

DM-Crypt, LUKS e Cryptsetup

Mensagem secreta

Se seus segredos são valiosos, que tal criptografar seus discos rígidos?

POR CLEMENS FRUHWIRTH AND MARKUS SCHUSTER

A criptografia de arquivos é um método bastante popular de assegurar que seus dados estão bem guardados e que sua confidencialidade é mantida. Um intruso que consiga passar pelo firewall não será capaz de ler os arquivos em seu PC se eles estiverem criptografados, certo?

Na verdade, a criptografia de arquivos individuais – oferecida por programas como o GnuPG, por exemplo – apaga alguns dos seus rastros, mas não todos. Um invasor ainda pode aprender coisas sobre seu sistema. Pode, por exemplo, meter o nariz em seus arquivos temporários, configurações, históricos de comandos, logs... Não é difícil que encontre a chave que decodifique seus tão amados segredos! A pasta `/var/spool/cups`, por exemplo, é uma arca de tesouro que contém cópias de arquivos que você já imprimiu no passado. Ferramentas como o *Gnome Thumbnail*

Factory também são danosas, já que podem mostrar miniaturas sem criptografia de suas imagens criptografadas.

Em vez de ficar penteando macaco, criptografando todo e qualquer arquivo que apareça pela frente e tomando cuidado com o “lixo” produzido pelo sistema, que tal tomar uma medida mais eficaz? Os usuários de Linux têm à sua disposição um modo de criptografar discos rígidos e partições inteiras com o *DM-Crypt*. O módulo do kernel *dm-crypt* trabalha diretamente com os dispositivos de bloco. O processo é completamente transparente para o programa que estiver acessando o disco – desde que, obviamente, o usuário tenha acesso permitido aos dados. O *DM-Crypt* criptografa o chamado *backing device* ou dispositivo de apoio (que, na realidade, é fisicamente o próprio disco rígido) e usa um dispositivo de bloco virtual para que o conteúdo possa ser acessado sob o dis-

positivo `/dev/mapper`. Os usuários podem acessar o dispositivo de bloco virtual para configurar e montar o sistema de arquivos. Este artigo examina a tecnologia por trás do *DM-Crypt* e do novíssimo *LUKS* (*Linux Unified Key Setup* ou configuração unificada de chaves no Linux).

Próxima parada: criptografia

O *DM-Crypt* foi desenvolvido sobre uma camada de abstração bastante versátil do kernel chamada de *device mapper* (daí o nome *DM* – em português seria mapeador de dispositivos). Os módulos do *device mapper* são configurados pelas chamadas *Tabelas DM* – arquivos simples de texto que especificam como o mapeador de dispositivos deve lidar com o disco virtual. O programa *dmsetup* interpreta esses arquivos de texto e usa chamadas à função *ioctl()* para passar os detalhes da transação ao kernel.

O formato da tabela DM usada pelo DM-Crypt é desnecessariamente desajeitado. O software de criptografia espera que a chave tenha tamanho fixo e seja uma cadeia hexadecimal de símbolos. O módulo usa a chave para criptografar os dados do dispositivo de blocos. Entretanto, deixar a chave permanentemente em uma tabela DM seria o mesmo que deixar a chave de casa pendurada na porta ao sair de manhã. Em vez disso, a chave precisa ser gerada sempre que o volume for montado.

Digitar de cabeça 32 caracteres em hexadecimal todos os dias antes do café da manhã talvez não seja a maneira como as pessoas gostariam de começar o dia, mas o programa *Cryptsetup* pode ajudá-lo nesse pormenor. Ele é uma ferramenta que gera uma chave criptográfica a partir de uma (muito mais simples) frase-senha. Depois disso, passa a senha ao kernel. A **figura 1** mostra o ambiente em que vive o *Cryptsetup*.

Dois recursos importantes do *Cryptsetup* podem ser parametrizados: geração de chave e criptografia. O primeiro específico como o *Cryptsetup* irá gerar a chave a partir de uma senha informada pelo usuário. O padrão é um algoritmo de *hash*, o que dá ao usuário total liberdade para selecionar uma senha de qualquer tamanho. O hash sempre transformará essa senha em uma cadeia de caracteres com um número fixo de bytes. A **figura 1** mostra o *Cryptsetup* usando o padrão: o hash *Ripemd-160* gera uma chave de 256 bits.

Dois parâmetros precisam ser escolhidos para o processo de criptografia: o algoritmo a ser usado e o modo. O *Cryptsetup* passa esses parâmetros e a chave gerada a partir da senha escolhida pelo usuário para o kernel. Depois disso, o módulo do DM-Crypt coordena os trabalhos daquele ponto em diante, usando a confiável *Crypto-API* para lidar com a criptografia.

Use a força, LUKS

Infelizmente, há uma desvantagem em se usar o *Cryptsetup*. Ele separa, da informação a ser criptografada das instruções que o sistema deve seguir para criptografá-la. Os parâmetros do *Cryptsetup* são guardados em scripts e arquivos de configuração que, por motivos óbvios e auto-explicativos, não podem estar em partições criptografadas. Se esses arquivos forem perdidos ou os dados estiverem em uma mídia removível (e você não se lembrar das configurações), os dados criptografados estarão irremediavelmente perdidos. O LUKS (*Linux Unified Key Setup*) acaba de vez com essa segregação.

“Primeiro que tudo” (como diria um saudosos e folclórico dirigente futebolístico paulistano), o LUKS é um padrão formal [1], implementado na ferramenta *Cryptsetup-LUKS* [2] (**figura 2**). Esta última é, na verdade, uma implementação paralela baseada no *Cryptsetup* original (ou seja, um “*fork*”). A despeito do nome, o LUKS

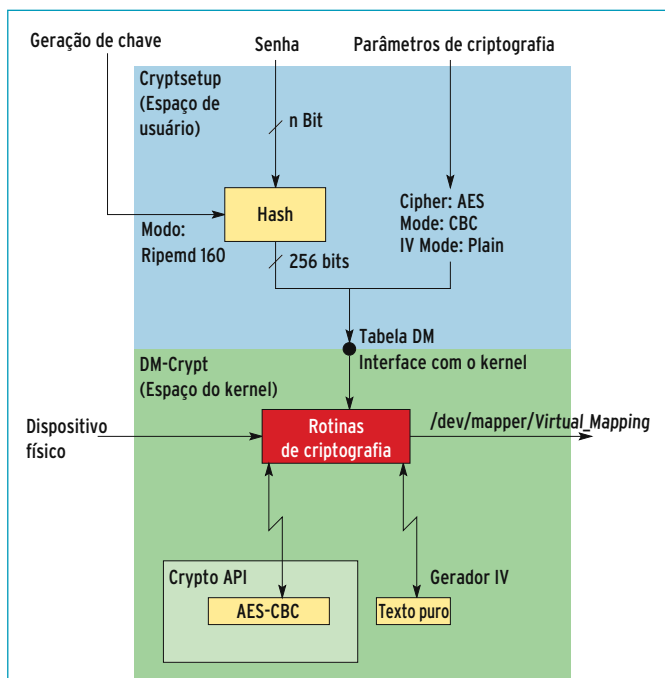


Figura 1: O *Cryptsetup* (no alto) solicita uma senha e usa um *hash* para criar uma chave de tamanho fixo, que é passada ao kernel (ao centro). O *DM-Crypt* (embaixo) usa essa chave para criptografar os dados do disco rígido (ou outro dispositivo qualquer).

