Programação segura para administradores de sistema – Parte 2

Podando pela raiz

Quase todo script ou programa abre e processa arquivos. Este tutorial mostra onde se encontra o perigo e dá ao administrador ou desenvolvedor os meios para neutralizar as falhas já na raiz.

Por Dominik Vogt

Há diversas armadilhas à espreita do administrador em operações aparentemente inofensivas, como a criação, abertura ou fechamento de arquivos. A parte 1 desta série [1] trabalhou com arquivos abertos durante a inicialização e no encerramento do programa. Esta etapa indica como dominar outras operações no sistema de arquivos em programas escritos em C, C++ e Shell Scripts.

A raiz do mal

O Unix organiza os sistemas de arquivos, dispositivos e outros recursos sob o diretório raiz, onde vários processos podem acessá-los simultaneamente. Acessos simultâneos podem levar às famosas *race conditions* (condições de disputa), que são o vetor principal de muitas falhas de segurança (veja o quadro **Links simbólicos, condições de disputa e seus efeitos**).

Quando um programa ou script opera com arquivos, podem aparecer algumas surpresas. Entre duas ações individuais um outro processo pode criar um novo arquivo ou diretório, excluí-lo ou movê-lo. O administrador monta ou desmonta um sistema de arquivos (`mount()`, `umount()`) ou modifica o proprietário de um arquivo; um usuário muda um *link* simbólico, cria ou exclui um *hardlink* ou altera as permissões de um arquivo. Todas essas ações podem apresentar problemas.

São vulneráveis, principalmente, os programas e scripts que operam negligentemente com nomes de arquivos. Nenhum dos sistemas usuais de arquivos garante que um nome se refira sempre ao mesmo *inode* em dois momentos diferentes. Dessa maneira o programador corre três riscos:

- Ele usa um outro arquivo que não o intencionado;
- As características do arquivo são alteradas inesperadamente;
- Um usuário ou um processo recebe mais (ou menos) permissões de acesso do que o desejado.

O desenvolvedor não pode evitar completamente essas panes, mesmo com as mais rígidas medidas de segurança, mas deve estar sempre preparado para elas. Nas suas tentativas de defesa ele pode partir do princípio de que o root não teria porque inserir intencionalmente essas situações perigosas.

Listagem 1: Pontos fracos dos links simbólicos

```
01 #!/bin/sh
02 for i in `find . -name "*.txt"`; do
03         tr "A-Z" "a-z" < $i > /tmp/inseguro.tmp
04         mv /tmp/inseguro.tmp $i
05 done
```

www.istpix.com

Exemplo: Administrador de email

No sistema do administrador, Antônio, operam separadamente dois servidores de emails, um para o grupo vermelho e um para o azul. Cada integrante do grupo possui permissões restritas de administrador nas caixas de email de seus colegas de grupo. Vários programas especializados com o bit SUID ligado distribuem a eles as autorizações necessárias. O diretório de emails `/var/mailequipe` pode ter a seguinte forma:

```
$ cd /var/mailequipe; ls -ld * .
drwxrwsrwt 2 root root 4096 … ./
-rw-rw-- 1 bernardo azul        11  … bernardo
-rw-rw-- 1 renata   vermelho     0  … renata
-rw-rw-- 1 rogerio  vermelho   738  … rogerio
```

O programa `esvaziar-mbox` na **listagem 2a** esvazia a caixa de entrada de emails (arquivo `mbox`) de um colega de grupo. O controle de acesso nas linhas **14** a **23** (`stat()`) assegura que o grupo do arquivo `mbox` seja o mesmo que o grupo que chama o programa. Então `fopen()` abre o arquivo, exclui (linha **25**) quaisquer conteúdos que possam existir ou, se necessário, cria o arquivo novamente. Por fim o programa de Antônio aplica, caso seja criado um novo arquivo, grupos e permissões adequadas (linhas **27** a **35**).

Para esvaziar a caixa de entrada do seu colega Rogério (grupo vermelho), Renata digita `esvaziar-mbox rogerio` – isso sem saber que Antônio deixou escapar cinco falhas graves de segurança; e Renata cai em todas. Melhor seria usar o programa na **listagem 2b**.

