Sistemas de arquivos criptografados para Debian, Gentoo e Red Hat

Top Secret!

Criptografar sistemas de arquivos inteiros é também uma maneira de proteger os dados em caso de roubo do disco rígido. Enquanto os usuários de SuSE Linux podem ativar a criptografia do sistema de arquivos durante a instalação, algum trabalho manual é necessário para usuários de outras distribuições.

POR CHRISTIAN NEY

Usuários de laptop têm os seus dados à mão aonde quer que estejam. O que por um lado é prático, torna-se um problema quando o portátil é roubado. Normalmente em tais casos, os arquivos contidos no sistema caem rapidamente em mãos erradas. Mesmo aqueles que somente levam o equipamento para reparos podem cair nas mãos de um técnico curioso, que terá tempo mais do que suficiente para fuçar nos arquivos particulares.

Existem técnicas de codificação para a proteção de dados confidenciais, e elas podem ser instaladas sem muito trabalho: A GNU Privacy Guard (GPG, [1]) codifica arquivos individuais sob demanda. Porém, para proteger vários arquivos, este procedimento é muito desconfortável. Em tais casos, é melhor codificar todo o sistema de arquivos de uma vez. A distribuição SuSE já mostra há anos (ver Figura 1) que isso pode ser feito de forma amigável. O sistema de arquivos criptografado pode ser criado durante a instalação, caso desejado. Em outras distribuições, isso pode ser feito com algum trabalho manual – e mesmo melhorado em alguns pontos. A distribuição Gentoo facilita muito a vida do usuário neste particular.

Casa trancada

Tomamos como exemplo para o nosso artigo a criptografia do sistema de arquivos instalado e montado separadamente em */home*. Ele deve ser montado durante a inicialização do sistema, independente de estar criptografado, e a chave de criptografia deve ficar em um chaveiro USB. Esta chave substitui a senha comum, servindo de base para a codificação do sistema de arquivos durante a inicialização do dispositivo *cryptoloop*. A vantagem deste método é que a chave consiste de valores aleatórios e é bem mais longa que a maioria das senhas. Assim, os chamados "ataques de dicionário" são excluídos automaticamente: um agressor teria que realmente tentar todas as possibilidades (força bruta) para descobrir qual é a chave de codificação. É quase impossível que ele obtenha sucesso com uma chave de tamanho razoável.

No Linux há diversas possibilidades de criptografia de sistemas de arquivos inteiros. A criptografia via dispositivo *loopback*, que se serve das funções criptográficas do kernel [2], oferece a maioria dos algoritmos disponíveis e, além disso, é muito simples de usar quando comparada com outras soluções.

Sanduíche

Os dispositivos de loop utilizam uma camada intermediária, que se situa entre o dispositivo de bloco (por exemplo */dev/hda1*) e o sistema de arquivos nela contido. Normalmente o sistema de arquivos acessa o dispositivo de bloco diretamente, como mostrado na Figura 2.

Dispositivos de loop comportam-se como dispositivos de bloco comuns. Entretanto, eles retransmitem os dados para um dispositivo normal ou para um arquivo (Figura 2). Qualquer um que já tenha montado uma imagem ISO com a opção *loop* conhece este procedimento.

Os dispositivos de loopback também são capazes de modificar os dados durante a operação. Com o dispositivo cryptoloop, o kernel intervém a cada escrita e leitura para codificar e decodificar os dados (Figura 2). O usuário não percebe nada – com exceção do pedido inicial de uma senha. Mesmo o pedido de senha pode ser evitado, se a chave for alojada num chaveiro USB. Infelizmente os patches necessários para criar sistemas de arquivos criptografados no Linux não estão contidos nos kernels de todas as distribuições Linux por padrão.