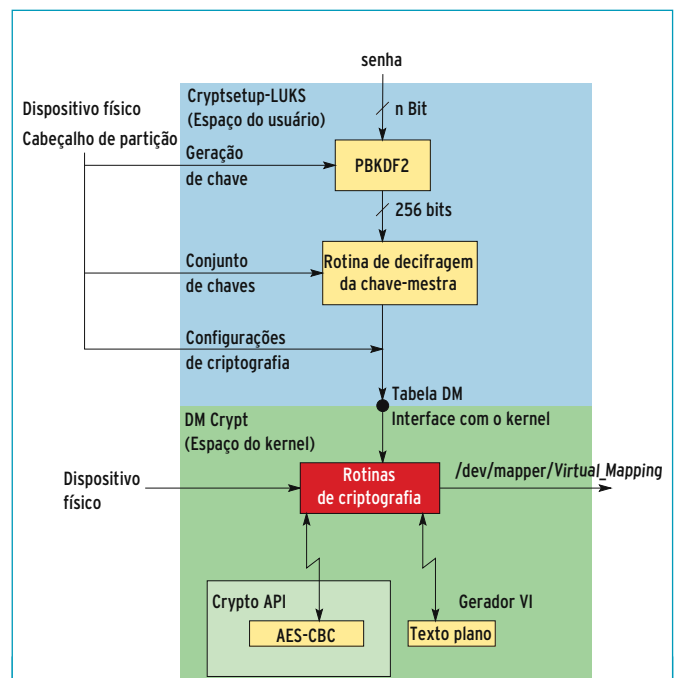


Figura 2: O *Cryptsetup-LUKS* guarda os parâmetros da partição criptografada em um cabeçalho no próprio dispositivo de armazenamento (no alto, à esquerda).

não está restrito ao Linux. Ele define um cabeçalho para a partição criptografada pelo DM-Crypt (**figura 3**); o cabeçalho inclui todas as informações necessárias para geração da chave, modo e algoritmo de criptografia. Como o cabeçalho é parte da partição criptografada, as configurações ficam à disposição no lugar em que são mais necessárias.

O Cryptsetup-LUKS e o Cryptsetup original também diferem no tocante ao modo de gerar a chave a partir da frase-senha – veja a **figura 2**. A administração de senhas do LUKS baseia-se em três conceitos: hierarquia de chaves, PBKDF2 e armazenamento seguro de informações, à prova de perícia técnica.

Gerenciamento seguro de senhas

O obsoleto Cryptsetup repassa a chave, gerada a partir da senha, diretamente ao kernel. A maior desvantagem disso é que o software precisa re-criptografar todos os dados – ou seja, toda a partição ou quicá o disco inteiro – sempre que a

senha muda. O Cryptsetup-LUKS introduz um recurso adicional para administração de senhas que acaba com essa necessidade. A hierarquia de chaves promove uma camada adicional de criptografia entre as chaves derivadas de senhas e a chave usada pelo kernel para proteger os dados na partição. Com isso, as chaves derivadas de senhas protegem apenas e tão-somente a chave mestra. Esta última é que, no devido momento, criptografa os dados da partição (**figura 2**).

Para mudar a senha, o Cryptsetup-LUKS decifra a chave-mestra usando a senha antiga e a recriptografa usando a nova. Depois disso, grava o novo valor da chave-mestra no lugar onde estava a anterior. Como a chave-mestra não criptografada não é afetada por esse processo, a partição criptografada ainda pode ser acessada. Isso pode salvar a sua pele caso precise trocar a senha de uma partição de 120 GBytes... Esse esquema de hierarquia de chaves reduz o tempo necessário, nesse caso, de um dia inteirinho para alguns segundos. Quando encontrar seus amigos

hoje no boteco, beba uma caneca bem grande de cerveja forte em reverência à hierarquia de chaves.

O LUKS armazena a chave-mestra já criptografada no cabeçalho da partição sem impor a condição de que a cópia seja única. Para que várias senhas possam ser usadas em uma única partição, o LUKS pode guardar várias cópias da chave mestra, todas equivalentes entre si, e criptografá-las cada uma com uma senha diferente. Todas essas senhas abrem ao usuário o acesso ao conteúdo não-criptografado do disco. Isso é particularmente útil se quisermos guardar uma segunda senha para o caso de a primeira apresentar problemas, ou para o caso de querermos que mais de uma pessoa acesse a partição, cada uma com sua senha. O LUKS reserva “chaveiros” para até oito senhas (**figura 3**).

Melhor que um hash

Assim como o Cryptsetup, o LUKS precisa de um algoritmo de hash para converter a frase-senha do usuário em uma chave de tamanho fixo. Para isso, o LUKS usa o sistema PBKDF2 (*Password-Based Key Derive Function, Version 2*, a segunda versão da função de geração de chaves baseada em uma senha). O PBKDF2 é um componente PKCS#5 (*Public Key Cryptography Standard 5*, ou padrão número cinco de criptografia por chave pública). O PKCS#5 teve seus padrões e parâmetros especificados pelo documento RFC 2898 [3]. Entre outras coisas, o PBKDF2 usa o que chamamos de “tempero” (em inglês, *salting*) e expansão (em inglês, *stretching*) para se prevenir contra ataques de força bruta por dicionário.

Os usuários sempre vão preferir senhas pequenas e fáceis de lembrar. Datas importantes (aniversário, casamento) e nomes dos filhos ou de animais de estimação são muito mais comuns do que seqüências aleatórias de 32 caracteres contendo letras, números e símbolos.

