Muitos administradores, sem intenção, são os causadores de seus próprios problemas de segurança ao escrever ou modificar scripts e programas. Este tutorial explica os erros mais freqüentes e perigosos e mostra a melhor maneira de agir.

Dominik Vogt

credits ou free images

Programação segura para administradores de sistema – Parte 1

Poluição do ambiente

A maioria dos administradores já ouviu falar dos famosos estouros de buffer e *cross site scripting* ou de ataques envolvendo a formatação de cadeias de caracteres (*string format*). Existem vários livros com explicações sobre a causa destas falhas e como evitá-las, como [1] e [2]. Ao contrário do que se poderia imaginar, esse conhecimento não é importante apenas para os desenvolvedores de software. Os administradores, também, devem saber como escrever (sob pressão) scripts seguros e como evitar falhas comuns ao modificar programas.

Mesmo quando a solução passa longe da teoria que aprendemos na faculdade ou a eficácia deixa a desejar, ninguém deve arriscar o surgimento de falhas de segurança [3]. Muitas vezes a regra é: desativar o programa e fechar as lacunas. Nesse caso o administrador deverá avaliar o que é realmente necessário e se uma atualização de correção (*patch*) resolverá mesmo o problema. Este tutorial tem a intenção de transmitir os conhecimentos específicos necessários.

Conheça o seu ambiente

O ambiente em que um programa ou um script funciona é uma estrutura muito complexa. Ele contém muitas variáveis de ambiente: o diretório de trabalho, o diretório raiz, direitos, limites de recur-

Quadro 1: Auditoria do ambiente

A pequena e útil ferramenta *env_audit* verifica objetivamente o ambiente de um processo. Depois de baixar e descompactar (de [4]) o arquivo `env_audit-2.0.tar.gz`, basta digitar o comando `make` para compilá-lo. Se a compilação for abortada, como aconteceu no sistema do autor, provavelmente o arquivo `sys/capability.h` não foi encontrado. Com algumas alterações, o env_audit funcionará sem as funções presentes nesse arquivo. Para isso, edite o arquivo `env_audit.c` e insira, antes da linha 48, o comando `#undef`:

```
47 #undef _POSIX_CAP
48 #ifdef _POSIX_CAP
49 #include </sys/capability.h>
50 #endif
```

Além disso, exclua a biblioteca `-lcap` da linha 22 do arquivo `Makefile`

Aplicação

No subdiretório `examples` do código-fonte do *env_audit* encontram-se vários exemplos de aplicações. A cada chamada o env_audit cria um novo arquivo `/tmp/env_auditXXXX.log`, onde XXXX é um número seqüencial de 4 dígitos começando em 0000. Alguns exemplos ainda usam como referência o nome `env_audit.log`; nesse caso, o usuário deve fazer manualmente a correção.

O env-audit pode, por exemplo, verificar o ambiente em que os scripts CGI rodam no servidor de web. Para isso, basta copiar o programa `env_audit` para o diretório `cgi-bin` e atribuir permissões de execução (`chmod 555 env_audit`). Abra seu navegador de Internet e acesse o endereço **http://localhost/cgi-bin/env_audit**. O script fornecerá as informações desejadas em cerca de dez segundos. Para que isso funcione, a ferramenta avalia a variável de ambiente `HTTP_ACCEPT`. Quando ela estiver presente, o env_audit escreverá os resultados na saída padrão e não no arquivo de *log*; desse modo os dados chegarão ao navegador.