- Para o Debian Sarge é necessário recompilar o kernel juntamente com o pacote *kernel-patch-cryptoloop*.
- A partir de sua versão 8, o Red Hat (agora Fedora) já dispõe de suporte a sistemas de arquivos com cryptoloop. Contudo, somente o módulo AES está disponível por padrão.
- O kernel padrão do Gentoo, *gentoo-sources* já contém o patch necessário há muito tempo. Porém, é preciso que as opções necessárias sejam selecionadas antes da compilação (Figura 3).

Pode-se verificar facilmente se a distribuição fornece os patches necessários: produza um arquivo codificado e tente integrá-lo a um sistema de arquivos nor-

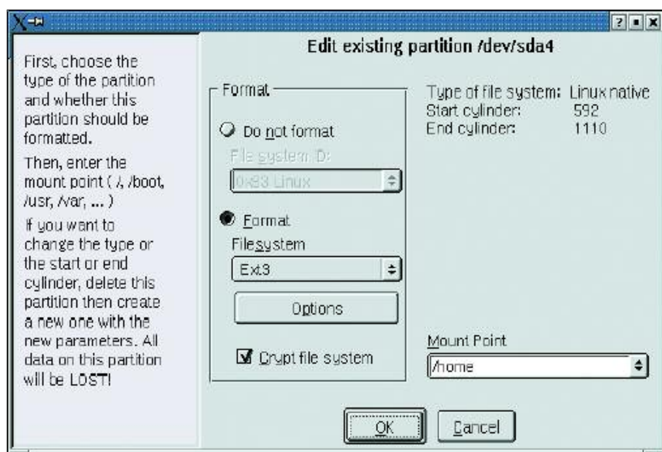


Figura1:No SuSE Linux, basta um clique do mouse na janela correta do YaST para ter o sistema de arquivos *home* codificado.

mal. É preciso antes fornecer, com a ajuda do comando *dd*, um arquivo que mais tarde irá conter o sistema de arquivos codificado. Um tamanho de 20 MBytes para este arquivo é adequado para o teste.

O dispositivo de loop

O comando *losetup* produz um dispositivo de loop a partir do arquivo criado. Ele pede para isso uma senha de codificação. Para fins de teste uma senha é suficiente; mais tarde criaremos uma chave aleatória. Para inserir a senha é preciso cautela: o *losetup* só pede a senha uma única vez. Quem errar a digitação perde o acesso aos dados codificados. Para finalizar, deve-se ainda criar um sistema de arquivos no dispositivo de loop, montar o sistema de arquivos e colocar nele alguns arquivos, conforme mostrado abaixo:

```
dd if=/dev/zero of=/root/cryptfile ➘
bs=1024k count=20
losetup -e aes -k 128 /dev/loop0 ➘
/root/cryptfile
mkfs -t ext2 -m 0 /dev/loop0
mount -t ext2 /dev/loop0 /mnt
cp /bin/{mv,rm} /mnt
```

Se o *losetup* retornar uma mensagem de erro logo de início, provavelmente o kernel ainda não está em condições de lidar com sistemas de arquivos criptografados. A Crypto-API está faltando. A referência [3] esclarece como adicioná-la ao kernel. Não é necessário recompilar todo o kernel – a CryptoAPI é composta apenas de módulos que o kernel carrega durante a execução.

Também é necessário adicionar suporte a chaveiros USB no kernel. As opções apropriadas para isso são *USB-Mass-Storage*, *SCSI-Disk-Support* e *SCSI emulation support*. A maioria dos kernels que vêm com as distribuições atuais, normalmente já contém os módulos apropriados.

Ferramentas do usuário

Além do suporte à codificação e decodificação no kernel, são necessários programas no espaço do usuário para criptografar os sistemas de arquivos e depois montá-los. Para isso o *losetup* e o *mount* (presente no pacote *util-linux*) precisam ser instalados e configurados.