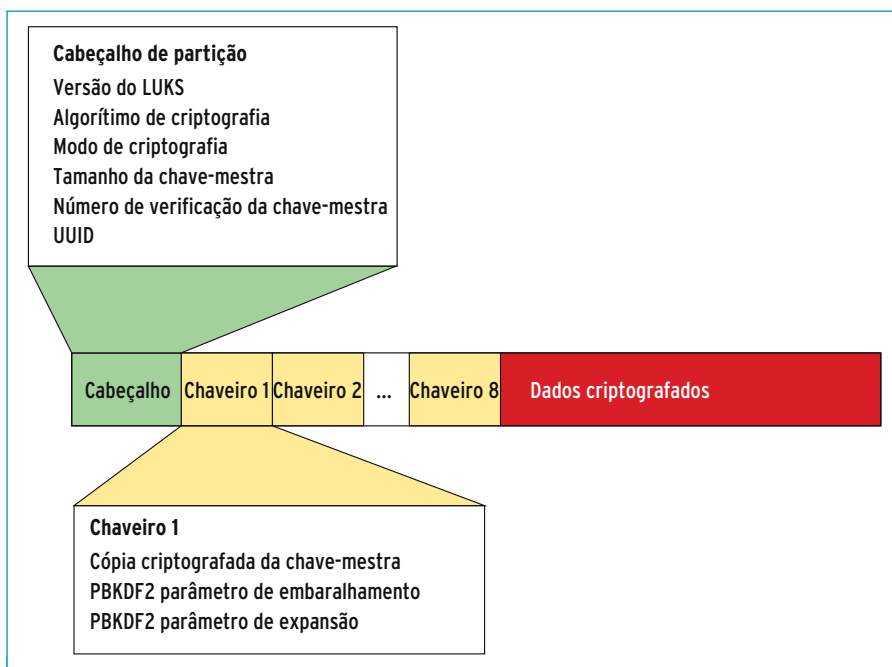


Figura 3: O LUKS fornece os parâmetros de que o Cryptsetup-LUKS precisa para gerar a chave a partir de uma frase-senha. Esses parâmetros são guardados no início da partição criptografada. Cada “chaveiro” contém uma cópia da chave-mestra usada pelo DM-Crypt para criptografar os dados.

Infelizmente, uma senha de 32 dígitos é exatamente o que você precisa para gerar uma chave de 128 bits. Mas pense: quantas pessoas em sã consciência você conhece que pensariam – e se lembrariam todo dia de manhã! – em uma senha do tipo `Sq5(oq7$01V#E5ir$Xau.a?` Não consigo pensar em ninguém sequer digitando uma senha assim, quanto mais conseguindo se lembrar dela no dia seguinte. É nesse angu de carçoço que entra a função de derivação: o usuário pode criar a senha que bem entender enquanto a função gera, baseada na senha, uma chave muito mais complexa.

Um algoritmo que triture uma senha fraca e a transforme em uma chave de 128 bits é algo digno de nota. Ele tem, em primeiro lugar, que estabelecer uma ponte sobre o (muito apropriadamente chamado de) “abismo de entropia”, que é a diferença entre o nível de aleatoriedade da senha do usuário e o nível de aleatoriedade da chave criptográfica necessária. Estufar com enchimento até completar os 32 dígitos até que produziria uma chave com o tamanho necessário, mas ela não seria nem um milímetro mais aleatória que a senha do usuário – que, via de regra, seria algo como “paco” ou “almirante” ou “12junho” e, portanto, fáceis de descobrir “no chute”.

Vamos imaginar que o usuário usou apenas palavras em português. Isso restringiria demais o alcance da senha e não produziria o caos necessário. Um agressor poderia simplesmente rodar um ataque de força bruta por dicionário, com algumas centenas de palavras bem escolhidas – ou seja, muito menos do que as 2^{128} possibilidades diferentes que uma chave de 128 bits possibilita. Algumas línguas possuem muito menos palavras do que o português, tornando a coisa mais fácil ainda. Um dicionário de alemão possui menos do que 2^{20} verbetes. Isso reduz o expoente em 108 vezes, se o compararmos ao da chave de 128 bits.

Uma redução fatal, já que praticamente qualquer um pode quebrar uma chave de apenas 20 bits...

Para anular a equação, o PBKDF2 usa uma função deliberadamente complexa para derivar a chave a partir da senha. Embora isso tome algum tempo, o usuário legítimo não se importará muito, já que a operação deve ser feita apenas uma vez. Um agressor precisaria tentar 2^{20} senhas completas. Se cada consulta durar um segundo, seriam precisos doze dias para testar todas as combinações (2^{20} segundos). Se o usuário, espertamente, combinar duas palavras para formar a senha, o ataque levaria meros 30 mil anos (2^{40} seconds). Essa barreira artificial é o que chamamos de *expansão* (*stretching*, em inglês, que entre outras coisas é o ato de esticar a massa de farinha com o rolo). O PBKDF2 faz uso de uma função de expansão que envolve um esforço computacional que varia infinitamente.

Temperando e esticando

Mas isso não é o suficiente para parar os empreendedores do mal. Um agressor poderia criar uma tabela enorme contendo entradas e saídas da função de expansão – e poderá usar isso em futuros ataques por força bruta. Para que nem isso funcione, o PBKDF2 tem mais uma carta na manga: adiciona à senha uma cadeia aleatória de caracteres antes de gerar a chave. É o que chamamos de embaralhamento ou, informalmente, de “tempero” (em inglês, eles colocam apenas sal, ou *salt*). O LUKS armazena uma cópia também dessa pitada de tempero no cabeçalho da partição.

Agora, o agressor precisa de bem mais do que o hash PBKDF2 para cada palavra do dicionário. Com o “tempero”, cada palavra do dicionário precisaria de não apenas seu hash, mas de todas as combinações possíveis entre a senha e o “tempero”. Quanto maior a pitada de tempero (ou seja, maior a cadeia de caracteres com que

vamos “dar aquele gostinho” à senha), mais tempo o agressor vai precisar para acertar a senha correta. O PBKDF2 torna astronômico o tamanho da tabela de que o agressor precisaria para quebrar a senha. O universo todinho possui menos átomos do que o número de verbetes que o dicionário do invasor precisaria ter para quebrar uma chave PBKDF2.

Já que o candidato a invasor não pode, pelos motivos citados, usar listas de palavras, é necessário voltar à boa e velha força bruta para tentar quebrar senhas.

Uma curiosidade: o mecanismo de criptografia de senhas do Unix (e isso inclui o Linux) usa um esquema semelhante. O “tempero” é, entretanto, bem menor: apenas 12 bits, armazenados nos primeiros 12 bits da senha (o primeiro caractere mais metade do segundo do hash armazenado em `/etc/shadow`).

