Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Strong Cryptography in the Linux Kernel

Discussion of the past, present, and future of strong cryptography in the Linux kernel

Jean-Luc Cooke
CertainKey Inc.
jlcooke@certainkey.com

David Bryson
Tsumego Foundation
david@tsumego.com
PGP: 0x74B61620

Abstract

In 2.5, strong cryptography has been incorporated into the kernel. This inclusion was a result of several motivating factors: remove duplicated code, harmonize IPv6/IPSec, and the usual crypto-paranoia. The authors will present the history of the Cryptographic API, its current state, what kernel facilities are currently using it, which ones should be using it, plus the new future applications including:

1. Hardware and assembly crypto drivers
2. Kernel module code-signing
3. Hardware random number generation
4. Filesystem encryption, including swap space.

1 History of Cryptography inside the kernel

The Cryptographic API came about from two somewhat independent projects: the international crypto patch last maintained by Herbert Valerio Riedel and the requirments for IPv6.

The international crypto patch (or 'kerneli') was written by Alexander Kjeldaas and intended for filesystem encryption, it has grown to also optionally replace duplicated code in the UNIX random character device (/dev/*random). This functionality could not be incorporated into the main line kernel at the time because kernel.org was hosted in a nation with repressive cryptography export regulations. These regulations have since been relaxed to permit open source cryptographic software to travel freely from kernel.org's locality.

The 2.5 kernel, at branch time, did not include any built in cryptography. But with the advent of IPv6 the killer feature of kernel space cryptography has shown itself. The IPv6 specification contains a packet encryption industry standard for virtual private network (VPN) technology. The 2.5 kernel was targeted to have a full IPv6 stack–this included packet encryption. The IPv6 and kernel maintainers in their infinite wisdom (!) saw an opportunity to remove duplicated code and encouraged the kerneli.org people to play with others.

And so, strong cryptography is now at the disposal of any 2.5+ kernel hacker.

2 Why bring cryptography into our precious kernel?

Cryptography, in one form or another, has existed in the main line kernel for many versions. The introduction of the random device driver

by Theodore Ts'o integrated two well known cryptographic (digest) algorithms, MD5 and SHA-1. Other forms of cryptography were introduced with the loopback driver (also written by Theodore Ts'o) these included an XOR and DES implementation for primitive filesystem encryption.

The introduction of cryptography for filesystem encryption, coupled with the kerneli patches, allowed users to hook the loopback device up to a cipher of their choosing. Thus providing a solution for secure hard disk storage on Linux.

With the advent of IPSec the introduction of crypto into the kernel makes setting up encrypted IP connections extremely easy. Previous implementatinons have used userspace hooks and required compilcated configuration to setup properly. With IPSec being inside the kernel much of those tasks can be automated.

More advanced features for cryptography in the kernel will be explained throughout this paper.

2.1 Example Code

The use of the API is quite simple and straightforward. The following lines of code show a basic use of the MD5 hash algorithm on a scatterlist.

```
#include <linux/crypto.h>

struct scatterlist sg[2];
char result[128];
struct crypto_tfm *tfm;

tfm = crypto_alloc_tfm("md5", 0);
if (tfm == NULL)
  fail();

/* copy data into */
/* the scatterlists */

crypto_digest_init(tfm);
crypto_digest_update(tfm, &sg, 2);
crypto_digest_final(tfm, result);

crypto_free_tfm(tfm);
```

Ciphers are implemented in a similar fashion but must set a key value (naturally) before doing any encryption or decryption operations.

```
#include <linux/crypto.h>

int len;
char key[8];
char result[64];
struct crypto_tfm *tfm;
struct scatterlist sg[2];

tfm = crypto_alloc_tfm("des", 0);
if (tfm == NULL)
  fail();

/* place key data into key[] */
crypto_cipher_setkey(tfm, key, 8);

/* copy data into scatterlists */

/* do in-place encryption */
crypto_cipher_encrypt(tfm,sg[0],
         sg[0],len);
crypto_free_tfm(tfm);
```

The encryption and decryption functions are capable of doing in-place operations as well as in/out (separate source and destination) operations. This example shows in-place opertaion. By changing the encrypt line to:

```
crypto_cipher_encrypt(tfm,
       sg[0], sg[1], len);
```

the code then becomes an in/out operation.

3 Kernel module code-signing

Signing of kernel modules has been a desired addition to the kernel for a long time. Many

people have attempted to do some kind of authenticated kernel module signing/encryption but usually by the means of an external user-mode program. With the movement of the module loader into the kernel in the 2.4 series a truly secure module loader is possible. The authors would like to propose a method for trusted module loading.

To create the secure module structure we need a way of designating a module as trusted. During compile time, a token can be created for each module. The token contains two identifiers.

- Time stamp token, denoting module creation time.
- A secure hash of the module in its compiled state.