Armadilha 1: Direitos

Quando um programa *SUID root* abre um arquivo por ordem de um usuário, é bom que ele verifique as permissões de acesso ao arquivo desejado. Em geral o desenvolvedor deixa essa tarefa preferencialmente para o sistema operacional, pois ele conhece as características dos diversos sistemas de arquivos. Em `esvaziar-mbox`, Antônio confia no grupo. Mesmo se Rogério proteger sua caixa de entrada com `chmod 600`, o programa autorizará o acesso a Renata.

O programa de Antônio comete um outro erro: lê primeiro as informações do arquivo com `stat()` (linha **15**), verifica-as na linha **20** e então abre o

Links simbólicos, condições de disputa e seus efeitos

As condições para um ataque de sucesso por links simbólicos (*symlinks*) são: um programa sem proteção e um diretório com acesso global de escrita, no qual o programa pode ler ou escrever arquivos. Na maioria dos casos o `/tmp`, o `~/tmp`, o `/var/tmp` ou o `/var/spool/mail` são adequados para isso. Se combinarmos o link simbólico com uma condição de disputa (*race condition*) aparece a situação mostrada na **figura 1**: o programa vulnerável verifica se o destino está correto (lado esquerdo), mas quando, logo em seguida, for executar a ação (à direita), a situação já pode ter se modificado.

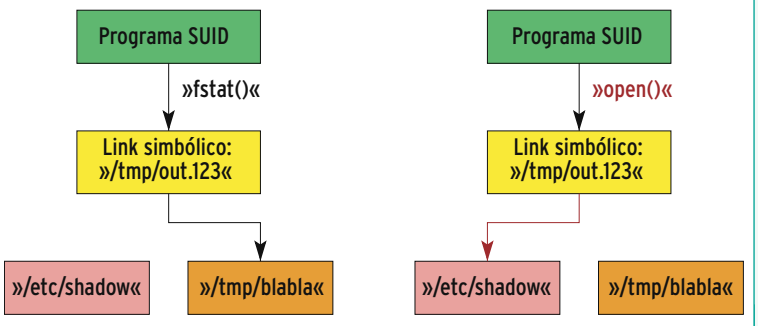

Figura 1: Em ataques por links simbólicos o agressor cria um symlink para um de seus arquivos. Com a função `fstat()` o programa descobre que o usuário pode escrever no arquivo. O usuário abre então o arquivo para escrita; nesse meio tempo, o agressor já terá escondido o symlink.

Freqüentemente a situação é ainda pior. O simples script na **listagem 1** facilita a ação do agressor através de um nome de arquivo constante, `inseguro.tmp`. Ele pode sobrescrever, excluir ou manipular seu conteúdo com arquivos estranhos. Com esse script o agressor tenta transformar todas as letras maiúsculas, de todos os arquivos `.txt` no diretório atual, em minúsculas.

Mas que bom que ninguém sabe disso

Antes de o script entrar em funcionamento, o sabotador cria um link simbólico com o nome de `/tmp/inseguro.tmp` apontando para qualquer arquivo que queira excluir e espera até que o administrador chame o script. Na linha **3** o script sobregrava o conteúdo do arquivo que o agressor indicou e depois move o symlink. Dessa forma, os dois arquivos serão danificados, o apontado pelo link simbólico e o arquivo cujo conteúdo o usuário queria modificar.

Nesse discreto programa de quatro linhas estão escondidos outros dois pontos de ataque. Quando um usuário executa o script várias vezes paralelamente para processar ao mesmo tempo múltiplos diretórios, os processos sobregravam seus arquivos temporários, danificando-os. Além disso, a variável `$i` não está entre aspas. Isso pode trazer resultados inesperados quando o nome do arquivo possuir caracteres especiais.

Cuidado com experimentos próprios: o script danifica tanto o arquivo de entrada quanto o de saída (**figura 2**). No caderno especial da Linux Magazine Internacional de abril de 2004 [2] "deixamos escapar" um script de uso rotineiro que estava vulnerável a essa falha. Ele caiu exatamente na armadilha descrita.

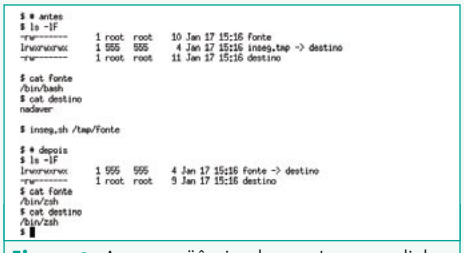

Figura 2: As conseqüências de um ataque por link simbólico. O script da **listagem 1** sobrescreveu o arquivo `/tmp/destino` (que nem estava envolvido no comando, foi "injetado" pelo agressor) e substituiu o arquivo original `/tmp/fonte` por um link simbólico que aponta para `/tmp/destino`.