Apagamento destrutivo

Já viu aquelas picotadoras de papel que tem no fim do corredor de grandes empresas? Elas servem para destruir documentos secretos que não podem cair em mãos erradas. Como já mencionamos anteriormente, fazer isso com arquivos em meio magnético é uma tarefa ingrata [4]. Para realizar mudanças (ou mesmo a remoção) de senhas na hierarquia de chaves, é essencial destruir completamente a cópia antiga da chave-mestra. Com um pouco de sorte, a nova senha que o usuário está gravando será gravada exatamente no mesmo setor do disco que a velha. Entretanto, sorte é algo que os usuários e os criptógrafos costumam não ter – ou, no mínimo, em que não costumam confiar.

O *firmware* de todos os discos rígidos modernos combate a destruição de dados até as últimas conseqüências – o que parece óbvio, já que o objetivo principal de um disco rígido é preservar os dados, não destruí-los. Uma das maneiras pelas quais os discos rígidos trabalham pela

integridade dos seus dados é o mapeamento de blocos defeituosos, uma técnica simples de detectar setores difíceis de ler. O firmware automaticamente copia esses setores em uma área do disco especialmente reservada para esse propósito e redireciona para ele qualquer operação de leitura ou escrita.

O setor original não pode ser apagado daí por diante, já que o firmware do disco vai redirecionar qualquer escrita para a zona reservada. Infelizmente, esses setores podem conter fragmentos da chave preservados para a posteridade. Qualquer oficina de conserto de HDs consegue acessar esses fragmentos usando um firmware modificado.

Isso é um problema e tanto para as chaves mestras do LUKS, já que elas cabem inteirinhas num setor (há setores de 128, 192 e 256 bits). Se o firmware resolver redirecionar esse setor para a zona reservada, a senha ficará ali, completa, até o fim da vida útil do HD e mesmo depois. Nenhum disco rígido, seja SCSI ou IDE, possui qualquer tipo de comando para acessar o setor original.

Passando a perna na perícia

O autor do LUKS incluiu mais uma traquinagem no programa: o *Anti-Forensic Information Splitting* (divisor de informações contra perícia técnica, ou simplesmente *AF Splitter*). Com ele, espera frustrar os esforços de peritos criminais, especialistas em recuperação de dados e “fuçadores” em geral. Para reduzir a probabilidade estatística de que traços de arquivos apagados possam sobreviver em cantos encardidos das mídias magnéticas, o AF Splitter expande os dados a um fator de quatro mil. Os dados expandidos não são redundantes; o registro completo é necessário para recuperar a chave-mestra. A operação inversa à divisão é a mesclagem: os dados são reunidos na memória RAM; basta cortar a alimentação elétrica para

que todos os dados sejam irremediavelmente apagados.

O AF Splitter distribui os dados originais (variável x) baseado na fórmula $x = a_1 + a_2 + a_3 + \dots + a_{4000}$. O algoritmo gera as variáveis de a_1 até a_{3999} aleatoriamente e, por último, calcula a_{4000} para que a equação fique equilibrada. Já o *Merger* (mesclador) soma os elementos a_i e precisa de todos eles para que a operação tenha sucesso. Não há, portanto, redundância: todos os “pedacinhos” são importantes. Se um único elemento estiver faltando, a equação não poderá ser resolvida e x não pode ser calculado.

Para triturar os dados, basta que apenas um dos 4000 setores envolvidos seja apagado – ou melhor, sobrescrito – já que o processo de mesclagem precisa de todo o registro expandido. Como qualquer um pode ver, é muito mais fácil “acertar” pelo menos um dos 4000 setores. As estatísticas mostram que esse esquema funciona maravilhosamente bem [5]. Graças ao AF Splitter, as senhas podem ser alteradas livremente pelos usuários sem que deixem “rabos” para trás. Combinando isso com as hierarquias de chaves e o PBKDF2, o “bolo” resultante mostra-se bastante eficiente para administrar as senhas de partições DM-Crypt.

Armazenagem segura de dados

Sem sombra de dúvida, o título acima é exatamente o que os usuários esperam de um sistema de criptografia de discos rígidos. O DM-Crypt trabalha com dois modos de criptografia: ECB (*Electronic Code Book* ou livro eletrônico de códigos) e CBC (*Cipher Block Chaining* ou encadeamento de blocos de cifragem). Os dois modos têm lá seus pontos fracos, todos eles aparentemente sanados pelo candidato mais promissor a substituí-los: senhoras e senhores, apresentamos o LRW-AES [6] [7] (LRW: as iniciais dos criadores Liskov, Rivest e Wagner; AES: *Advanced Encryption Standard* ou padrão avançado de criptografia).

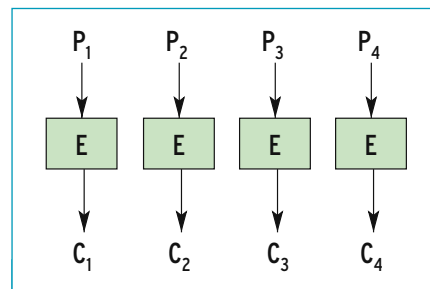


Figura 4: A criptografia em modo ECB (*Electronic Code Book*) codifica cada bloco de dados de forma independente dos outros blocos. Isso quer dizer que, se injetarmos na entrada duas cópias idênticas de um mesmo bloco de dados P_i , a função de criptografia E produzirá dois blocos C_i idênticos na saída.

O ECB (figura 4) nem mereceria ser chamado de “modo de criptografia”, pois armazena cada bloco individual de dados que, embora já criptografados, não sofrem nenhum tipo de cálculo adicional. Trocando em miúdos: para uma mesma chave, um mesmo bloco de dados será codificado exatamente do mesmo jeito. Um exemplo: a palavra “elefante” será criptografada sempre do mesmo jeito se usarmos a mesma chave. Se expressarmos o problema em linguagem matemática, uma função ECB é *bijetora* – a cada elemento da imagem corresponde a um único elemento no domínio. Isso é perigosíssimo já que, se um agressor tiver acesso a uma pequena porção dos dados antes e depois de criptografados, pode usar essa informação para inferir a chave usada.

Agora pense um pouco: o cabeçalho da partição é montado segundo um padrão. Lendo a documentação, o agressor tem como saber como o cabeçalho é sem criptografia. Depois, olhando o cabeçalho já criptografado no disco, o agressor tem como saber como o cabeçalho é *com* criptografia. Comparando as duas, é possível inferir muitas coisas.