After these three tokens are created they are encrypted by an internal private key (protected by a separate password of course) bound to the kernel. The encrypted file is then stored in a file on the local disk.

Loading of the module occurs as follows.

1. A request to load module `rot13` is made by the system.
2. The kernel reads the encrypted file for module `rot13`.
3. Using the kernels public key the file is decrypted, and the tokens are placed in memory
4. A hash is computed against the file on resident disk of module `rot13` and compared against the signed token.
5. If the hashes are equal the module is trusted and code loaded into memory.

This allows for a large degree of flexibility. Anybody on the system who has access to the kernels public key can verify the validity of the modules. Plus the kernel does not need to have the private key in memory to authenticate since the public key can do the decryption. Thus reducing the time that the private key is stored in resident memory unencrypted.

However this approach can only protect a system to a point. If a malicious user is on your system and is at the stage where they can load modules (root access) this will only slow them down. Nothing prevents them from compiling a new kernel with a ‘dummy’ module loader that skips this check (solutions to this problem welcome!).

This system requires that the kernel contain functionality to support arbitrarily large integers and perform asymmetric cryptography operations. Currently, there is preliminary code that supports this functionality, but has yet to be formally tested or introduced to the community.

4 Cryptographic Hardware and Assembly Code

A new exciting aspect of cryptography in the kernel is the ability to use hardware based cryptographic accelerators. Many vendors offer a variety of solutions for speeding up cryptographic routines for symmetric and asymmetric operations.

The chips provide cheap, efficient, and fast cryptographic processors. These can be purchased for as little as $80.00USD and offer a considerable speedup for the algorithms they support. The proposed method of integrating hardware and assembly is to have the chip or card register its capabilities with the API.

This way the API can serve as a front end to the

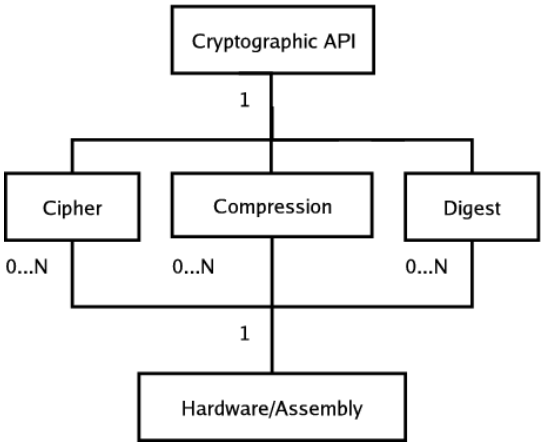

Figure 1: The proposed hardware interface model

hardware driver module. Instead of a the hardware registering "aes" it would register "aes-<chipset name>" with the API. Calls to the API can then specify which implementations of the ciphers that are desired depending on what performance is needed.

As of this writing (*May 2003*) there is also no way to query the API for the fastest method it has for computing a cipher. There is in the design stage an asynchronous system for dynamically receiving requests and distributing them to the various pieces of hardware (or software) present on the system. The OpenBSD API and cryptography sub-system is being used as a reference model.

This method would allow users of the API to send queries to a scheduler with a call similar to the current interface, but adding using the cypher name `aes_fastest` or `aes_hardware`. The scheduler then sends the requested command to a piece of hardware that is waiting for requests, and fulfills the requested hardware requirements.

Drivers that are currently finished and/or under development include:

- Hifn series processors 7751 and 7951.
- Motorola MPC190 and MPC184 series.
- Broadcom BCM582[0|1|3] series.

4.1 Hardware Random Number Generation

Most cryptographic hardware and lately some motherboards have been including a hardware random number generation chip. These are a wonderful source of generating entropy because they are both fast and produce very random data. A set of free-running oscillators usually generates the data. The oscillators frequency clocks drift relative to each other and to the internal clock of the chipset. Thus, producing randomly flipped bits in a specified 'RNG' register.

Random number generation in the kernel uses interrupts from the keyboard, mouse, hard disk, and network interfaces to create an entropy pool. This entropy pool produces the output of `/dev/random` and the less secure `/dev/urandom`.

The current interface is missing a way to add random data from an arbitrary hardware source. By using tying the random driver into the Cryptographic API the random driver can gain both extra sources of entropy and the acceleration from making its MD5 and SHA-1 functions available for hardware execution. The result would be a faster and better entropy pool for random data generation.

5 Filesystem encryption

By far, the cryptographic API has the largest user-base with filesystem encryption. Several distributions have shipped with support for filesystem loopback encryption for over a year. Let us take a moment to explore the details of

filesystem encryption. When a write or read request occurs in the kernel the information destined for a device passes through the VFS layer, then down through the device driver layer and onto the physical media.

