Proteção de Memória com o PaX e o Stack Smashing Protector

Pax Romana

Aqui está um patch para o kernel que vai trazer um pouco de paz de espírito aos usuários paranóicos no que diz respeito à segurança. Peter Busser, desenvolvedor do Adamantix, explica os princípios básicos do PaX (em latim, "paz") e justifica sua inclusão como um módulo de sua distribuição ultra-segura. POR PETER BUSSER

Muito tem se falado e escrito sobre segurança de sistemas. O assunto do momento são as maneiras de se limitar o que os processos podem fazer com outros objetos no sistema, como por exemplo arquivos, dispositivos e memória compartilhada. Com tal limitação, os especialistas esperam aperfeiçoar a integridade do sistema, a confidencialidade dos dados e a disponibilidade dos serviços. Prova cabal dessa preocupação é a enorme quantidade de *patches* para o kernel do Linux e mesmo de programas *user space* que existem para esse fim. Um belo exemplo é o RSBAC [1], usado pela distribuição Adamantix [2] para atingir esses objetivos.

Pouco se fala, entretanto, sobre a proteção dos únicos objetos realmente ativos em um sistema Linux: os próprios processos! Mais importantes que os arquivos, dispositivos e memória já citados, os processos desempenham um papel crucial no Linux: são o único lugar no sistema em que algum código pode ser realmente executado. A despeito dessa importância, o mundo Linux tem uma triste tradição de ignorar o problema da proteção do quinhão de memória usado pelos processos. Com isso, um número avassaladoramente alto de *exploits* utiliza-se da corrupção da memória (os famosos *buffer overflow*) dos processos para atingir seus objetivos malévolos.

Por que é tão importante proteger a memória?

Em um mundo cor-de-rosa, nenhum software seria infestado por "bugs" e, portanto, nenhum maldito invasor seria capaz de conseguir acesso de escrita e leitura na memória usada pelos processos. Há pessoas clamando a altos brados que, para ingressar nesse mundo perfeito, bastaria parar de desenvolver em C e começar a usar Java, C-LISP, Ada ou qualquer outra linguagem que considerem ser a melhor do universo. Como se a vida fosse simples assim...

A afirmação acima nunca foi comprovada, já que linguagens como Java, Perl e Python dependem de interpretadores escritos... na linguagem C! Mesmo que

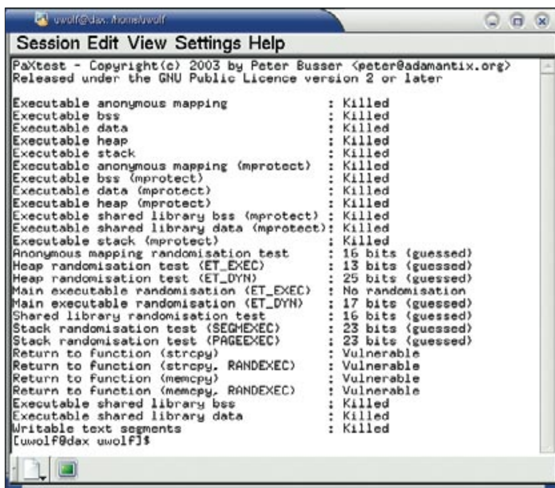

Figura 1: O *paxtest* verifica o funcionamento do PaX.

fosse verdadeira e começássemos a converter décadas de código em C para a novíssima, mais segura e sexy linguagem *XYZ*, anos e anos seriam necessários. Nesse meio tempo, os sistemas ainda dependeriam de programas escritos na insegura, vulnerável e "arcaica" linguagem C.

Nosso mundo não é perfeito e temos que aceitar o fato. Mesmo os melhores programadores cometem erros, e muitos deles podem possibilitar a corrupção da porção de memória interna usada pelos processos. Como os invasores externos sabem disso, o melhor a fazer é proteger nossa memória.

No início era o caos...

Há muitas razões para a atenção quase nula que os desenvolvedores dão à memória dos processos e sua proteção. A mais gritante é, infelizmente, a falta de compreensão da gravidade do problema. Tradicionalmente, as pesquisas sobre segurança voltaram-se quase que exclusivamente para o desenvolvimento de técnicas de controle de acesso e mecanismos de criptografia. A integridade dos processos não era um problema até que os estouros de buffer foram descobertos. Como tais falhas são mais populares a cada ano, o problema agora é assunto de Estado.