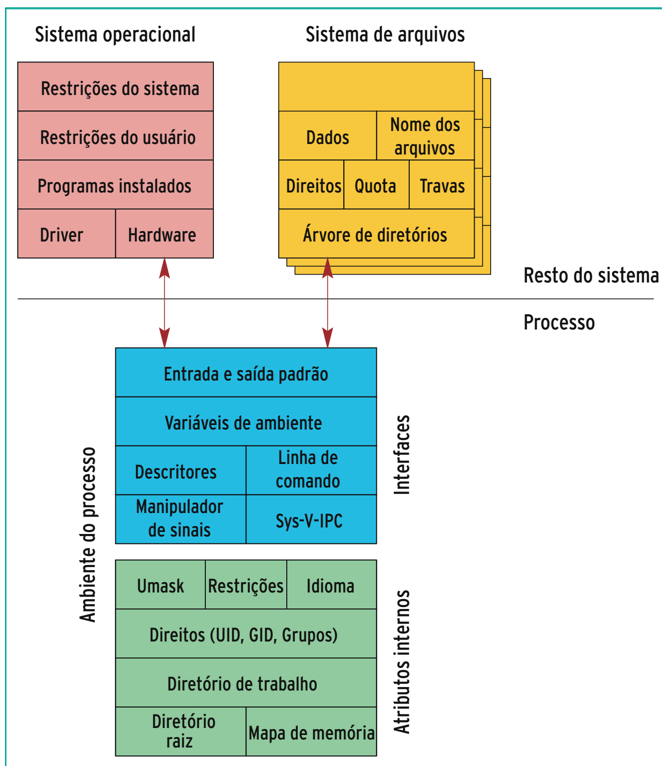

Figura 1: O ambiente de um processo registra características como umask e UID, assim como variáveis do ambiente e as definições de entradas e saídas. O restante do sistema também tem influência no processo

sos, máscara de criação de arquivos (como as definidas pelo comando *umask*), descritores de arquivos, manipuladores de sinais e muito mais. Há que se lembrar ainda do perigo de um invasor conseguir manipular – remotamente – os processos que estão em execução. Cuidado especial deve ser tomado quando os programas rodam com bit SUID ativado – com ele, um usuário normal pode rodar um programa com os direitos do administrador da máquina, o usuário *root*.

A abrangência dos ataques usando esses processos privilegiados é extremamente grande porque os sabotadores podem manipular o programa já no início. Na inicialização, o processo já herda o ambiente de seu processo pai, sendo influenciado também pelo shell e pelo kernel. Muitas características têm origem indireta no restante do sistema, como por exemplo a quantidade de RAM disponível ou a estrutura do sistema de arquivos (ver **figura 1**).

Respeito ao ambiente

A pequena ferramenta Linux *env_audit* [4] mostra muitas características de um processo de forma legível aos humanos. A **tabela 1** explica a instalação e o uso da ferramenta. A **listagem 1** mostra o relatório do env_audit executado como script CGI na instalação padrão de um Apache-1.3.26 no Debian Woody 3.0r2.

Listagem 1: resultados simplificados do env_audit

```
003  Process ID: 10369
004  Parent Process ID: 10353
005  User ID: 33 - www-data
006  Group ID: 33 - www-data
007  Effective User ID: 33 - www-data
008  Effective Group ID: 33 - www-data
009  Supplemental Groups:  www-data
010  Process Group ID: 10299
011  Session ID: 10299
012  Parent Session ID: 10299
013  Current Working Dir: /var/www/cgi-bin
014  Umask: 22
015  Process Priority: 5
018  Environmental Variables
028  $PATH=/bin:/usr/bin:/sbin:/usr/sbin
046  WARNING $IFS undefined
050  Resource Limits
051  Name            Current     Max
052  RLIMIT_CORE     (infinity)  (infinity)
053  RLIMIT_CPU      (infinity)  (infinity)
054  RLIMIT_DATA     (infinity)  (infinity)
055  RLIMIT_FSIZE    (infinity)  (infinity)
056  RLIMIT_MEMLOCK  (infinity)  (infinity)
057  RLIMIT_NOFILE         1024        1024
058  RLIMIT_OFILE          1024        1024
059  RLIMIT_NPROC          6144        6144
060  RLIMIT_RSS      (infinity)  (infinity)
061  RLIMIT_STACK       8388608  (infinity)
062  RLIMIT_AS       (infinity)  (infinity)
065  Open file descriptor: 0
066  User ID of File Owner: www-data
067  Group ID of File Owner: www-data
068  Descriptor is stdin.
069  No controlling terminal
070  File type: fifo, inode - 10051, device - 7
071  The descriptor is: pipe:[10051]
072  File descriptor mode is: read only
085  Open file descriptor: 2
086  User ID of File Owner: root
087  Group ID of File Owner: root
088  Descriptor is stderr.
089  No controlling terminal
090  File type: regular file, inode - 222552, device - 769
091  The descriptor is: /var/log/apache/error.log
092  File's actual permissions: 644
093  File descriptor mode is: write only, append
106  Open file descriptor: 4
107  User ID of File Owner: root
108  Group ID of File Owner: root
109  WARNING - Descriptor is leaked from parent.
110  File type: regular file, inode - 333258, device - 769
111  The descriptor is: /tmp/session_mm_apache0.sem
112  File's actual permissions: 600
113  File descriptor mode is: read and write
```