Listagem 2a: Esvaziando a caixa de entrada sem segurança

```
01 #include <errno.h>
02 #include <limits.h>
03 #include <stdio.h>
04 #include <sys/types.h>
05 #include <sys/stat.h>
06 #include <unistd.h>
07
08 int main(int argc, char **argv) {
09   char fn[PATH_MAX];
10   struct stat buf;
11   int rc;
12
13   snprintf(fn, PATH_MAX, "/var/mailequipe/%s", argv[1]);
14   /* Verificando permissões de acesso */
15   rc = stat(fn, &buf);
16   if (rc != 0 && rc != ENOENT) {
17     fprintf(stderr, "Erro\n");
18     return 1;
19   }
20   if (rc == 0 && buf.st_gid != getgid()) {
21     fprintf(stderr, "Acesso negado\n");
22     return 1;
23   }
24   /* Arquivo clonado ou truncado */
25   if (fopen(fn, "w+") == NULL)
26     return 1;
27   /* Configurando proprietário e direitos */
28   if (chown(fn, buf.st_uid, getgid()) != 0) {
29     remove(fn);
30     return 1;
31   }
32   if (chmod(fn, 0066) != 0) {
33     remove(fn);
34     return 1;
35   }
36
37   return 0;
38 }
```

arquivo (linha **25**). Entre o `stat()` e o `fopen()` talvez Rogério mude para o grupo azul (**figura 3**); os dados do comando stat então não serão mais válidos.

Solução para os Shell Scripts: O desenvolvedor escreve um pequeno programa em C que verifica as permissões ou evita completamente esses testes. Melhor ainda é não usar scripts Shell para tarefas SUID Root.

Solução para programas em C e C++: O programa mostrado na **listagem 2b** devolve primeiro as permissões de root (veja a linha **18**) e então abre o arquivo com a chamada `open()`, sem sobrecrevê-lo-lo (linha **20**). Só quando isso funcionar o usuário poderá acessar o arquivo. Um problema permanece, pois Rogério ainda poderá mudar de grupo depois de abrir o arquivo. Nesse caso resta apenas a opção de ferramentas pesadas como o *Posix Mandatory Locks*.

Um desenvolvedor responsável sempre evita a seqüência de funções `stat()`, `lstat()` e `access()` e, em vez disso, usa `open()` e `fstat()`.

Listagem 2b: Esvaziando a caixa de entrada com segurança

```
01 #define _GNU_SOURCE
02 #include <fcntl.h>
03 #include <limits.h>
04 #include <stdio.h>
05 #include <sys/types.h>
06 #include <sys/stat.h>
07 #include <unistd.h>
08
09 int main(int argc, char **argv) {
10   char fn[PATH_MAX];
11   uid_t alte_euid;
12   int rc;
13   int fd;
14
15   snprintf(fn, PATH_MAX, "/var/mailequipe/%s", argv[1]);
16   /* Tornando-se o usuário */
17   alte_euid = geteuid();
18   seteuid(getuid());
19   /* Abrindo o arquivo */
20   fd = open(fn, O_NOFOLLOW);
21   if (fd >= 0) {
22     /* Se existir, permitir acesso e truncar */
23     rc = ftruncate(fd, 0);
24     close(fd);
25     return (rc == 0);
26   }
27   /* Criando novo arquivo */
28   fd = open(fn, O_NOFOLLOW | O_CREAT | O_EXCL, 0660);
29   if (fd < 0)
30     return 1;
31   close(fd);
32   /* Definir grupo como root */
33   seteuid(alte_euid);
34   if (fchown(fd, getuid(), getgid()) != 0) {
35     unlink(fn);
36     return 1;
37   }
38
39   return 0;
40 }
```

Armadilha 2: Abrir arquivos

As funções da família *Open* servem como porta de entrada principal dos ataques por links simbólicos. A função `fopen()`, principalmente, não permite que o programador detecte que o alvo é um link simbólico. Mas o perigo não está apenas no nome do arquivo. Se um agressor tiver permissão de escrever em um diretório, ele poderá criar um link simbólico, mover arquivos existentes ou mesmo, deus me livre, excluir árvores de diretório inteiras. A armadilha 4 descreve como isso pode ser verificado.