A recusa de Linus Torvalds de incluir no kernel o patch do projeto OpenWall, que impede a execução de código na pilha [3], foi um evento importante na história do Linux. Como o patch implementava apenas o controle da pilha, Linus argumentou que ele não era completo o suficiente, pois haviam outras porções de memória a proteger. O argumento era extremamente coerente, mas o que fizeram as outras pessoas em resposta a ele? Melhoraram o patch do OpenWall? Escreveram outros patches para proteger outras porções de memória? Não. Em vez disso, muitas pessoas começaram a considerar que **qualquer tipo** de proteção de memória era inútil. Isso é uma tolice tão grande quanto acreditar que a Terra é chata. Tristemente, é também uma tolice altamente disseminada e extremamente persistente.

Nos velhos e bons tempos em que todas as máquinas pequenas eram computadores pessoais, o assunto "segurança" não estava na ordem do dia. Desde que o computador não se conectasse a uma rede ou que as outras pessoas se mantivessem a uma distância segura de seu teclado ou mouse, o usuário podia se considerar a salvo. Muitos dos programadores de hoje cresceram nessa realidade. Entretanto, com o crescimento da Internet a situação mudou dramaticamente. Até então era fácil ignorar o problema e dizer às pessoas que não havia perigo. Hoje, fingir que o monstro não está embaixo da cama não é mais uma opção.

...e um dia veio a PaX!

Houve inúmeros esforços para criar patches que propiciassem melhor proteção de memória desde o fatídico dia em que Linus rejeitou a contribuição do projeto OpenWall. O site oficial do PaX [4] lista alguns deles. Muitos patches foram abandonados e não são mais mantidos, deixando o PaX como único espécime ainda vivo. Graças à persistência do autor do PaX, os usuários do Linux podem desfrutar da melhor proteção de memória do mundo livre.

O trabalho no PaX começou há mais ou menos trinta meses. Depois de examinar muitos "exploits", o autor do PaX chegou à conclusão de que a única maneira de neutralizá-los seria com uma eficaz proteção da memória usada pelos processos. Infelizmente não foi usado em larga escala até que o patch do *grsecurity* começou a incluir também o PaX. O *grsecurity* é bastante conhecido e, como muita gente o usa, repentinamente as atenções começaram a se voltar para o PaX.

Muitas distribuições, como por exemplo a versão segura do Gentoo ("hardened", ou "reforçada"), possuem o patch *gr-security* e, por conseguinte, o PaX. Isso criou uma demanda benéfica para a equipe de desenvolvimento: muitas pessoas começaram a pedir a inclusão de recursos, relatar falhas e mesmo contribuir para portar o PaX para outras plataformas. Onde havia apenas um desenvolvedor agora há uma pequena – e muito ativa! – comunidade que usa e desenvolve o software.

A diferença entre o PaX e os outros patches para proteção de memória é o fato de que ele não tenta prevenir exploits específicos. Em vez disso, tenta prevenir algumas classes de exploits. Como é que é? Classes? Do que diabos esse cara está falando?

Vou tentar explicar. Animais e plantas são categorizados tendo como referência suas similaridades. Há diversas classes de animais: mamíferos, aves, anfíbios, peixes... Suponha agora que alguém invente uma cerca que proteja suas terras contra javalis selvagens. Um expediente utilíssimo se temos uma lavoura freqüentemente vítima desses animais. Entretanto, a mesma cerca será inútil contra um estouro de elefantes ou mesmo pequenos roedores. Em vez disso, uma cerca que protegesse contra TODOS os animas terrestres seria, essa sim, muito superior.

O mesmo se aplica aos exploits e ataques. No mundo do software livre, o PaX é como essa supercerca, protegendo o sistema contra toda uma classe de exploits. Em outras palavras, proteção contra TODOS os animais terrestres. Infelizmente, a cerca ainda deixa passar pássaros e insetos. Precisaríamos, então, de outros softwares para trabalhar ao lado do PaX e proteger seu sistema contra os outros tipos de animais. É triste constatar, entretanto, que os poucos softwares similares que existem atualmente não dão conta do recado. Podemos dizer, por exemplo, que o patch do OpenWall protege apenas contra os roedores. Os projetos *W^X* [5] e *exec-shield* [6] do OpenBSD prote-

gem contra todos os animais terrestres, exceto elefantes. Confuso? Falaremos mais sobre isso logo adiante.

Classes de ataque

De modo geral, há três classes de ataques que os patches de proteção de memória tentam evitar:

(1) Injeção e execução arbitrária de código.
(2) Execução de código já existente mas fora da ordem original em que estava no programa.
(3) Execução de código já existente no programa e na ordem original, mas com dados arbitrários.