- O pacote *util-Linux* da Debian Sarge já é preparado para as condições de sistemas de arquivos codificados, não sendo necessário um patch.
- Até a versão 9 do Red Hat, o pacote *util-linux* não continha os patches necessários. Neste caso também é necessário trabalho manual. As etapas são muito bem explicadas em [4].
- O Gentoo acrescenta ao pacote *util-linux* as funções para tratar dos sistemas de arquivos criptografados com a opção (“use flag”) *+crypt*.

Antes de colocar as mãos na massa, chegou a hora de fazer aquele backup que você estava planejando faz tempo. Caso contrário, uma vez que nossas instruções a seguir levarão à exclusão de todo o sistema de arquivos */home* original, os dados contidos nele seriam irrevogavelmente perdidos.

Um sistema de arquivos codificado só representa um ganho genuíno de segurança quanto não restar no disco nenhum resquício da velha partição. Isso significa destruir totalmente os dados do disco: exclusão ou formatação não são suficientes. Certifique-se de que os arquivos originais estão no backup, eles serão, mais tarde, recolocados no novo sistema de arquivos criptografado.

Para inutilizar o antigo sistema de arquivos, é mais simples enchê-lo com valores aleatórios: o administrador desmonta o sistema de arquivos */home* e usa o gerador de números aleatórios do kernel para, literalmente, entupir a partição com “lixo”. No exemplo a seguir, a partição */home* fica em */dev/hda6*. Quem dá valor à melhor eficiência na geração de valores aleatórios e dispõe de bastante tempo, deve dar preferência ao */dev/random*, ao invés de usar o */dev/urandom* (ver Quadro 1: “Eficiência demanda tempo”).

```
dd if=/dev/urandom of=/dev/hda6 ➘
bs=4k conv=notrunc
```

Este comando sobrescreve todo o dispositivo de bloco com valores aleatórios. Desta forma, fica mais difícil depois diferenciar “lixo” e dados codificados. A sobrescrita leva mais ou menos tempo, dependendo da quantidade de entropia existente no gerador de números aleatórios e do tamanho do sistema de arquivos. Por incrível que pareça, mesmo todos estes procedimentos ainda não são suficientes para impedir restauradores profissionais de dados de recuperar os dados antigos. Ainda há magnetismo remanescente suficiente no disco rígido [5]. Quem for realmente paranóico deve repetir este procedimento várias vezes.

Quadro 1: Eficiência demanda tempo.

Ambos os dispositivos */dev/random* e */dev/urandom* são interfaces para o gerador de números aleatórios do kernel. Eles coletam eventos os mais aleatórios possíveis e os reúnem em uma “pilha de entropia”. Cada acesso ao */dev/random* reduz a quantidade de entropia existente na pilha.

Caso não haja mais entropia suficiente, o kernel bloqueia o processo de leitura até que colete novamente a quantidade necessária de valores randômicos. Isso confere alta qualidade ao gerador de valores aleatórios. Muitas funções criptográficas (por exemplo: OpenSSL, GPG e outros) usam este gerador.

O dispositivo */dev/urandom*, entretanto, retorna todo o volume de dados requisitado, sem examinar se há entropia suficiente. Com isso pode acontecer que agressores consigam tirar conclusões sobre os dados aleatórios retornados. Contudo, tais valores são suficientes para tarefas simples.

A chave do sistema de arquivos

Como próximo passo, vamos gerar o arquivo com a chave. Esta chave serve para codificar e decodificar os dados do sistema de arquivos criptografado. Para evitar que a chave possa ser quebrada por um ataque de dicionário, ela deve ser a mais longa e aleatória possível. Somente assim haverá realmente uma vantagem em comparação com uma senha comum, geralmente mais curta. Já que o arquivo com chave de codificação deve estar no chaveiro USB, ele deve estar montado no sistema, o que faremos a seguir em */mnt/usb/*:

```
head -c 2880 /dev/random | ↩
uuencode -m -| head -n 65 | tail ↩
-n 64 | gpg --symmetric -a > ↩
/mnt/usb/key.gpg
```

A linha de comando acima lê somente 2880 bytes do */dev/random* e utiliza então *uuencode* para codificar os valores aleatórios em Base64. Os comandos *head* e *tail* eliminam as saídas de texto puro do *uuencode*.