Solução para Shell Scripts: O administrador sempre deve manter os diretórios seguros. Se logo no começo do script o administrador executar o comando `set -C`, o Shell não sobrescreverá nenhum arquivo com o redirecionamento de saída `>`. Uma ajuda bem-vinda: o pouco conhecido shell *Zsh* (e apenas ele) (veja mais em [3]) se recusa terminantemente a seguir links simbólicos com o comando `cd -s diretório`.

Solução para programas em C e C++: O importante é escolher cuidadosamente o modo de acesso aos arquivos. A **listagem 2a** (veja a linha **25**) faz isso corretamente usando o parâmetro `w+`. Com a função `fopen()` não há proteção contra ataques por links simbólicos. O desenvolvedor pode evitar os ataques usando a função `open()` com a opção `O_NOFOLLOW` (veja a **listagem 2b**, linha **28**). Infelizmente a opção `O_NOFOLLOW` só existe em sistemas GNU se o parâmetro `_GNU_SOURCE` estiver definido (linha **1**). Quem necessitar mesmo assim de um ponteiro para arquivos pode encontrar alguma ajuda na função `fdopen()`, como abaixo:

```
int fd = open("arquivo", O_RDWR);
FILE *f = fdopen(fd, "r+");
```

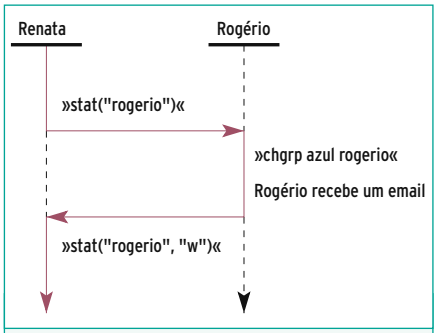

Figura 3: Renata chama `esvaziar-mbox Rogério`. Logo após a chamada à função `stat()`, Rogério muda para o grupo azul e recebe um email. Renata já recebeu a permissão de acesso e exclui a caixa de entrada de Rogério com o novo email dentro. Ela não poderia ter permissão de acesso a essa caixa.

A maioria das funções que operam com caminhos ou com ponteiros para arquivos também existe em versões que trabalham com descritores de arquivos. Alguns exemplos são `stat()` e `fstat()`, `read()` e `fread()` ou `trunca-`

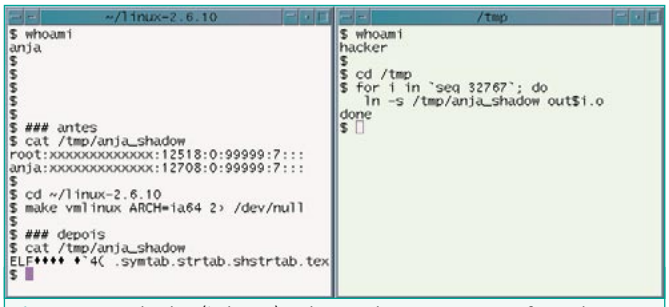

Figura 4: Um hacker (à direita) utiliza um bug nos arquivos-fonte do Linux para um ataque por link simbólico. Quando um usuário compilar o kernel ele sobrescreverá sem querer o arquivo de destino. Felizmente esse ataque não funciona com o `gcc` quando o root compila o arquivo.

`te()` e `ftruncate()` (veja a **listagem 2b**, linha **23**). Os nomes das funções são bastante confusos: às vezes uma função cujo nome começa com `f` trabalha com descritores de arquivos ou nomes, ao contrário de sua "irmã" sem o f; outras vezes, o que acontece é exatamente o oposto.

Armadilha 3: Criar arquivos

Além do mencionado até agora, o desenvolvedor deve criar um arquivo o quanto antes com as permissões corretas. Se ele deixar para ajustá-las tarde demais o agressor pode já ter aberto o arquivo **[1]**. O programa `esvaziar-mbox` na **listagem 2a** comete exatamente esse erro (linhas **25, 28, 32**). **Solução para Shell Scripts:**

```
umask 077
set -C
: > "Arquivo"
set +C
chmod modo arquivo
```

Solução para programas em C e C++: Novamente o `open()` tenta entrar em ação. As opções `O_CREAT` e `O_EXCL` cuidam para que a operação Open só crie o arquivo se ele ainda não existir. O terceiro argumento dá os direitos iniciais (**listagem 2b**, linha **28**). Importante: se o programador escolher a variante *Open* com apenas dois argumentos o arquivo receberá permissões ao acaso! A função `fchown()` substitui a `chown()` (linha **34**).