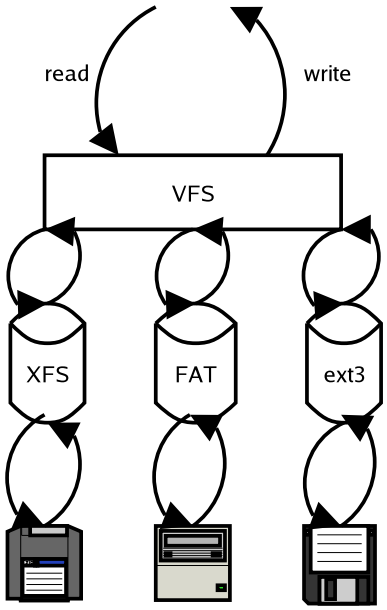

Figure 2: A diagram of the Linux VFS

Looking for a place to encrypt the data isn't easy, we could intercept the information at the VFS layer, but the result is encrypted data with plaintext metadata. Thus giving an attacker an edge, for example being able to track down your `/tmp/.SCO_source_code` directory and begin attacking the encrypted data there. The next place to intercept the plaintext data would be at the filesystem level. But writing per filesystem hooks to encrypt both filesystem data and filesystem metadata would be a nightmare to implement, not to mention a horrible design decision. So the only place left is somewhere outside of the filesystem code, but before the data is passed to the device driver for the media. Enter loopback drivers.

The loopback device driver in Linux allows us to send the data (plaintext) and metadata (plaintext) through a layer of memory copying before it is written to a device. Here is where the encryption will be done—this way all data written to the device can be encrypted instead of just the filesystem data.

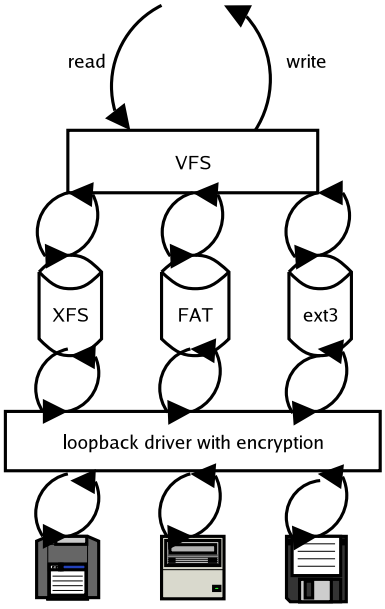

Figure 3: A diagram of the loopback encryption layer

By using the loopback driver an added level of flexibility is added. Users can have their home directories stored as large encrypted files on the primary drive. These would then be loaded via the cryptographic loopback driver upon login and unmount when the user exits all sessions.

5.1 Swap memory Encryption

Encryption of swap is a difficult problem to approach. Any system in which the filesystem data is encrypted has the chance of the data being moved out to swap memory when the OS gets low on RAM. This can easily be solved by 'locking' all memory into RAM and not allowing it to be swapped to a physical media with the `mlock()` function. However, this vastly reduces the usability of the system (Linux tends to kill processes when out of

memory). In the past, Linux has implemented encrypted swap with a loopback device running through the swap accesses. This approach works, but is slow and cumbersome to implement.

What is needed is a policy for encrypting **all** pages swapped to disk. The OpenBSD community has had a similar policy for a long time and feels that the performance loss (roughly 2.5 times longer to write a page to disk) is worth the added security.

The 2.4 series crypto (not a part of the mainline the kernel) named the "International Kernel Cryptography Patch" included a loopback encryption driver. The driver had limited features but did the job of encrypting data fairly well. In the 2.5 series driver there have been some performance improvements, like multithreaded (and SMP) reading from loopback devices, and code readability improvements.

6 Userspace Access

Access to the API is not currently possible from user space. Discussions over how to implement this have come up with a variety of proposals. The current direction is to have a device provide access to the API via ioctls.

Compatibility with the already mature OpenBSD API has been suggested. This would decrease application porting time to almost nothing.

7 Final Comments

Thus far the 2.5 Cryptographic API has been under constant development with the adding of new ciphers and functionality since its inclusion in the kernel. The API is young and has promising plans to expand, hopefully the authors of this paper have given you an adequate intro to the capabilites of the API.

References

[Bryson] David Bryson, *The Linux CryptoAPI: A Users Perspective*. 16 May 2002. `http://www.kerneli.org/howto/index.php`.

[Morris] Morris, James, and David S. Miller. *Scatterlist Cryptographic API*. 10 May. 2003. Linux Kernel Documentation linux/Documentation/crypto/api-intro.txt.

[Steve] Steve, `steve@trevithick.net`. *Re: [CryptoAPI-devel] Re: hardware crypto support for userspace ?* 11 Dec. 2002 via `http://www.kerneli.org/pipermail/cryptoapi-devel`.

[Hifn] Hifn, Inc. *7951 Data Sheet - Device Specification*. 2 June. 2001. `http://www.hifn.com/docs/a/DS-0028-02-7951.pdf`.

[Provos] Provos, Niels. *Encrypting Virtual Memory*. `http://www.openbsd.org/papers/swapencrypt.ps`.