O resultado já tem características aleatórias muito boas, e a codificação final com GPG melhora ainda mais a qualidade. Para isso o GPG requer uma frase-senha, que serve de chave simétrica. Apenas com ela é possível decodificar os dados – o que significa dizer que, se este arquivo-chave for perdido, os dados ficarão inacessíveis. Por esta razão, é extremamente importante que uma cópia dele seja mantida em lugar seguro.

A codificação adicional com GPG também serve para proteger a chave contra usuários não autorizados: ao invés de utilizar a versão codificada, pode-se decodificar novamente o arquivo-chave com auxílio da frase-senha do GPG e usar os dados resultantes como chave. Deste modo, um agressor precisaria não só estar em poder do chaveiro USB, como também conhecer a frase-senha do GPG para conseguir acesso aos dados.

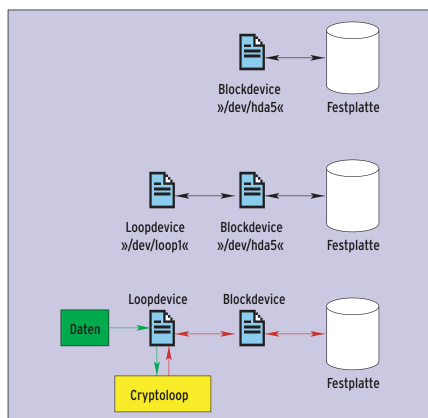


Figura 2: O kernel dá acesso a um dispositivo de bloco no disco rígido (acima). O dispositivo de loop (centro) entra antes do dispositivo de bloco. Com a criptografia via loopback (abaixo) somente dados criptografados vão parar no disco rígido.

Os preparativos agora acabaram: o próximo passo é configurar o dispositivo de loop. Ao mesmo tempo, são oportunas algumas reflexões sobre o algoritmo de codificação a ser utilizado e comprimento de chave. O exemplo a seguir usa o Twofish [6]. Este algoritmo é empregado há alguns anos em muitos produtos e foi avaliado por muitos especialistas em criptografia. Ele é considerado bastante maduro (veja o Quadro 2).

```
losetup -e twofish -k 128 ↩
/dev/loop0 /dev/hda6 -p 0 < ↩
/mnt/usb/key.gpg
```

Quadro 2: "Decifra-me, ou te devoro!"

Um bom ponto de referência para a seleção de um algoritmo apropriado é oferecido pelo comparativo [8] da AES (Advanced Encryption Standard), escrito, entre outros, pelo papa da criptografia Bruce Schneier. Ele mostra que o Rijndael e o Twofish são os algoritmos mais rápidos e também os mais freqüentemente utilizados. No entanto, o Rijndael perde em velocidade na medida em que o comprimento da chave aumenta.

O Twofish oferece um pouco mais de segurança [9]. Apesar de o algoritmo de Rijndael ser um pouco mais recente, já existem ataques teóricos contra este algoritmo. De qualquer maneira, o nível de seguraça oferecido por Rijndael, considerado o melhor segundo a AES, também deve ser totalmente suficiente.

Outro ponto é a escolha de um comprimento adequado para a chave. Pode-se escolher 128, 192 ou 256 bits. Quanto maior o comprimento, mais seguro é o algoritmo contra ataques de força bruta; por outro lado, a velocidade diminui. Se houver deficiências na codificação usada para gerar a chave, ou seja, se pudermos de algum modo inferir os dados contidos nela, de pouco adianta o seu maior comprimento [10]. Por esta razão 128 ou 192 bits são considerados suficientes.