Cada tipo possível de corrupção de memória pertence a uma dessas três classes. Por exemplo, muitas falhas populares usando estouros de pilha pertencem a (1). O exemplo citado por Linus Torvalds, usando a técnica return-to-libc, pertence a (2). Exploits pertencentes à classe (3) são raros, mas existem. Normalmente é mais fácil usar as classes (1) e (2). Observe que essa é uma classificação de técnicas de exploração de falhas, não das próprias falhas. Ou seja, uma mesma técnica pode ser usada para explorar diferentes bugs, e uma mesma falha pode ser explorada de mais de uma forma.

A idéia por trás do PaX é fazer com que classes inteiras de técnicas de exploração parem de funcionar. Até o momento, o PaX conseguiu parar ataques da classe (1) – ou seja, é uma cerca contra todos os animais terrestres. Os exploits da classe (2) estão com os dias contados, pois suas técnicas estão sendo discutidas. A classe (3) será tratada em algum ponto do futuro, mas as pesquisas ainda não estão concluídas. O que destaca o PaX dos outros patches de proteção de memória é o fato de lidar com a classe (1) inteira, além de não ignorar (2) e (3).

Classe (1)

Injeção e execução de código arbitrário significa que é possível:

- Sobrescrever código que já está na memória da máquina;
- Sobrescrever dados que já estejam na memória e executá-los como se fossem código;
- Carregar código do disco para a memória e executá-lo.

Se é possível sobrescrever código, um invasor pode injetar o seu próprio código, invariavelmente malicioso, num processo vulnerável e fazer com que o próprio processo execute esse código. Em vez de fazer o que o programa foi projetado para fazer, o processo faz o que o invasor deseja que ele faça.

O mesmo vale para a segunda técnica. Dados do programa, guardados numa porção da memória, são sobrescritos por dados injetados pelo invasor e executados como se fossem código. Você pode pensar que o sistema deveria separar o que é código e o que são dados, mas os programadores preferem a facilidade de tratar tudo do mesmo jeito. Como resultado, essa técnica é usada pela maioria dos ataques de estouro de buffer. O PaX assegura que código seja código e dados sejam dados, permitindo que possamos ler e gravar a memória onde os dados estão, mas nunca executar código nela. De forma semelhante, o patch permite ler e executar código na área de memória apropriada, mas *nunca* escrever nada por lá.

As duas primeiras técnicas requerem apenas acesso de escrita e execução na memória. O PaX lida com essas técnicas à sua maneira. A terceira técnica é diferente porque requer, além de operações com a memória, acesso a arquivos no disco. A eletricidade é muitas vezes usada para acentuar a eficiência de cercas. O PaX faz algo semelhante ao influenciar ACLs e outros mecanismos de acesso como o RSBAC [1]. Isso garante uma proteção perfeita contra qualquer ataque de classe (1). É o único patch de proteção de memória que vem com uma garantia desse tipo.

Classe (2)

Lembram do exemplo que Linus Torvalds deu para rejeitar o patch do projeto OpenWall? Pois é, aquele era um ataque de classe (2). Explicando de forma simples, num ataque de classe (2) o invasor sobrescreve com novos dados um endereço de memória usado para controlar a maneira como o processo trabalha. Por exemplo, a alteraração do valor do ponteiro de retorno da pilha faz com que, quando a função chamada devolve o controle ao programa que a chamou, o processamento seja desviado não para o lugar certo no programa que está sob ataque, mas para outra posição de memória qualquer à escolha do cracker. Geralmente, nessa posição de memória já se encontra algum código malicioso, cuidadosamente colocado lá pelo invasor antes dele provocar o "estouro" da pilha.

Esse é o apenas exemplo mais comum, mas há inúmeros outros lugares em que algum tipo de endereço é armazenado. Cada um deles, obviamente, é usado para uma finalidade específica dentro do programa que se está atacando e,