Listagem 3: cwdsafe.c

```
01 #include <fileguardian_headers.h>
02
03 int main(void) {
04   return !!fileg_check_cwd_safety();
05 }
```

Armadilha 4: Diretório seguro

Arquivos e diretórios estarão protegidos de ataques externos quando o diretório atual e todos os diretórios acima dele só concederem permissão de escrita para o root ou para o proprietário do arquivo. As permissões dos arquivos em si protegem apenas incompletamente, já que um agressor com permissão de escrita pode mudar o nome de um arquivo no diretório e criar um novo arquivo com o antigo nome.

Desde a primeira parte desta série **[1]**, a biblioteca *Gate Guardian* **[4]** continua crescendo e contém, a partir da versão 0.9.3, uma biblioteca compartilhada adicional e também funções para o manuseio seguro de arquivos. Uma outra solução está em **[5]**.

Solução para Shell Scripts: O programa `cwdsafe` na **listagem 3** verifica o diretório de trabalho e utiliza para isso a *libgateguardian*. O administrador instala essa biblioteca como sempre, com a sequência mágica: `./configure && make && make install` e então compila o programa `cwdsafe.c` com o comando `gcc -o cwdsafe cwdsafe.c -lgateguardian`. Se ele copiar o programa para mais um ponto do PATH (o local recomendado é: `/usr/local/bin`), qualquer usuário poderá usar `cwdsafe` em seus scripts:

```
umask 077
cd diretório
cwdsafe || exit 1
echo Olá > arquivo
```

Solução para programas em C e C++: O procedimento é praticamente idêntico ao usado no Shell. Como de rotina no Gate Guardian, o programador pode incluir e chamar o código completo simplesmente através de diretivas *include*, sem necessidade de utilizar uma biblioteca:

```
#include "fileguardian.c"
[…]
if (fileg_check_cwd_safety() != 0)
  exit(1);
```

Listagem 4: Bug nos fontes Linux

```
01 #!/bin/sh
02 dir=$(dirname $0)
03 CC=$1
04 OBJDUMP=$2
05 tmp=${TMPDIR:-/tmp}
06 out=$tmp/out$$.o
07 $CC -c $dir/check-gas-asm.S -o $out
08 […]
```

Além disto, o Gate Guardian oferece outras funções que aceitam qualquer diretório: o `fileg_check_dir_safety_with_mode()` permite a indicação de permissões proibidas de acesso, o `fileg_safe_opendir()` abre o diretório verificado e mostra um ponteiro de diretórios e o `fileg_safe_opendir_with_mode()` combina os dois.

Armadilha 5: Arquivos temporários

As dificuldades descritas acima passam facilmente despercebidas pelos programadores quando gravam dados temporários por curto espaço de tempo. Como medida de segurança o diretório `/tmp` implanta normalmente o *Sticky Bit* (numa tradução livre, *bit grudento*). Mesmo que ofereça permissões de escrita a todos, só o proprietário pode excluir um arquivo, mas qualquer um pode criar novos arquivos. Apenas quem toma cuidado ao abrir um arquivo (para ler ou escrever) não precisa se preocupar.

O código-fonte da versão atual do Linux, 2.6.11, contêm alguns scripts para a arquitetura IA64 que escrevem sem os devidos cuidados. A **listagem 4** mostra trechos do script `linux-2.6.10/arch/ia64/scripts/check-gas`. As linhas **5** e **6** colocam o nome `/tmp/outPID.o` na variável `out` para um arquivo temporário. O shell substitui `$$` através da ID do processo. A chamada do compilador na linha **7** cria então o arquivo. A facilidade com que um agressor consegue sobrescrever arquivos com links simbólicos é comprovada pela **figura 4**.