As linhas de **3** a **12** informam sobre o gerenciamento e as permissões do processo. No Debian testado, os scripts CGI funcionam com as permissões do usuário *www-data* e do grupo de mesmo nome. Isto pode trazer conseqüências desagradáveis quando o servidor hospedar vários clientes que têm permissão para instalar os scripts CGI. Quando isso acontece, os CGIs de todos os clientes rodam usando o mesmo usuário.

Numa situação dessas, pode acontecer dos processos de um cliente enviarem sinais de controle aos processos de **outro** cliente – por exemplo, o comando `kill -9`, que interrompe a execução do processo contra o qual foi disparado (no jargão popular, "mata" o processo). Outras possíveis formas de se explorar essa falha seria examinar – através do sistema de arquivos `/proc` – a memória usada pelos outros CGIs e, assim, obter dados úteis sobre eles – por exemplo: se um dos sites que o servidor hospeda tem um site de comércio eletrônico, o dono de outro site pode ver os números de cartão de crédito e o volume de vendas do primeiro. Ainda aproveitando o fato de todo mundo ser *www-data*, um usuário malicioso pode manipular os arquivos temporários de outros usuários. Como se sabe, arquivos temporários podem conter informações valiosas.

Listagem 2: Opa, esse não é o sudo não!

```
01 /* O bit SUID (Set UID Root) deve estar ativo.
02  * As senhas estão em /etc/password
03  * no formato "usuário:senha".
04  */
05 #include <stdio.h>
06 #include <stdlib.h>
07 #include <unistd.h>
08
09 int main(void) {
10   char pwd[9], busca[99], linha[99];
11   uid_t uid;
12   FILE *df;
13
14   /* Abrir Banco de dados */
15   df = fopen("/etc/password", "r+");
16   /* Por garantia: Desliga o buffer de dados */
17   setvbuf(df, 0, _IONBF, 0);
18
19   /* Leitura de UID e Senha */
20   printf("Por favor, digite o User ID: ");
21   fscanf(stdin, "%d", &uid);
22   printf("Por favor, digite a senha: ");
23   fscanf(stdin, "%8s", pwd);
24
25   /* Busca entrada no banco de dados */
26   sprintf(busca, "%d:%s", uid, pwd);
27   while (1) {
28     /* Busca por linha */
29     if (fscanf(df, "%98s", linha) != 1)
30       exit(1); /* Fim do arquivo */
31     if (strcmp(linha, busca) == 0)
32       break; /* encontrado */
33   }
34
35   setreuid(uid, uid); /* Devolve ao root */
36   execl("/bin/Skript", 0); /* inicia */
37   return 255;
38 }
```

De acordo com a linha **14**, o *umask* abre mão de restringir as permissões de outros usuários sobre novos arquivos. Os scripts que manuseiam dados sensíveis devem levar isso em conta e ajustar o umask manualmente para um valor seguro. A partir da linha **18** o env_audit mostra as variáveis do ambiente. A variável `PATH` parece ser adequada porque o Apache usa um valor padrão. Muitas vezes isso é importante para evitar a inserção de um `.` (sim, um mísero caracter de ponto, que representa o diretório corrente) nela por um invasor ou usuário malicioso.

Além de `PATH` , a variável `IFS` (*Inter-Field Separator*, ou Separador entre Campos) também é extremamente perigosa. Ela contém os símbolos que o shell usa para separar as palavras na linha de comando. Normalmente são espaços, tabulações e quebras de linha. Na linha **46** o env_audit alerta que o IFS ainda não está ajustado.