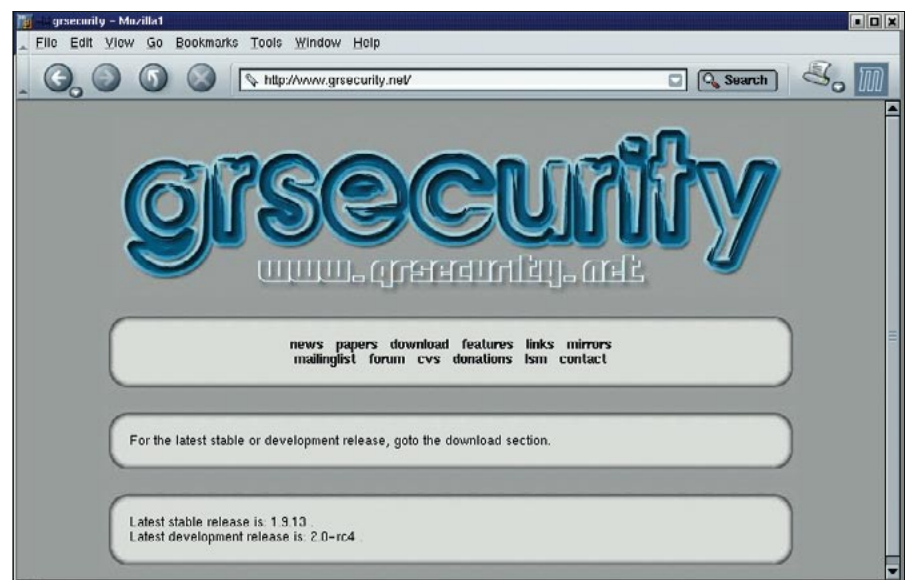

Figura 2: O *grsecurity* é uma outra alternativa em termos de proteção de sistemas Linux.

pelo menos em teoria, todos eles podem ser usados por invasores para influenciar o comportamento dos processos.

Classe (3)

Nessa classe, dados importantes são alterados durante o ataque. Nos desenhos animados, uma piada recorrente mostra o mocinho pondo um disfarce qualquer e fazendo o bandido pular num precipício ou correr na direção errada. Por incrível que pareça, os programas em geral são, via de regra, tão burros que é possível atacá-los usando o mesmo velho truque sujo. Caso um invasor possa gravar novos dados por cima de informações importantes, o programa pode ser levado a acreditar que o caminho certo a seguir é o caminho da perdição, trilhando uma lógica completamente diferente e fazendo coisas inesperadas.

Peguemos como exemplo o comando *mount*. Podemos configurá-lo para permitir que certos usuários possam montar discos. O programa faz algumas verificações para discernir entre usuários autorizados e não autorizados, montando os volumes de acordo com as permissões encontradas.

Se um invasor puder de alguma forma influenciar essas verificações, o comando mount pode acreditar que o atacante é um usuário autorizado e montar um disco para ele – ou seja, mais uma base para disparar novos ataques. Esse exemplo foi puramente hipotético. Exemplos reais são dificílimos de encontrar, pois falhas desse tipo são muito difíceis de se descobrir, que dirá de explorar.

Vista todas as suas armaduras

Proteger-se contra uma única classe de ataques não é lá uma atitude muito sábia. Linus usou um ataque de Classe (2) para driblar a proteção contra ataques de Classe (1) proporcionada pelo patch do projeto OpenWall. Isso é válido para qualquer classe: se o sistema protege contra um tipo, basta usar o outro!

As pessoas em geral (e mesmo desenvolvedores experientes) concentram-se em apenas uma classe de ataques, esquecendo-se das demais – foi o que o pessoal do projeto OpenWall fez. Pior do que isso, as pessoas costumam ignorar a utilidade de se proteger contra uma classe de ataques simplesmente porque precisam instalar outros mecanismos para se proteger de ataques em outras classes. Esse foi o erro cometido por Linus Torvalds. Em vez de esperar por uma ferramenta abrangente, o correto seria combinar diferentes mecanismos de proteção para cobrir todos os casos (e classes) possíveis. Não há isso de *tamanho único* em Unix. A abordagem correta é, e sempre será, a de ter várias ferramentas trabalhando juntas.

Stack Smashing Protector

A combinação de diferentes mecanismos de defesa é a razão pela qual o Stack Smashing Protector (SSP) foi agregado ao Adamantix. O SSP [7] (ou *ProPolice*) é um patch para o GCC que toma providências para evitar que alguns tipos de falhas das Classes (1), (2), e (3) possam ser exploradas. Mais precisamente, ele cria defesas contra os chamados *estouros de pilha* (*stack overflows*).

O SSP usa dois mecanismos para chegar a esse objetivo:

- Posiciona uma espécie de “mina terrestre” na pilha quando detecta uma função potencialmente perigosa. O mecanismo de detecção não é à prova de imbecis – tanto pode detonar a “mina” sem perigo como pode falhar e deixar a pilha ser estourada;
- Muda a ordem das variáveis locais, deixando as mais perigosas o mais perto possível da “mina”. Isso aumenta muito a possibilidade de detecção.