Outros programadores também cometem o mesmo erro freqüentemente. No sistema Debian Woody (3.0r4) do autor o comando a seguir:

```
find / -type f -perm +111 -print0 | xargs -0 grep '/tmp.*$$'
```

encontra uns 85 scripts, em cerca de 550 pacotes diferentes, que contêm em uma linha construções como `/tmp/out$$`. Por experiência pessoal, posso dizer que um em cada três desses scripts são vulneráveis a ataques por links simbólicos.

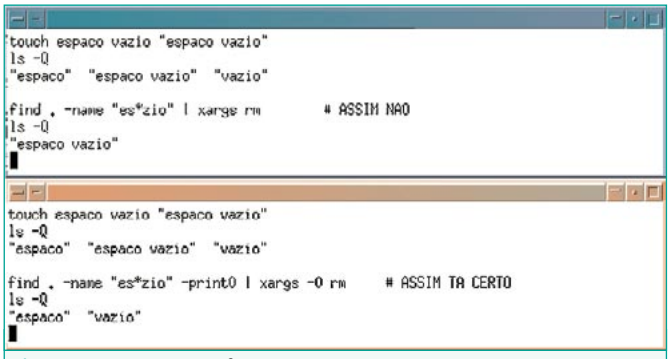

Figura 5: Acima, um inofensivo `find ... | xargs ...` encontra os arquivos errados. Abaixo, com as opções específicas do GNU `-print0` e `-0`, a pipeline mantém os espaços em branco nos nomes de arquivos.

Solução para Shell Scripts: O comando *mktemp* gera um novo arquivo temporário, cujo nome é calculado parcialmente pelo mktemp sem cair nas *race conditions* (como mostrado na **listagem 5a**). Em uma emergência, basta um nome qualquer definido pelo programador, pois o script desliga a sobrescrita de um arquivo já existente com `set -C`. Além disso, através do sinal *Trap*, a **listagem 5a** cuida para que o script exclua o arquivo temporário no caso de uma parada anormal do programa, como quando a execução é abortada com ***Ctrl+C***.

Solução para programas em C e C++: A função `tmpnam()` gera um nome de arquivo temporário que ainda esteja livre. A armadilha 3 descreve como criar então um arquivo com segurança. Muitos manuais de dicas aconselham a não usar a função `tmpnam()`, e em seu lugar chamar `mkstemp()`. Essa função, porém, não cria um caminho de volta e por isso o programador não pode verificar a segurança do diretório onde o arquivo está sendo criado.

Com o Gate Guardian são formados arquivos temporários seguros com as funções `fileg_safe_tmpfile()` ou `fileg_safe_tmpfile_with_name()` (**listagem 5b**). Elas criam um arquivo temporário seguro ou retornam o valor `-1`, se isso não for possível.

Listagem 5a: Arquivos temporários no shell

```
01 TMPF=""
02 TMPDIR=${TMPDIR-/tmp}
03
04 clean_up () {
05   test ! x"$TMPF" = x && rm -f "$TMPF"
06 }
07
08 trap clean_up 0 1 2 13 15
09 umask 077
10 TMPF=`mktemp -q "$TMPDIR/foo.XXXXXXXX"` || exit 1
```

Listagem 5b: Arquivos temporários com o Gate Guardian

```
01 #include <fileguardian_headers.h>
02
03 int main(void) {
04   char name[L_tmpnam];
05   FILE *f = fileg_safe_tmpfile_with_name(name);
06   return 0;
07 }
```

Armadilha 6: Excluir arquivos

Para apagar um arquivo o programador pode escolher entre `unlink()` e `remove()`. As duas funções usam o nome do arquivo como argumento. Se um outro processo já houver excluído o arquivo e criado um novo com o mesmo nome, ambas as funções removerão o arquivo errado. Não existe proteção contra isso.

Às vezes o programador quer ter a certeza que realmente está excluindo os dados gravados. Um simples `unlink()` não basta, porque o agressor pode criar um *hardlink* para o arquivo. Qualquer usuário que tenha permissão de leitura para o diretório em que se encontra o arquivo pode fazer isso. Assim, o agressor não transgride as permissões de arquivo, mas conserva os dados até que tenha entrado no sistema.

Solução: O programa Wipe [6] sobregrava várias vezes o arquivo com padrões predefinidos ou aleatórios e então o exclui.