Um exemplo prático: o agressor sabe que o primeiro setor de uma partição criptografada começa com uma série de zeros. Basta, então, anotar como esses zeros estão representados criptografica-

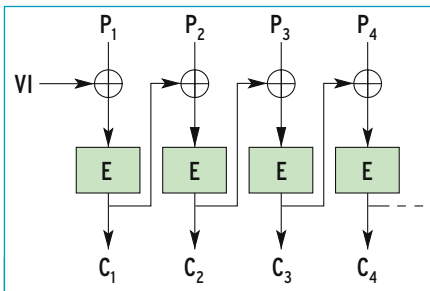


Figura 5: O modo de criptografia CBC (*Cipher Block Chaining*) faz uma operação binária XOR entre o resultado criptográfico do bloco anterior e o bloco que se quer criptografar. Isso assegura que blocos idênticos de dados não criptografados vão produzir representações criptográficas diferentes.

mente no início da partição. Comparando a anotação com outros blocos cifrados, se o agressor encontrar blocos idênticos, saberá que o conteúdo descriptografado dessa posição no disco também possui zeros – e nem precisa da chave para isso! Analogamente, qualquer porção do disco da qual o agressor saiba qual o conteúdo sem criptografia e possa ver no disco o mesmo conteúdo já criptografado pode ser usado para, na comparação, encontrar blocos de dados semelhantes.

Esconde-esconde

Há, basicamente, dois métodos para esconder essas redundâncias nos dados antes da criptografia. Um deles é adicionar mais um componente ao processo de criptografia; esse componente deve ser único para cada local do disco. Com isso, dados que sejam idênticos (por exemplo, dos blocos de texto) que sejam armazenados em posições diferentes no disco resultariam em uma representação criptográfica diferente.

O segundo método usa um modo de criptografia que leva os blocos já criptografados em conta. A maneira mais fácil de implementar esse método é usar a recursividade. O modo CBC (*Cipher Block Chaining*), mesmo sendo bastante simples, é um modo bastante eficiente de criptografia com recursividade. Basicamente,

o CBC faz uma operação XOR (*exclusive or*; em português, “ou exclusivo”) entre o último bloco de dados, já criptografado, e o bloco de dados atual, ainda sem criptografia. No resultado da operação XOR, o CBC aplica o algoritmo de criptografia e usa o valor obtido na operação XOR do próximo bloco de dados.

A **figura 5** mostra como o CBC funciona. Mesmo quando inúmeros blocos contíguos de dados não-criptografados forem idênticos, a recursividade provoca uma espécie de “efeito bola de neve”. A interdependência criada implica em que dois ou mais blocos de dados idênticos geram, cada um, representações criptográficas diferentes.

Bola de neve

Uma das características desse tipo de recursividade é a de que a primeira rodada de criptografia produz um efeito nas subseqüentes. Isso não é lá muito útil em discos rígidos, já que isso implicaria em ter que criptografar novamente a partição toda se o conteúdo do primeiro setor for alterado. A resposta óbvia ao problema é processar cada setor (onde cabem vários blocos de dados) de forma independente.

Mas aí o problema volta: dois setores com conteúdo idêntico terão representações criptográficas também idênticas. Embora os setores sejam muito maiores que os blocos que contêm, o conteúdo ainda seria idêntico – imagine, por exemplo, o usuário criando várias cópias do mesmo arquivo (`[Ctrl]+[C]`, `[Ctrl]+[V]`, `[Ctrl]+[V]`, `[Ctrl]+[V]`, `[Ctrl]+[V]` ...). Aqui aplicamos nosso primeiro truque: o número do setor é usado para “temperar” a criptografia. Basta aplicá-lo ao vetor de inicialização (VI, **figura 5**). Duas modificações diferentes no mesmo bloco de dados desencadeiam duas “avalanches” diferentes e produzem duas representações criptográficas diferentes.

O padrão DM-Crypt injeta o número do setor diretamente no VI – é o que chamamos de geração simples (“*plain*”) de

VI. Infelizmente, ainda estamos vulneráveis a um ataque conhecido como “marca d’água”. Nele, o agressor “planta” dados no disco e mais tarde confere como ficaram depois de criptografados. Assim, pode montar uma tabela com várias amostras – ainda sem precisar da chave!

As marcas d’água podem conter até 5 bits de informação [5]. Um agressor poderia, por exemplo, preparar uma mensagem de email com elas. Ao enviar esse email à vítima, fica fácil descobrir onde a vítima guarda seu correio eletrônico. Adicionar marcas d’água a arquivos MP3, imagens, documentos de texto (ou qualquer outro) é bem simples. Um padrão desconfiado poderia enviar esse arquivo a qualquer um de seus funcionários e bisbilhotar sua privacidade de forma fácil e limpa. Sem precisar descriptografar os dados, o espião tem acesso a informações valiosas sobre o disco rígido da vítima.

Mas há esperança! Um gerador de VIs chamado ESSIV (*Encrypted Salt-Sector IV* ou VI para criptografia “temperada” de setores) pode proteger os fracos e oprimidos. Para que o ataque por marcas d’água funcione, é preciso estabelecer um relacionamento simples entre os VIs de dois setores contíguos. Nos VIs do tipo *plain*, é fácil inferir (“chutar”) o valor do próximo: o VI do setor *n* é seguido do VI do setor *n+1*. O ESSIV adiciona uma pitada de complexidade à seqüência, tornando impossível ao agressor calcular a seqüência sem saber ao menos uma pequena parte da chave secreta (veja o quadro **ESSIV**).

O processo de marcas d’água joga um balde de água fria nisso tudo: basta aplicar o valor *Pi-1* (em vez de *Pi-1*) ao segundo setor. O VI do setor dois é, por definição, uma unidade mais do que o VI do setor um. Para compensar, basta subtrair o valor 1 de *Pi-1* (**figura 6b**). Se o agressor conseguir que todos os setores subseqüentes apliquem o mesmo *Pi-i* do primeiro setor, a representação criptográfica, no disco, será idêntica. ➔

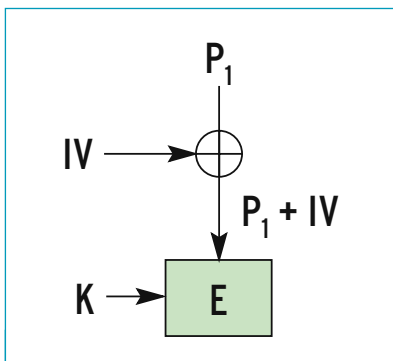


Figura 6a: Um CBC tradicional começa a criptografia com um XOR entre o IV e o primeiro bloco de dados não criptografado.

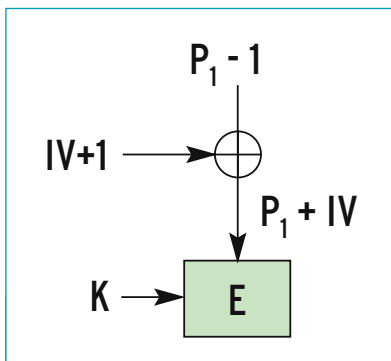


Figura 6b: Ataques por “marca d’água” compensam a mudança do IV pela reversão da mudança em P1.