O termo “mina terrestre”, embora seja uma boa alegoria para o mecanismo, não virou jargão. A palavra mais conhecida no meio técnico é “canário” (*canary*), em referência aos passarinhos usados por mineiros de carvão para detectar monóxido de carbono (CO) em níveis letais. O gás mata os passarinhos muito mais rápido do que mataria os trabalhadores, dando a eles tempo de fugir. Na pilha, o “canário” é um número aleatório colocado em um ponto estratégico. Um ataque por estouro de pilha certamente alteraria esse número, evento que seria detectado pelo SSP antes que o código malicioso do invasor pudesse ser executado. O SSP então envia uma mensagem ao registro de eventos do sistema (syslog) e interrompe a execução do programa atacado.

Esse tipo de verificação é bastante dispendioso em termos de memória e tempo de CPU, especialmente quando funções muito pequenas são usadas. O SSP tenta detectar funções que possam ser vulneráveis a estouros de pilha e adiciona as rotinas de verificação apenas a elas. Como o mecanismo de detecção não é perfeito, é bem provável que o SSP deixe de injetar as rotinas em funções que delas precisem ou as inclua em lugares desnecessários. Às vezes, ambas as coisas.

O SSP também pode ser usado para compilar o kernel. Sempre é uma boa idéia incluir uma camada de proteção em volta do núcleo do sistema. Quanto ao desempenho, uma surpresa: as otimizações do SSP não deixam o kernel perceptivelmente mais pesado!

Existem técnicas mais elaboradas para fazer com que programas escritos na linguagem C sejam mais seguros.

Comparação: PaX versus outros patches.

A melhor coisa a respeito do *paxtest* é o fato de que foi possível usá-lo para testar a proteção de memória proporcionada pelos *outros* patches. Baixe o paxtest do site oficial do PaX [4] e compile-o em seu sistema. Se estiver usando Adamantix, basta um *apt-get install paxtest*. Disparar o paxtest contra um kernel “remendado” com o *OpenWall* mostra que apenas a pilha é protegida contra execução de código – coisa que, aliás, é o esperado. Em outras palavras, isso significa que quase não há proteção.

Quando o paxtest é rodado contra o *exec-shield*, obtemos diferentes resultados, dependendo da versão do paxtest em uso. Algumas falhas em versões antigas do paxtest fizeram com que as pessoas acreditassem que o *exec-shield* oferece uma proteção maior do que realmente o faz.

Na realidade, a proteção que o *exec-shield* oferece é realmente impressionante – só que para menos. Será interessante acompanhar a agitação dos desenvolvedores de exploits e ver quão rápido se adaptarão às fraquezas do *exec-shield*. Seria divertido se alguém portasse o paxtest para o OpenBSD. Não acredito que o i386 W^X – tão festejado pela comunidade OpenBSD – mostre resultados convincentes.

Uma delas é a famosa verificação de limites (*bounds checking*). Essa verificação significa que o acesso a *todo e qualquer dado* é testado. Pelo lado bom, isso traz mais segurança para qualquer programa. A desvantagem: o programa fica **pesado**. A velocidade é comparável à do Java – que, por acaso, faz verificação de limites por padrão. Essa queda brutal no desempenho é algo com que, na vida real, poucas pessoas querem (ou podem) conviver.

Aleatorização

Um recurso encontrado no PaX e em outros patches de proteção de memória é o chamado ASLR – "Address Space Layout Randomization", algo como *disposição aleatória do espaço de endereços*. Com o ASLR, diferentes partes do programa são gravadas em locais diferentes da memória. A posição de cada porção do programa na RAM muda a cada vez que ele é executado.

Isso não é um mecanismo de proteção, pois não há pontos de controle. Entretanto, dificulta bastante uma tentativa de exploração de uma falha conhecida, uma vez que o atacante nunca pode ter certeza da exata localização das coisas na memória. Os diversos patches de proteção de memória diferem na quantidade de entropia usada. Normalmente, quanto mais melhor, pois faz com que ataques de força bruta (*brute force*) sejam gradativamente mais difíceis. Atualmente, o PaX oferece a maior aleatoriedade dentre todos os patches de proteção de memória para Linux. Aliás, melhor até do que a do OpenBSD.