Autocompromisso voluntário

As linhas **50** a **62** mostram, em forma de tabela, os limites de utilização dos recursos do sistema, parecido com o mostrado por `ulimit -a`, porém de forma mais detalhada. A divisão em limites atuais (*Current*) e limites máximos (*Max*) é mais esclarecedora. A primeira divisão pode ser modificada pelo usuário comum; por exemplo, a memória disponível na pilha `RLIMIT_STACK`, usando-se a função `setrlimit()`. Os valores máximos podem ser alterados apenas pelo root. Nesse caso é melhor o administrador limitar o uso de recursos.

No final, a partir da linha **65** encontram-se informações sobre os descritores de arquivos abertos. O descritor 0 (entrada padrão, linhas **65** a **72**) é um *fifo* (ou *pipe*) para o processo pai, assim como também a saída padrão (não copiada). Bastante interessante é o descritor 2 (saída de erro padrão, *stderr*, linhas **85** a **93**). O script CGI tem permissão para escrever (linha **93**) no arquivo de log `/var/log/apache/error.log` (linha **91**), mesmo este arquivo pertencendo ao root (linhas **86** e **87**). Isso significa que o script pode falsificar qualquer mensagem lá escrita, ultrapassar sua quota do disco e assim inundar o sistema de arquivos com dados (causando uma negação de serviço), o que certamente não é o desejado pelo administrador.

No descritor 4 (linha **106**), o env_audit alerta sobre uma falha no descritor do processo pai (linha **109**). Mas apenas com essas mensagens não é possível definir se essa foi a intenção dos desenvolvedores do Apache. Novamente o programa CGI poderá contornar as restrições de

Listagem 3: começando certo

```
001 [...]
002 #define NIFS "IFS= \\n"
003 #define NPATH "PATH="_PATH_STDPATH
004
005 static void disable_core_dumps (void) {
006   struct rlimit r = { 0, 0 };
007   if (setrlimit(RLIMIT_CORE, &r) != 0)
008     exit(1);
009 }
010
011 static void set_minimal_env (void) {
012   extern char **environ;
013   static char **ne = NULL;
014
015   ne = malloc(3 * sizeof(char *) + sizeof(NIFS) +
sizeof(NPATH));
016   /* Configura variáveis de ambiente */
017   ne[0] = (char *)&(ne[3]);
018   memcpy(ne[0], NIFS, sizeof(NIFS));
019   ne[1] = ne[0] + sizeof(NIFS);
020   memcpy(ne[1], NPATH, sizeof(NPATH));
021   ne[2] = NULL;
022   /* Substitui ambiente antigo */
023   environ = ne;
024 }
025
026 static void close_descriptors (void) {
027   int nd;
028
029   /* Válido apenas no Linux, senão use getdtablesize() */
030   if ((nd = sysconf(_SC_OPEN_MAX)) <> 0)
031     exit(1);
032   while (--nd > 2)
033     close(nd);
034 }
035
036 static void open_stdfiles (void) {
037   struct stat buf;
038   FILE *f[3];
039   char *m[3] = { "rb", "wb", "wb" };
040   int i;
041
042   f[0] = stdin;
043   f[1] = stdout;
044   f[2] = stderr;
045   for (i = 0; i <> 3; i++) {
046     if (fstat(i, &buf) == 0)
047       continue;
048     if (errno != EBADF)
049       exit(1);
050     if (freopen(_PATH_DEVNULL, m[i], f[i]) != f[i])
051       exit(1);
052   }
053 }
054
055 static void reset_sighandlers (void) {
056   int i;
057
058   for (i = 1; i <>= NSIG; i++)
059     signal(i, SIG_DFL);
060 }
061
062 /* Versão mais segura na segunda parte! */
063 static void change_workdir (char *path) {
064   if (chdir(path) != 0)
065     exit(1);
066 }
067
068 static void safe_chroot (char *path) {
069   /* Primeiro fecha todos os descritores: open_stdfiles()
*/
070   if (chroot(path) != 0)
071     exit (1);
072   if (chdir("/") != 0)
073     exit(1);
074   /* Dá poderes de root: set_credentials() */
075 }
076
077 static void set_credentials (uid_t uid, gid_t gid) {
078   /* Somente root pode chamar setgroups */
079   if (geteuid() == 0 && setgroups(1, &gid) != 0)
080     exit(1);
081   /* Somente no Linux: */
082   if (setregid(gid, gid) != 0)
083     exit(1);
084   if (setreuid(uid, uid) != 0)
085     exit(1);
086 }
087
088 int main(void) {
089   /* A seqüência é importante! */
090   disable_core_dumps();
091   reset_sighandlers();
092   umask(0077);
093   set_minimal_env();
094   close_descriptors();
095   open_stdfiles();
096   safe_chroot("/chroot");
097   change_workdir("/path/workdir");
098   set_credentials(geteuid(), getegid());
099
100   /* ... */
101   return 0;
102 }
```