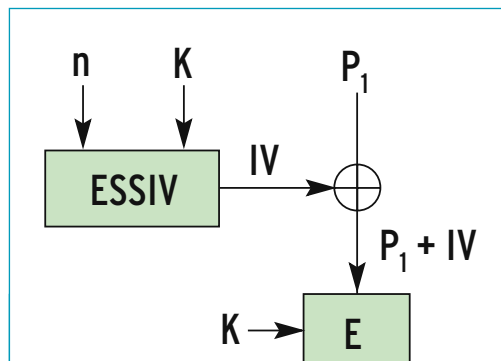


Figura 6c: o ESSIV impede que o agressor calcule o IV, já que não possui a chave secreta K.

O ESSIV (*Encrypted Salt Sector IV*) dá cabo desse problema. Ele passa o número do setor a uma função. O resultado obtido depende da chave secreta (**figura 6c**). O agressor não consegue mais manipular o P_{i-1} no setor dois para compensar a diferença no IV, já que não tem acesso à chave secreta – que, como todos sabemos, é gerada pela senha do usuário.

“Branqueamento” de dados

Você pode estar se perguntando por que o DM-Crypt usa uma combinação da manipulação de dados baseada na posição do disco rígido com a recursividade – apenas a manipulação citada seria já suficiente. Há, entretanto, uma razão histórica para usar o CBC: além de ser testada e aprovada, suas propriedades foram esmiuçadas por muitas pessoas. As alternativas, que baseiam-se inteiramente no número do bloco, são ainda muito recentes em termos criptográficos.

ESSIV

Para gerar uma marca d’água, um agressor precisa criar dois setores idênticos no disco. O objetivo é manipular o mecanismo de criptografia de forma a obter duas representações criptográficas idênticas quando dois setores idênticos são gravados. Na figura 5 podemos ver que o agressor consegue identificar todos os valores de entrada P_i , mas não o IV. Esse é o valor usado para modificar o primeiro bloco de dados, conforme mostra a **figura 6a**.

O novíssimo modo LRW de criptografia leva o número do bloco em consideração quando criptografa as coisas – e o faz de forma simples e eficiente. Em primeiro lugar, o LRW calcula um “fator de branqueamento” baseado na chave secreta e no número do bloco. Depois, realiza uma operação de adição (“soma”) entre o fator de branqueamento e o bloco de dados a criptografar. O resultado dessa soma é, então, criptografado. Depois da criptografia, o “branqueamento” é aplicado novamente. A **figura 7** talvez esclareça o processo. Os dois processos são conhecidos como “pré-branqueamento” e “pós-branqueamento”. Os dados a serem criptografados são, com eles, associados à sua posição específica no disco. Com isso, é impossível que dois blocos de dados idênticos, quando gravados em posições diferentes do disco, tenham a mesma representação criptográfica.

O LRW também evita as vulnerabilidades atribuídas ao modo CBC e, ainda por cima, melhora o desempenho geral. Enquanto o CBC não se dá bem em arquiteturas multiprocessadas, já que cada iteração é baseada nos resultados de uma etapa anterior, o LRW extrai o máximo de todos os processadores disponíveis. O autor do LUKS, Clemens Fruhwirth, que por acaso é também co-autor deste artigo, já implementou e testou o LRW no DM-Crypt. A nova versão, que já trabalha com LRW, deve chegar às prateleiras a qualquer momento.

Frustrado pelo kernel

Pois é, mas no momento o LRW não está disponível no DM-Crypt – e por um bom motivo. O gerenciador de memória do kernel do Linux trabalha com um modelo chamado “*high/low memory*”. Isso significa que um mesmo processo, se originado de um módulo do kernel, pode acessar apenas e tão somente duas áreas de dados na memória. A implementação LRW baseia-se em uma reimplementação genérica do *Scatterwalk*

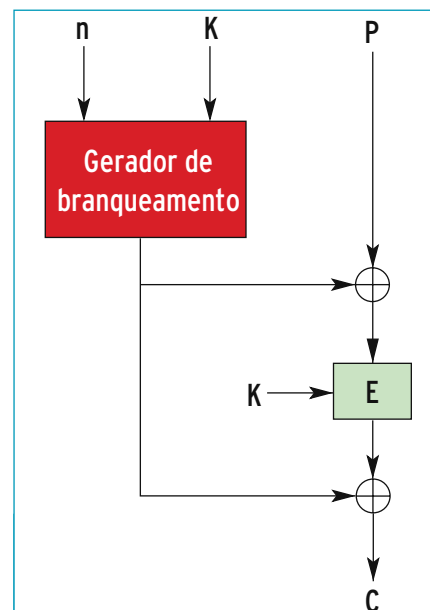


Figura 7: O modo criptográfico LRW não usa recursividade. Em vez disso, evita o ataque ao estilo do ECB (**figura 4**) simplesmente “branqueando” os dados. O “fator de branqueamento” é calculado levando em conta a posição do bloco no disco (n) e a chave secreta (K).

(parte da *Crypto-API*), que precisa necessariamente acessar simultaneamente um número arbitrário de áreas na memória alta. Como o kernel, por enquanto, só permite que duas áreas sejam acessadas, a nova implementação não iria conseguir fazer o que o autor imaginou. Extremamente frustrado, ele simplesmente desistiu de tudo [8].

Até o presente momento, o DM-Crypt é a implementação mais segura da dobradiça CBC-ESSIV – ou, pelo menos, até que algum programador de verdade, que não dê a mínima pelota para as discussões inúteis, fúteis e intermináveis da lista de discussão do kernel [9], se digna a criar um substituto decente para o Scatterwalk. Os autores deste artigo ficariam assaz felizes se vissem isso acontecer ainda nesta encarnação. O LRW já está pronto e implementado de uma maneira que obedece a todos os padrões.

Instalação

Para usar o DM-Crypt, o Cryptsetup e o LUKS, são necessários alguns módulos do kernel e uma ferramenta no espaço do usuário. As opções para o DM-Crypt estão escondidas sob *Device Drivers | Multi-device support | Device mapper support* no menu de configuração do kernel e abaixo de *Crypt target support* (figura 8) na mesma seção. Observe que é necessário ativar a opção *Prompt for development and/or incomplete code/drivers* em *Code maturity level options*, senão o Crypt-Target permanecerá oculto.

Como o DM-Crypt depende das funções da Crypto-API, é necessário escolher pelo menos um algoritmo em *Cryptographic options | Cryptographic API* (figura 9). O autor recomenda AES. Um único algoritmo de criptografia é tudo o que você precisa. A ferramenta Cryptsetup-LUKS

cuida do resto – como, por exemplo, gerar a chave secreta a partir da senha.