Figura 3: Os kernels padrão (sem patches suplementares) já contém opções de criptografia. Para ativá-las basta escolher *Cryptographic API* e os algoritmos desejados sob o menu *Cryptographic Options*, usando uma das muitas interfaces de configuração kernel.

Listagem 1: Script de inicialização

```
01 #!/bin/sh -e
02
03 case "$1" in
04   start)
05     /sbin/modprobe sd_mod
06     /sbin/modprobe usb-storage
07     /sbin/modprobe cryptoapi
08     /sbin/modprobe cryptoloop
09     /sbin/modprobe twofish
10     /bin/mount -t vfat ➘
   /dev/sda1 /mnt/usb
11     /sbin/losetup -e twofish ➘
   -k 128 /dev/loop0 /home -p 0 ➘
   < /mnt/usb/key.gpg
12     /bin/mount /dev/loop0 /home
13     /bin/umount /dev/sda1
14     ;;
15   stop)
16     /sbin/losetup -d /dev/loop0
17     /bin/umount /dev/loop0
18     ;;
19 esac
```

Este comando inicializa o dispositivo de loop */dev/loop0* com o algoritmo Twofish. O comprimento da chave é de 128 bits. O dispositivo de loop armazena os dados codificados no dispositivo de bloco */dev/hda6*. O programa espera por uma senha como base para a chave, que, neste caso, não vem pelo teclado, e sim do arquivo key.gpg no "chaveiro", redirecionado através da entrada padrão com o comando: *-p 0 < /mnt/usb/key.gpg*.

Aquecendo os motores...

Agora o dispositivo de loop está pronto para ser usado com qualquer sistema de arquivos. Sistemas de arquivos com journaling podem, contudo, em casos desfavoráveis, causar problemas: eventualmente o dispositivo de loop escreve no disco numa seqüência diferente da esperada pelo sistema de arquivos. O journal torna-se, então, totalmente inútil e o que pode levar a insconsistências graves no sistema de arquivos no caso de crashes.

```
mkfs -t ext2 /dev/loop0 mount ➘
/dev/loop0 /mnt
```

No nosso exemplo utilizamos como sistema de arquivos o Ext 2. Após a montagem, é aconselhável criar alguns arquivos para testar o novo sistema de arquivos. Depois, desmontar o sistema de arquivos novamente e desativar o dispositivo de loop:

```
umount /mnt
losetup -d /dev/loop0
```

Para evitar surpresas desagradáveis mais tarde, deve-se repetir diversas vezes a criação do dispositivo de loop e a montagem do sistema de arquivos. Não havendo ainda nenhum dado importante no sistema de arquivos criptografado, em caso de dúvida, pode-se começar tudo do início novamente.

Aos seu lugares...

Ainda são necessárias algumas mudanças no sistema, uma vez que todo o procedimento, inclusive a montagem do chaveiro USB, deve ocorrer automaticamente. Algumas distribuições já estão preparadas para isso. A Gentoo, por exemplo, fornece o script de inicialização *cryptoloop*. Os comentários contidos neste script esclarecem eventuais dúvidas sobre seu uso. As soluções genéricas a seguir não são tão elegantes, mas devem funcionar na maioria das distribuições comuns. A variante mais simples usa um script de inicialização já existente; geralmente */etc/init.d/rc.local* é uma boa escolha. Mais elegante ainda seria criar um script de inicialização próprio, obviamente observando a seqüência de chamada de tais scripts pelo sistema durante o boot.

Um exemplo de *rc.local* para as versões de kernel a partir do 2.4.22 pode ser vista na Listagem 1. Os nomes dos módulos de algoritmo foram modificados a partir do kernel 2.4.22. Para as versões mais antigas os nomes devem começar com um *cipher-*. A linha 9, para um kernel mais antigo, ficaria assim: */sbin/modprobe ciper-twofish*.