Armadilha 7: Nomes de arquivos em scripts

Provavelmente nenhum autor de script conta com espaços em branco em nomes de arquivos. Uma breve procura com `grep "#!/bin/sh" *` em `/bin` ou `/usr/bin` já mostra resultados: se criarmos um arquivo qualquer chamado `"dois nomes"`, com as palavras "dois" e "nomes" separadas por espaços, muitos scripts do sistema não saberão o que fazer com eles. Alguns exemplos: *colormake*, *ps2ps*, *ps2ascii* e *xdvi*. Ao invés de processar o arquivo desejado ele tenta abrir dois arquivos, o `dois` e o `nomes`.

Solução: Um administrador cuidadoso sempre coloca variáveis Shell entre aspas simples ou duplas (**listagem 6**). Note a substituição das aspas invertidas (**listagem 1**, linha **2**) por `$(comando)`.

O melhor é usar comandos que aceitem caracteres especiais. Na **listagem 6**, linha **7**, o `**/*.txt` substitui o comando `find`. O Zsh já conhece há muito tempo o asterisco duplo e versões recentes do Bash também o conhecem. Ele é usado como "contêiner" para representar todos os arquivos em todos os subdiretórios.

Seqüência de comando

A variável do shell `$*` também é freqüentemente usada de maneira incorreta. O seguinte script não é confiável:

```
#!/bin/sh
grep $*
```

Vamos supor que o script seja chamado de *procura*. O comando `procura xuxubeleza "dois nomes"` procura a palavra *xuxubeleza*, nos arquivos `dois` e `nomes`, quando o correto seria procurar no arquivo `dois nomes` (ou seja, levar em considera;cão o espaço em branco). Mesmo com aspas o `grep "$*"` falha. Dessa forma, o que acaba acontecendo é que o grep procura pela seqüência de caracteres `xuxubeleza dois nomes` na entrada padrão.

Solução: Coloque o `$*` entre aspas duplas, ou seja, `grep "$*"`. O Shell memoriza os limites da palavra nessa variável independentemente de espaços em branco.

Listagem 6: Listagem 1 melhorada

```
01 #!/bin/sh
02 IFS=" \t\n"
03 PATH="/bin:/usr/bin"
04 umask 077
05 TDIR=${TDIR-/tmp}
06 TFILE=`mktemp -q "$TDIR/seguro.XXXXXXXX"` || exit 1
07 for i in **/*.txt; do
08   tr "A-Z" "a-z" < "$i" > "$TFILE"
09   mv "$TFILE" "$i"
10 done
```

Dupla diabólica

Os mui usados comandos `find` e `xargs` tornam-se muito perigosos se utilizados, por exemplo, desta maneira:

```
find / -name "*~" | xargs rm
```

No pipe a informação sobre espaços em branco, quebras de linha e tabuladores é perdida. Imagine que você queira apagar um arquivo chamado `/etc/passwd bak~` (com um espaço entre as palavras). O que ocorre é que o comando `rm` exclui tanto o arquivo `/etc/passwd` como o arquivo `./bak~`. A **figura 5** ilustra o resultado.

Solução: O GNU find conhece a opção `-print0`. Dessa maneira, ele separa a saída com zeros. No outro lado do pipe a opção `-0` do *xargs* cuida para que os programas sejam compatíveis. Mas cuidado! isso só funciona com o comando xargs. Nenhuma das ferramentas Unix sabe o que fazer com o parâmetro `-print0` .

Trabalho detalhado

No trabalho com arquivos, o demônio está nos detalhes. Este tutorial trata desse extenso assunto apenas superficialmente. Consulte as referências abaixo para saber mais sobre o assunto, e fique de olho nas futuras edições da Linux Magazine. Até lá! ■

Informações

[1] Dominik Vogt, *Poluição do ambiente – Programação segura para administradores de sistema, Parte 1*: Linux Magazine Brasil, décima edição, pág. 68

[2] Peer Heinlein, *Simples – Dois exemplos de uso do bash na rotina do administrador*: caderno especial da Linux Magazine Alemã número 04/04, pág. 18.

[3] Zsh: zsh.sunsite.dk

[4] Dominik Vogt, projeto Gate Guardian: sourceforge.net/projects/gateguardian

[5] John Viega e Matt Messier, *Secure Programming Cookbook*, editora O'Reilly, ISBN 0-596-00394-3: www.secureprogramming.com

[6] Wipe: abaababa.ouvaton.org/wipe/