Muitas distribuições Linux ativam essas opções por padrão. Para ver se a sua fez isso, verifique com o comando `modprobe dm-crypt`. O DM-Crypt é um componente oficial do Linux desde a versão 2.6.4 do kernel; o gerador ESSIV precisa do kernel 2.6.10 ou posterior.

O LUKS pode ser baixado de [2]. Há pacotes para Debian, Gentoo, Suse e Red Hat; no Fedora, o componente *cryptsetup-luks* já é instalado por padrão. Se você for um feliz usuário do Slackware (ou outras distribuições), baixe o código fonte e cante a velha canção gaulesa de instalação de programas: `./configure && make && make install`. Não se esqueça de que as bibliotecas *Libpopt*, *Libgcrypt* (versão 1.1.42 ou posterior) e *Libdevmapper* devem estar instaladas.

Listagem 1: Cryptsetup-LUKS

```
01 $ dd if=/dev/zero of=secretpraxuxu.loop bs=52428800 count=1
02 1+0 records in
03 1+0 records out
04 $ losetup /dev/loop0 secretpraxuxu.loop
05 $ cryptsetup -c aes-cbc-essiv:sha256 -y -s 256 luksFormat /dev/loop0
06
07 WARNING!
08 =====
09 This will overwrite data on /dev/loop0 irrevocably.
10
11 Are you sure? (Type uppercase yes): YES
12 Enter LUKS passphrase: *****
13 Verify passphrase: *****
14 $ cryptsetup luksOpen /dev/loop0 secretpraxuxu
15 Enter LUKS passphrase: *****
16 key slot 0 unlocked.
17 $ mkfs.xfs /dev/mapper/secretpraxuxu
18 [...]
19 $ mount /dev/mapper/secretpraxuxu /mnt
20 $ umount /mnt
21 $ cryptsetup luksClose secretpraxuxu
22 $ cryptsetup luksAddKey /dev/loop0
23 Enter any LUKS passphrase: *****
24 key slot 0 unlocked.
25 Enter new passphrase for key slot: *****
26 $ cryptsetup luksDelKey /dev/loop0 0
27 losetup -d /dev/loop0
```

Cryptsetup-LUKS

O programa chama-se *cryptsetup* e possui muitas ações e parâmetros. É daquelas ferramentas para Linux que permitem ao administrador associar sistemas de arquivos a dispositivos de bloco para que possam ser montados. A **listagem 1** dá um exemplo. Para que nossas experiências não causem muita celeuma em seu sistema (afinal, sabemos que você está testando em uma máquina de produção, não é mesmo? Tsc, tsc, tsc...) o comando `dd` na **linha 1** cria um recipiente de apenas 50 Mbytes, que é depois transformado em um dispositivo de bloco (com um *loop*) na **linha 4**.

Inicialmente, a ação mais importante do Cryptsetup é `luksFormat`, que prepara o dispositivo físico (em nosso teste, o dispositivo de bloco que acabamos de criar) para o ambiente criptografado. Há ainda a etapa em que decidimos qual o algoritmo de criptografia a ser usado. A ação de formatação pede o dispositivo de blocos e, opcionalmente, um arquivo – cujo conteúdo será usado como senha. O LUKS se refere a

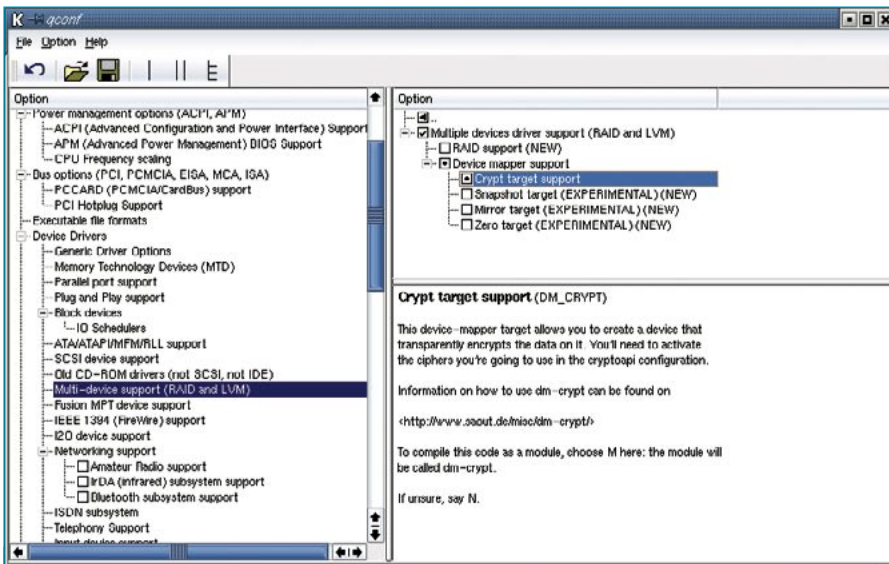


Figura 8: O mapeador de dispositivos está localizado em *Device Drivers | Multi-device support (RAID and LVM) | Device mapper support* no menu de configuração do kernel. O *Crypt target support* é necessário para o DM-Crypt.

esse arquivo como *arquivo chave* (*key file*). Os seguintes parâmetros nos serão bastante úteis:

- `-c` especifica o algoritmo e o modo de encadeamento e o gerador de VIs. Esses três parâmetros devem ser separados por um hífen (padrão: `aes-cbc-plain`). A variante mais segura seria `aes-cbc-essiv:sha256`.

- `-y` ordena que o Cryptsetup pergunte duas vezes pela senha para evitar erros de digitação.

- `-s` especifica o comprimento da chave. Na [linha 5](#) da [listagem 1](#) podemos ver o comando completo. Ao digitar `YES` na [linha 11](#), o usuário concorda em destruir os dados pré-existentes na partição a ser criptografada. Depois, confirma a senha ([linha 12](#)) e a repete (na [linha 13](#)). Entretanto, observe que ele só pede essa repetição porque o parâmetro `-y` foi explicitamente declarado).

Mapeamento do sistema de arquivos

Para poder usar o dispositivo de blocos que acabamos de criar, o Cryptsetup-LUKS precisa mapear o dispositivo de

blocos físico, associando-o ao dispositivo virtual. A ação `luksOpen`, que solicita os nomes dos blocos físico e virtual como parâmetros, toma conta disso ([linha 14](#)). Se a senha estiver em um arquivo (ver `luksFormat`), o Cryptsetup precisa que o parâmetro `-d` seja informado, contendo o nome do arquivo de chaves. Em nosso

exemplo não usamos o arquivo; o usuário digita sua própria senha ([linha 15](#)).