A montagem do chaveiro USB seria ainda mais confortável com a ajuda do montador automático de volumes *supermount* [7]. Para isso é necessário um outro módulo de kernel, que pode ser facilmente instalado. Adicionalmente, é necessária uma entrada especial no arquivo */etc/fstab*:

```
/mnt/usb/mnt/usb  supermount ➘
sync,rw,fs=auto,dev=/dev/sda1 0 0
```

Com isso, podemos economizar as montagens explícitas no script de inicialização. O *supermount* monta o chaveiro USB sob demanda. Pode-se utilizar a mesma técnica para montar automaticamente os CD-ROMs, bastando para isso alterar o dispositivo e o ponto de montagem indicados acima, com */dev/cdrom* e */mnt/cdrom*, respectivamente.

Segurança total!

Todos os nossos esforços terão sido em vão caso dados confidenciais ainda puderem ser encontrados sem codificação em outro lugar do disco. O pior caso é aquele em que o kernel move exatamente a zona da memória em que a chave se encontra para a partição *swap*. Outro risco são os famosos "core dumps", que às vezes escrevem a zona memória de um processo após um erro grave no disco rígido.

Ambos os problemas são fáceis de resolver. Na área de *swap* (memória virtual) não há nenhum dado que, em condições normais, ainda seja necessário após uma reinicialização do sistema. Por isso é perfeitamente possível criptografar esta área de modo transparente, e melhor ainda, com uma chave aleatória que ninguém conheça.

A desvantagem deste método é um atraso mínimo adicional ao durante o "boot": uma vez que o sistema codifica a partição *swap* após cada inicialização do sistema com uma outra chave, ele deve reinicializar esta partição a cada vez.

Para isso é necessário modificar o script de inicialização responsável pela montagem da partição *swap*. Com o comando *grep swapon /etc/init.d/** identifica-se qual o script a ser modificado. Neste script, antes da linha que contém o comando *swapon*, devem ser inseridas duas linhas suplementares:

```
/bin/dd if=/dev/urandom bs=1 ↩
count=16 | /sbin/losetup -e ↩
twofish -k 128 -p 0 /dev/loop1 ↩
/dev/hda2
/sbin/mkswap /dev/loop1
/sbin/swapon -a
```

Estes comandos produzem um outro dispositivo de loop. Sua senha se compõe de 16 bits de valor aleatório. Como esta chave muda a cada nova inicialização, a partição *swap* é reinicializada a cada novo boot através do comando *mkswap*.

Além disso, deve-se substituir no arquivo */etc/fstab* a linha que corresponde ao dispositivo de bloco da partição *swap* (em nosso exemplo, */dev/hda2*) pelo dispositivo de loop correspondente (aqui */dev/loop1*). Os mais paranóicos devem desativar momentaneamente a partição *swap* com *swapoff -a*, sobrescrevê-la com o comando *dd* e o dispositivo */dev/urandom*, e executar manualmente os três passos descritos acima.

Adeus, "core dump"

Em quase todas as distribuições, a geração de core dumps é desativada por padrão. Tal comportamento pode ser verificado através do comando *ulimit -a | grep core*. Ele determina o número máximo de blocos que um arquivo "core" pode ocupar. Se o valor retornado pelo comando indicado for *0*, o sistema não produz core dumps. Se o valor for maior que *0*, é possível desativá-lo globalmente no arquivo */etc/security/limits.conf*. A entrada conveniente para isso é:

```
*        soft     core     0
```

Na maioria dos casos, esta linha já está até presente no arquivo, mas comentada, bastando apenas remover o caractere *#* para ativá-la.

E o Kernel 2.6?

Quem quiser usar partições criptografadas sob o kernel 2.6, vai seguir passos semelhantes aos descritos aqui, mas deve fazê-lo apenas quando estiver usando exclusivamente o novo kernel, pois os dispositivos *cryptoloop* criados sob o kernel 2.4 infelizmente não podem ser montados pelo kernel 2.6.

