.

2.3.1 Practical Uses

The simplest use of Kerberos is to authenticate users locally on a workstation. The *login, xdm,* and *su* programs in OpenBSD have the necessary code to allow Kerberos authentication. The next step is to provide authentication for network protocols. The *rlogin, rsh,* and *telnet* programs have been modified to use Kerberos. In addition to that, they can use the session key, obtained in the authentication phase, to encrypt the data-stream for privacy. Another very practical use is in "kx" - a protocol to authenticate and forward X11 connections in a secure way. Other programs using Kerberos for authentication include *cvs, sudo,* and *xlock.* Kerberos authentication is also used in *AFS.*

One of our future goals is to allow kerberized applications to use IPsec services when possible, thus avoiding double-encryption (and consequently degraded performance). Furthermore, we intend to integrate the Kerberos 5 clone being developed at KTH as soon as it is stable, especially since Kerberos IV only supports DES [26] encryption.

2.4 S/Key

S/Key [11, 10] is a one-time password system used for authentication. It provides protection against replay attacks where a third party captured a password, *e.g.,* by means of network sniffing, and tries to reuse it in a new authentication session.

S/Key uses a user supplied secret pass-phrase which is processed by a one-way function to generate a sequence of one-time passwords. In OpenBSD the one-way function can be chosen from a variety of computationally non-invertible hash functions like MD5 [34] or SHA1 [28], available in *libc.* S/Key is still useful when other cryptographic protocols are not available, or their implementations are not fully trusted, *e.g.,* when using a conference terminal room to login to a home machine.

3 Pseudo Random Number Generators

A Pseudo Random Number Generator (PRNG) provides applications with a stream of numbers which have certain important properties for system security:

- It should be impossible for an outsider to predict the output of the random number generator even with knowledge of previous output.
- The generated numbers should not have repeating patterns which means the PRNG should have a very long cycle length.
- A PRNG is normally just an algorithm where the same initial starting values will yield the same sequence of outputs.

Some applications have criteria which affect the type of PRNG which is needed. For instance, later on we will discuss IP datagram IDs and DNS [30] query-IDs, both of these issues have qualities which make it extremely desirable to have a PRNG which makes efforts to avoid emitting repetitions (thus ruling out use of a true-random source).

Many other operating systems also have random number device drivers and other related mechanisms, but largely make no use of them. Some such systems even provide such support only as optional device drivers, therefore discouraging use (*i.e.,* reliance). OpenBSD deviates by actually using these mechanisms in numerous ways. A few major interfaces or techniques are used:

- `/dev/?random` and similar kernel interfaces
- *arc4random(3)* in libc
- non-repeating PRNG

Each of these, and their uses in OpenBSD, will be covered in the following sections.

3.1 Kernel Randomness Pool

Computers are (generally) deterministic devices making it very hard to produce real random numbers. The PRNGs we use in OpenBSD do not generate random numbers themselves. Rather, they expand the randomness they are given as input. Fortunately, a multi-user operating system has many external events from which it can derive some randomness. In OpenBSD the kernel collects measurements from various devices such as the inter-keypress timing from terminals, the arrival time of network packets, and the finishing time of disk requests. The randomness from these sources is mixed into the kernel's *entropy pool.* When a userland program requests random data from the kernel, an MD5 hash is calculated over the whole entropy pool, "folded" in half by XOR-ing the upper and lower word of the MD5 output, and returned. The user can choose the quality of the generated random numbers by reading output from the different `/dev/?random` devices.

3.2 arc4random(3)

The *arc4random(3)* interface, available in the OpenBSD `libc`, makes use of the kernel randomness pool, described in the previous section, for seeding the keystream generator employed by the *ARC4* cipher (a cipher equivalent to RSADSI's RC4). The interface provides support for applications to "add" randomness to the pool maintained by *arc4random(3).* This interface is intended as a drop-in replacement for the traditional Unix *random(3)* interface, for those applications that need higher-quality random numbers.

3.3 Non-repeating Random Numbers

In OpenBSD, we designed a non-repeating pseudo-random number generator that was very fast and did not require additional resources.

For 16-bit non-repeating numbers, we used a prime $2^{14} < p < 2^{15}$ and g a randomly chosen generator for $\mathbb{Z}_p$. Being a generator, g has the property that any value $0 < x < p$ can be generated as $x = g^y \pmod{p}$, for some value y.

We then pick random a, b and m with $2^{14} < m < 2^{15}$ so that

$$f(n) \equiv a \cdot f(n-1) + b \pmod{m}$$

becomes a linear congruential generator (LCG).