O Cryptsetup-LUKS cria automaticamente o dispositivo de bloco com o nome que especificamos (no caso, `secretopraxuxu`) sob o diretório `/dev/mapper/`. O comando `mkfs.xfs` na [linha 17](#) formata o dispositivo no sistema de arquivos XFS. Pronto! Podemos montar o dispositivo como mostrado na [linha 19](#). Não esqueça de desmontá-lo antes de aceitar as alterações! ([linha 20](#)).

Dando o fora

Depois de terminar, desmapeie o disco criptografado para evitar que algum invasor tenha acesso a seu conteúdo. A ação `luksClose` dá conta do recado. É preciso informar o nome do dispositivo virtual ([listagem 1, linha 21](#)).

Como mencionado anteriormente, o Cryptsetup-LUKS pode trabalhar com mais de uma senha para cada dispositivo. Por conta disso, é bem fácil mudar uma senha comprometida sem ter que criptografar todos os dados novamente – uma bela economia de tempo. A ação `luksAddKey` pede como parâmetro o nome do dispositivo ([listagem 1, linha 22](#)). Depois

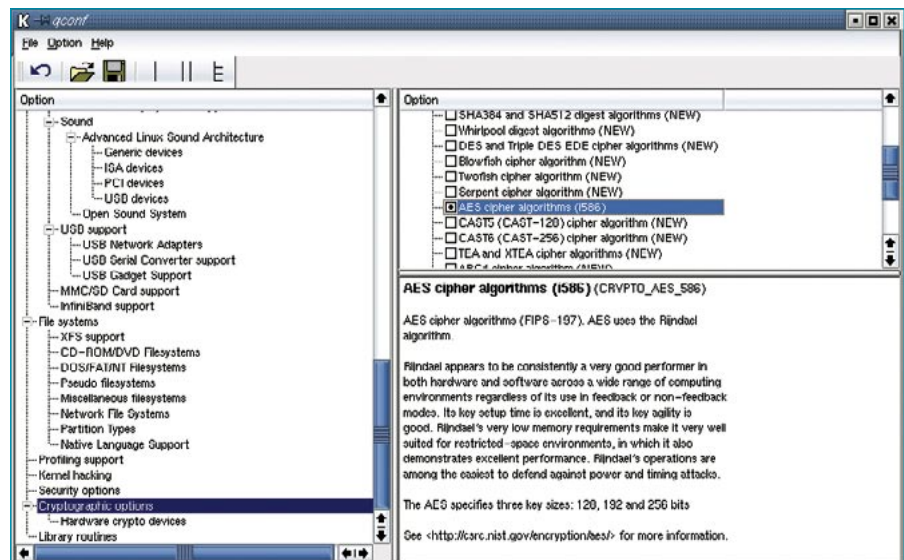


Figura 9: Como o DM-Crypt depende da Crypto-API para suas funções de criptografia, é necessário selecionar pelo menos um algoritmo em *Cryptographic options | Cryptographic API*. Por enquanto, o algoritmo mais indicado é o velho e fiel AES.

de digitar qualquer uma das senhas atuais válidas para aquele dispositivo, a ferramenta solicita a nova. Em vez de senha, o usuário pode preferir especificar um arquivo chave.

Já a ação `luksDelKey` (linha 26) simplesmente apaga uma senha existente. Deve-se informar o dispositivo físico e o número do “chaveiro” a apagar. Relembrando, o que chamamos de “chaveiro” é a posição em que aquela chave está gravada. Como o `Cryptsetup-LUKS` mantém até oito senhas por padrão, os chaveiros de 0 a 7 são tudo o que você precisa. O programa informa em qual chaveiro uma senha está armazenada ao chamar `luksOpen` (linha 16) ou `luksAddKey` (linha 24).

Fatos da vida

No mundo de Hollywood, se a CIA, a máfia ou o presidente corrupto de uma empresa forem usar o `Cryptsetup-LUKS` e o `DM-Crypt` para criptografar dados comprometedores, os espectadores já saberão desde o início que isso vai falhar e o mocinho vai descobrir tudo. Graças à Virgem, temos certeza que os diretores de cinema não dão muita bola para a realidade. Os usuários do Linux podem se recostar na poltrona do cinema e abrir um sorriso delicioso, tendo a certeza de que sabem muito mais do que esse bando de ignorantes que ganham milhões filmando bobagens. ■

INFORMAÇÕES

- [1] Clemens Fruhwirth, "Especificação do formato LUKS" (em inglês): luks.endorphin.org/LUKS-on-disk-format.pdf
- [2] Site oficial do LUKS: luks.endorphin.org
- [3] RFC 2898, "PKCS #5: Especificação de criptografia baseada em senhas, versão 2" (em inglês): rfc.net/rfc2898.html
- [4] Peter Gutmann, "Apagamento seguro de dados em mídias magnéticas e em memórias de estado sólido" (em inglês): www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html
- [5] Clemens Fruhwirth, "Novos métodos para criptografar discos rígidos" (em inglês): clemens.endorphin.org/publications
- [6] Moses Liskov, Ronald L. Rivest e David Wagner, "Brincando com a cifragem de blocos" (em inglês): www.cs.berkeley.edu/~daw/papers/tweak-crypto02.pdf
- [7] IEEE, "Proposta de documento para a criptografia manipulável de blocos estreitos" (em inglês): www.siswg.org/docs/LRW-AES-10-19-2004.pdf
- [8] Clemens Fruhwirth, "O LRW para Linux está morto" (em inglês): grouper.ieee.org/groups/1619/email/msg00253.html
- [9] Mudanças no Scatterwalk (em inglês): lkml.org/lkml/2005/1/24/54

Tecnologias IP disponíveis pela Commlogik

Rapidez, economia, confiabilidade e sucesso para seus negócios.

Sua grande oportunidade de parar de “improvisar” com hardwares!

Faça você também como algumas empresas, utilizando equipamentos adequados e confiáveis por um preço acessível para as suas soluções de telefonia IP. Viabilize seus negócios de forma rápida e segura beneficiando-se dos recursos encontrados em softwares e hardwares IP de primeira linha.



Placas E1 / Ramais IAXy



Versão Empresarial do Software Open Source PBX



Gateways



Hardware CT



Channel Banks para ramais



Telefones IP, Áudio e Vídeo Conferência IP



SIP Proxy / IP-PBX



Servidores IP



Commlogik do Brasil Ltda.
 Av. das Nações Unidas, 13.797 - Bl. II - 6º andar - Morumbi - 04794-000 - São Paulo - SP
 Tel.: (11) 5503-1011 - Fax: (11) 5506-1033
 vendas@digiumbrasil.com.br

www.commlogik.com.br