Até mesmo uma atualização de kernel dentro da série 2.4 pode resultar no fato do novo kernel não mais decodificar os dados criptografados. As situações mais críticas ocorre quando com atualizações para versões do kernel a partir da 2.4.22. Depois desta versão o kernel oficial também contém *crypto code* e os patches descritos neste artigo não são mais necessários. Infelizmente as mudanças não se restringem somente aos nomes de cada módulo (como por exemplo *aes* ao invés de *cipher-aes*), mas são tão profundas que o antigo sistema de arquivos criptografado não pode mais ser lido pela nova versão. Felizmente, a atualização do kernel 2.4.22 para outros da série 2.4 não traz mais nenhum problema no que tange a dispositivos criptografados.

Devido à flexibilidade do procedimento *cryptoloop*, há espaço para diversas variações e melhorias, que tornem o sistema ainda mais personalizável ou seguro. Quem, por exemplo, precisa de um sistema de arquivos FAT no chaveiro USB para armazenar outros dados que possam ser lidos no Windows, pode adicionalmente criar nele um pequeno sistema de arquivos Ext 2 para guardar o arquivo-chave, de forma que ele fique protegido de modificações inadvertidas.

Na medida

Apesar das operações adicionais de codificação e decodificação, é difícil perceber perdas de desempenho nos procedimentos de leitura e escrita em laptops comuns, exceto pelo uso ligeiramente maior de CPU, quando comparamos com a velocidade de tais procedimentos em sistema de arquivos não codificado. ■

Uma alternativa: Loop-AES

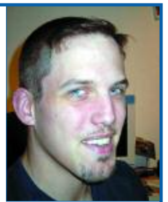
O Loop-AES [11] foi desenvolvido especialmente para a codificação de sistemas de arquivos. Ele também é capaz de codificar a partição *swap* sem a necessidade de adaptações especiais. O Loop-AES usa o algoritmo AES, também conhecido como Rijndael, mas também pode usar outros algoritmos como o Blowfish, o Twofish ou o Serpent, se desejado.

Como no caso da Crypto-API, com o Loop-AES também não é necessário compilar outro kernel; as funções necessárias podem ser incorporadas ao sistema na forma de módulos. Na maioria das distribuições, entretanto, o pacote *util-linux* ainda vai precisa de um patch. Ambos estão muito bem descritos em [12].

O Loop-AES simplifica o projeto em alguns pontos, apesar da flexibilidade oferecida pela solução oferecida pela Crypto-API. Quando o administrador do sistema desejar permitir o acesso ao sistema por diferentes usuários via chave GPG, deverá dar uma olhada na solução com Loop-AES.

SOBRE O AUTOR

Christian Ney administra de sistenas Unix e Firewalls para uma pequena empresa aérea, e trabalha em vários projetos Open Source no tempo livre.



INFORMAÇÕES

[1] GPG: *http://www.gnupg.org/*

[2] CryptoAPI: *http://www.kerneli.org/index.phtml*

[3] Instalação da Crypto-API: *http://www.kerneli.org/howto/node2.php*

[4] Patches do *util-linux* para o Red Hat/Fedora: *http://www.q-vadis.net/index.php?mID=stories&lng=en&art=5*

[5] Peter Gutmann, "Exclusão segura de dados de meios magnéticos e de estado sólido": *http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html*

[6] Documentação do Twofish: *http://www.schneier.com/paper-twofish-paper.pdf*

[7] Supermount: *http://supermount-ng.sourceforge.net/*

[8] Comparativo da AES: *http://www.schneier.com/paper-aes-performance.pdf*

[9] Comparações entre o Twofish e o Rijndael: *http://www.schneier.com/paper-twofish-final.pdf*

[10] Tamanho de chaves: *http://mitglied.lycos.de/cthoeing/crypto/keylen.htm*

[11] Loop-AES: *http://loop-aes.sourceforge.net/*

[12] Loop-AES-Readme: *http://loop-aes.sourceforge.net/loop-AES.README*