We then determine the actual ID as

$$ID(n) = w \oplus (g^{f(n)} \mod p),$$

where w is a random seed. After the linear congruential generator has been exhausted, the most significant bit in $ID(n)$ is toggled and all parameters g, a, b, m, and w from above are chosen anew. Because the linear congruential generator does not repeat itself and a new number space is chosen after reinitialization, the generated IDs do not repeat themselves. The PRNG is typically seeded with material from the kernel randomness pool.

3.3.1 Randomness Used Inside the Kernel

- Dynamic sin_port allocation in bind(2).

  When an AF_INET socket is bound to a specific port number using the *bind(2)* system call, the process can choose the specific port, or elect that the system choose. Normal UNIX behavior resulted in the system allocating port numbers starting at 1024 and incrementing. Our new code chooses a random port, in the range 1024 to 49151.

  A similar issue existed with reserved port creation, using the *bindresvport(3)* and *rresvport(3)* library routines, which are supposed to pick a free port in the reserved range (typically between 600 and 1023). The old behavior was to allocate decreasing port numbers starting at 1023. The old code for these library routines effected this downward search using successive calls to *bind(2)*; we have replaced this with code using a newer kernel interface which is much more efficient and chooses a random port number within the reserved range.

  There are a number of poorly designed protocols (*e.g.,* rsh, ftp) which are affected by predictable port allocation; we believe that our approach is making it harder for attackers to gain an edge.

- Process PIDs.

```
char buf[20];

sprintf(buf, "/tmp/foo-%d", getpid());
(void) mkdir(buf, 0600);
```

Figure 3: **The Wrong Way To Generate A "Random" Directory**

```
struct timeval now;

gettimeofday(&now);
call_msg.rm_xid = getpid() ^ now.tv_sec ^
                  now.tv_usec;
```

Figure 4: **Typical RPC Initialization Code**

Programmers often use this value as if it is random, possibly because of the compellingly attractive argument that "pid numbers are effectively random on a busy enough system." Code like "srandom(getpid())" is quite common, as is code similar to that shown in figure 3.

In a normal system the attacker will have a very easy time predicting the PID and thus the obvious race attack is trivial. The race is as follows: the attacker creates the directory first, choosing the mode and ownership; subsequently it is possible to look at and replace files in the directory.

In OpenBSD, we use randomized PIDs, with a couple of obvious exceptions, *e.g., init(8)*.

- RPC transaction IDs (XID).

  Sun Microsystems Remote Procedure Call (RPC) messages contain a Transaction Identifier (XID) which matches a sent query against its received reply. In most RPC systems, the XID of the first message a process transmits will be initialized using the code shown in figure 4.

  Subsequently, the XID for each packet is simply incremented from this. Previously we mentioned that a local user might be able to guess what kind of range the next PID on the system might fall into; here we see that an outside attacker might also be able to determine this information. Our new code uses arc4random() to initialize the XID, and also avoids using two identical numbers consecutively.

- NFS RPC transaction IDs (XID).

  The NFS protocol uses RPC packets for communication. The RPC XID issue also applied to the NFS code we encountered, and we now use the same mechanism for NFS XIDs.

- Inode generation numbers.

  The *fsirand(8)* program makes use of *arc4random(3)* to generate random `inode` numbers for filesystem objects (files, directories, *etc.*). This increases the security of NFS-exported filesystems by making it difficult for an attacker to guess filehandles (which are partially derived from inode numbers).

- IP datagram IDs.

  Each IP packet contains a 16-bit identifier which is used, if the packet has been fragmented, for correctly performing reassembly at the final destination. Previously, this identifier simply incremented every time a new packet was sent out. By looking at the identifier in a sequence of packets, an outsider can determine how busy the target machine is. Another issue was avoiding disclosure of information when using IPsec in tunneling mode, as per section 2.2.2. A naive implementation might create a new IP header with an ID one more than the ID in the existing IP header. This could lead to known-plaintext attacks [4] against IPsec.

  To avoid these problems, we use the non-repeating PRNG described in section 3.3.

- Randomness added to the TCP ISS value for protection against spoofing attacks.

  Inside the kernel, a 32-bit variable called *tcp_iss* declares the Initial Send Sequence Number (ISS) to use on the next TCP [32] session. The predictability of TCP ISS values has been known to be a security problem for many years [25]. Typical systems added either 32K, 64K, or 128K to that value at various different times. Instead, our new algorithm adds a fixed amount plus a random amount, significantly decreasing the chances of an attacker guessing the value and thus being able to spoof connection contents.

- Random data-block padding for cryptographic transforms, as in RFC1827 IPsec ESP [2].