Compatibilidade

Não é necessário recompilar todos os seus programas para que eles funcionem num kernel com o PaX. A maioria deles rodará bem sem qualquer modificação. Já as bibliotecas podem causar algum estresse. Por experiência própria, sei que o Debian Woody possui algumas "pegadinhas" nessa área. Poucas bibliotecas são afetadas, mas algumas delas são extremamente importantes – a zlib é um belo exemplo. Conta-se que versões antigas do Red Hat possuem muitos problemas de compatibilidade com bibliotecas, embora eu nada tenha ouvido sobre as versões mais novas. Fiz testes preliminares com o Debian Sarge e, exceto pelo servidor gráfico (*XFree86*), tudo funcionou sem problemas com o kernel com PaX.

A maioria dos programas funciona sem nenhum problema com o PaX, mesmo que esteja configurado para ser bastante restritivo – como é o caso do Adamantix, por exemplo. Das várias centenas de pacotes já adaptados para uso no Adamantix, apenas uns poucos causam problemas com o PaX. Em muitos deles, o conserto foi bem fácil. As alterações mais comuns são de *flags* de compilação, especialmente para bibliotecas. Em alguns casos precisamos usar código em C em vez de assembly (por exemplo, na zlib e no gnupg). Em outros, algumas linhas de código tiveram que ser reescritas para se adequar ao método de trabalho do compilador.

Alguns programas – poucos, na verdade – não puderam ser adaptados. Todos eles precisam, por definição, criar código executável diretamente na memória. Um bom exemplo é o ambiente de Java da Sun, o SUN Java Runtime Environment (JRE).

Quando isso acontece, é possível usar o comando *chpax* para definir exceções. As configurações do *chpax* são gravadas no interior do próprio executável. Quando o executável é invocado, o PaX detecta essas configurações e desativa algumas das verificações. Um software problemático pode, então, ser colocado em funcionamento sem que seja necessário mudar uma única linha de código.

O duro teste da paz

Quando o PaX foi incluído no kernel do Adamantix e eu passei a recompilar os programas para o ASLR do sistema, tudo funcionou tão maravilhosamente bem que eu comecei a duvidar da honestidade dos desenvolvedores. Antes de começar, eu já estava preparado para encarar um grande número de programas deixando de funcionar. Nada disso pareceu ocorrer. Como isso era possível? Uma das alternativas era a de que o PaX estava inoperante, ou estava realmente ativado mas não fazia nada de nada. Em vez de especular, decidi encontrar alguma prova de (não) funcionamento. Para isso, desenvolvi o *paxtest*.

O *paxtest* é uma pequena coleção de programas de teste. Cada um deles verifica um aspecto funcional do PaX. Um teste escreve código de máquina em uma variável do tipo *string* e tenta executá-lo. Um PaX funcionando bem detectaria a tentativa e mataria o processo imediatamente. Outros criavam diversas situações vulneráveis, cada uma relativa a um aspecto que o PaX deveria proteger, e tentavam explorá-las.

Há também testes que verificam a profundidade de aleatorização do ASLR. Juntos, todos os programas nos dão uma idéia do nível de proteção oferecido pelo PaX. Quanto mais testes informam "killed" (ou seja, processo interrompido), melhor. Usando os dados obtidos com o *paxtest*, pude comprovar que o PaX estava se dando muito bem com o kernel do Adamantix. Melhor que isso: a ausência de programas indicava que o PaX estava funcionando muito melhor do que o esperado.

Conclusão

A proteção da memória usada pelos processos é assunto da mais alta importância, mas até agora não conseguimos encontrar nada que nos proteja contra as três classes de ataques ao mesmo tempo. O PaX, desde seu surgimento, nunca foi superado por outro patch de proteção de memória. Com o auxílio de controle de acesso (ACLs) e outros mecanismos auxiliares, o PaX consegue garantir proteção perfeita contra ataques de Classe (1).

Outros mecanismos, como o SSP, são necessários para evitar alguns ataques nas Classes (2) e (3). O custo de implementação é relativamente baixo. Qualquer distribuição de Linux que se importa com aspectos de segurança tem a obrigação de planejar a inclusão desse tipo de proteção. ■

INFORMAÇÕES

[1] *http://www.rsbac.org/*

[2] *http://www.adamantix.org/*

[3] *http://old.lwn.net/1998/0806/a/linus-noexec.html*

[4] *http://pageexec.virtualave.net/*

[5] *http://archives.neohapsis.com/archives/openbsd/2003-04/1362.html*

[6] *http://people.redhat.com/mingo/exec-shield/*

[7] *http://www.research.ibm.com/trl/projects/security/ssp/*