3.3.2 Randomness Used in Userland Libraries

- DNS query IDs typically start at 1 and increment for each subsequent query. An attacker can cause a DNS lookup, *e.g.,* by telneting to the target host, and spoof the reply, since the content of the query and the ID are known or easily predictable. Since host authentication is still in wide-spread use, this is a serious security vulnerability present in virtually all systems. To avoid this issue, we have modified our in-tree copy of *bind(8)* and our *libc* resolver to make use of the non-repeating PRNG.

- *arc4random(3)* seeding, as mentioned in section 3.2.
- Stronger temporary names.

  Processes typically create temporary files by generating a random filename via *mktemp(3)* and then opening that file in the /tmp directory. A more secure way for doing so is through *mkstemp(3),* which generates the filename and opens the file in one atomic operation, thus eliminating the potential for races. Both functions, which reside in `libc`, make use of *arc4random(3)* to generate the random filenames, making it much harder for an attacker to guess the names in advance.
- Generate salts for the various password algorithms. For some more details, see section 4.1.

3.3.3 Randomness Used in Userland Programs

- For generating fake S/Key challenges.

  One problem with most versions of RFC1938-based one time password (OTP) systems is that it is often possible to use them to determine whether or not a user has an account on a machine. The most trivial example of this is systems that provide a different prompt if the user has an entry in the OTP database. However, even for systems that always provide an OTP prompt, the prompt itself is rarely convincing and can be trivially identified as a fake. To address this problem, the OTP code in OpenBSD generates a consistent, credible challenge for non-existent users and users without an entry in the OTP database. It does so by generating the prompt based on the hostname and a hash of the username and the contents of a file generated from the kernel random pool. This file is usually created at install time and provides a constant source of random data. Thus, all three components of the challenge are constant, but only the hostname and username are known to the attacker.
- *isakmpd* and *photurisd* use the kernel randomness pool for generating IKE "exchange identifiers" (*i.e.,* protocol cookies and message IDs), random Diffie-Hellman [7] values, and random nonces.
- Certain games make use of the *arc4random(3)* interface for higher quality random numbers.

4 Secure Storage

One of the areas of least development in OpenBSD has been that of secure storage. While a number of utilities (*e.g., vi(1), ed(1), bdes(1), etc.*) directly support encryption services, our goal is to provide this service as transparently as possible to users. Ideally, we would like a layer either over or under the current native filesystem that would provide safe storage services.

As an interim solution, CFS [5] is included in the OpenBSD ports system and can be readily used. However, it does not provide the level of transparency we would like, and its performance is well below what we consider acceptable for general use. Clearly, more work is needed in this area.

Another issue related to secure storage is that of secure logging. Logs (and especially security-related logs) are extremely important in determining whether a system is under attack or has been compromised. The current logging facility, *syslog*, does not provide any facilities for detecting log-tampering, other than the option to send log messages to another host's *syslogd.* We are currently porting the *ssyslog* package [37] and are hoping to seamlessly replace the currently-used *syslogd.*

The remainder of this section briefly covers our *bcrypt*, approach to protecting user passwords, developed inside OpenBSD.

4.1 Bcrypt

Increasing computational power makes the use of cryptography to further system security more feasible and allows for more tuneable security parameters such as public key length. However, one security parameter - the length and entropy of user-chosen passwords - does not scale at all with computing power. Many systems still require user-chosen secret passwords which are hashed to keep them secret. When the UNIX password hash *crypt(3)* was introduced in 1976, it could not hash more than four passwords per second. With increasingly more powerful attackers it is common to compute more than 200,000 password hashes per second. In OpenBSD we use the *bcrypt* algorithm to make the cost of password hashing parameterizable. Its design makes it hard to optimize *bcrypt*'s execution speed or use commodity hardware instead of software. *bcrypt* uses a 128-bit salt and encrypts a 192-bit magic value. It takes advantage of the fact that the Blowfish algorithm (used in the core of *bcrypt* for password hashing) needs a fairly expensive key setup, thus considerably slowing down dictionary-based attacks. *bcrypt* uses the *arc4random(3)* interface for password salt-generation. A comparison between this approach and the mechanism used in certain other Unix systems for generating salts has shown that while *arc4random(3)* behaved extremely close to the statistical theoretical expectations; in contrast, other systems produced large numbers of collisions, making dictionary attacks faster.

A special configuration file, *passwd.conf(5)*, is used to determine which type of password scheme is used for a given user or group. It is possible to use different password schemes for local or YP passwords. For *bcrypt,* the number of rounds is also included. This facilitates adapting the password verification time to increasing processor speed. Currently, the default number of rounds for a normal user is 2^6, and 2^8 for "root." *bcrypt* is used in OpenBSD as the default password scheme since version 2.1. For more details, see [33].

5 Conclusion

In this paper, we gave an overview of the cryptography used in OpenBSD. We presented the supported network security mechanisms, with particular emphasis on IP security. We then discussed the various uses of randomness throughout the system. Finally, we briefly covered our plans for future work in the area of secure storage.

A lot of work remains to be done. In the short term, we need to complete the remaining parts of those mechanisms still under development, keeping in mind of course that security (and standards) is a moving target, and constant maintenance and updating will be needed. Beyond that, integration with existing and new utilities is a major item in our agenda. Finally, we are considering new mechanisms that address different problems, *e.g.,* untrusted-code containment.

It is important to note that all the mechanisms described in this paper are currently in use, solving real problems. We hope that this paper will encourage others to add these or similar mechanisms in their systems.

6 Acknowledgments

We would like to thank Hugh Graham, Todd Miller, and Chris Turan who provided comments (and sometimes text) in earlier versions of this paper. We would also like to thank all the OpenBSD developers for the work they contribute to the project, and our users for their continuing support.

7 Availability

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

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

8 Disclaimer

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] R. Atkinson. Security Architecture for the Internet Protocol. RFC 1825, August 1995.

[4] S. Bellovin. Probable Plaintext Cryptanalysis of the IP Security Protocols. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 155–160, February 1997.

[5] M. Blaze. A Cryptographic File System for Unix. In *Proc. of the 1st ACM Conference on Computer and Communications Security*, November 1993.

[6] T. Dierks and C. Allen. The TLS protocol version 1.0. Request for Comments (Proposed Standard) 2246, Internet Engineering Task Force, January 1999.

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

[8] W. Diffie, P.C. van Oorschot, and M.J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[9] D. Eastlake, 3rd, and C. Kaufman. Domain name system security extensions. Request for Comments (Proposed Standard) 2065, Internet Engineering Task Force, January 1997.

[10] N. Haller. The S/KEY one-time password system. Request for Comments (Informational) 1760, Internet Engineering Task Force, February 1995.

[11] Neil M. Haller. the s/key one-time password system. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, 1994.

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

[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. IP authentication header. Request for Comments (Proposed Standard) 2402, Internet Engineering Task Force, November 1998.

[15] S. Kent and R. Atkinson. IP encapsulating security payload (ESP). Request for Comments (Proposed Standard) 2406, Internet Engineering Task Force, November 1998.

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

[17] A. Keromytis and N. Provos. The use of HMAC-RIPEMD-160-96 within ESP and AH. Internet Draft, Internet Engineering Task Force, February 1999. Work in progress.

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

[19] RSA Laboratories. *PKCS #1: RSA Encryption Standard*, version 1.5 edition, 1993. November.

[20] C. Madson and R. Glenn. The use of HMAC-MD5-96 within ESP and AH. Request for Comments (Proposed Standard) 2403, Internet Engineering Task Force, November 1998.

[21] C. Madson and R. Glenn. The use of HMAC-SHA-1-96 within ESP and AH. Request for Comments (Proposed Standard) 2404, Internet Engineering Task Force, November 1998.

[22] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet security association and key management protocol (ISAKMP). Request for Comments (Proposed Standard) 2408, Internet Engineering Task Force, November 1998.

[23] 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.

[24] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, MIT, December 1987.

[25] R. T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. Computing Science Technical Report 117, AT&T Bell Laboratories, February 1985.

[26] Data Encryption Standard, January 1977.

[27] Digital Signature Standard, May 1994.

[28] Secure Hash Standard, April 1995. Also known as: 59 Fed Reg 35317 (1994).

[29] D. Piper. The internet IP security domain of interpretation for ISAKMP. Request for Comments (Proposed Standard) 2407, Internet Engineering Task Force, November 1998.

[30] J. Postel. Domain name system structure and delegation. Request for Comments (Informational) 1591, Internet Engineering Task Force, March 1994.

[31] Jon Postel. Internet Protocol. Internet RFC 791, 1981.

[32] Jon Postel. Transmission Control Protocol. Internet RFC 793, 1981.

[33] Niels Provos and David Mazières. A Future-Adaptable Password Scheme. In *Proceedings of the Annual USENIX Technical Conference*, 1999.

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

[35] Bruce Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, December 1993.

[36] William Simpson. IP in IP Tunneling. Internet RFC 1853, October 1995.

[37] Secure Syslog. `http://www.core-sdi.com/Core-SDI/english/slogging/ssyslog.html`.

[38] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH protocol architecture. Internet Draft, Internet Engineering Task Force, February 1999. Work in progress.