sua quota de disco, desta vez usando como artifício sua permissão de escrita no diretório `/tmp`.

A lei de Murphy

É impressionante o que está por trás deste exemplo. A **listagem 2** mostra a rapidez com que surgem as falhas de segurança. O programa, com o nome de `sudo-clone`, foi escrito para servir como ferramenta de ajuste do UID root. Ele implementa um mecanismo próprio para simular o comportamento do *sudo*. O programa cai em algumas das armadilhas descritas a seguir. O parágrafo no final deste artigo mostra que armadilhas são essas. Uma pequena observação: o invasor poderá ler ou danificar o arquivo de senhas `/etc/passwd`, usado pelo sudo-clone. No final do artigo também descrevemos como isso pode ser evitado.

A proteção contra as nove armadilhas que mostraremos a seguir – e que estão ligadas ao ambiente do processo – pode ser conseguida se aplicarmos medidas simples quando programamos em C e C++ ou mesmo em *Shell Scripts*. Quem necessitar de técnicas mais sofisticadas irá encontrá-las em [1] e [5]. As linguagens de script tem ambientes mais difíceis de organizar do que os programas em C. O programa *Super* encontrado em [6] ajuda. Ele oferece uma "jaula" na qual os scripts funcionam – sem risco – com permissões do superusuário, root. O *sudo* [7] é parecido com o Super e muito mais popular, porém trabalha com menos rigidez.

Listagem 4: Excluindo variáveis com Zsh

```
01 function zsh_clear_env () {
02   for V in `set +`; do case "$V" in
03     '!'|'$'|'*'|' '|'?'|'-'|'#'|[0-9]|V);;
04     **) typeset +r "$V"; unset "$V";;
05   esac; done
06   unset V
07   emulate zsh
08   export IFS=" \t\n"
09   export PATH="/bin:/usr/bin:/sbin:/usr/sbin"
10 }
```

Cópias da memória

Armadilha 1: em caso de "capotamento" (o programa é abortado por algum motivo), os processos deixam uma cópia da memória no diretório de trabalho (arquivo `core`). Para a depuração, isso pode ser bastante útil, mas também cria problemas. Arquivos core às vezes contêm informações confidenciais, como senhas que não devem ser gravadas no disco rígido em texto puro. Mesmo que o kernel do Linux grave os arquivos core com as permissões 600 (aliás, é o que acontece), não custa nada ajustar rigorosamente o umask do sistema.

Além disso, um arquivo core pode alcançar facilmente um tamanho bastante grande em bytes. Em casos extremos, toda a memória real e virtual, incluindo o arquivo de troca (*swap*), será gravada no disco rígido – dependendo do sistema e do que estiver rodando no momento, o arquivo poderá ter vários gigabytes (isso mesmo, GBytes!) de tamanho. Um invasor local ou externo poderá provocar esta falha para iniciar uma negação de serviço (DoS).

Programas Linux com o bit SUID ou SGID ativados não podem gerar, em hipótese alguma, um arquivo core. O administrador do Linux (e de outras variantes Unix também) pode limitar o tamanho máximo dos *core dumps*. O comando `ulimit -a | grep core` mostra os limites válidos em blocos de 512 Byte:

```
core file size  (blocks, -c) ➘
unlimited
```

A solução para shell scripts: quando um script iniciar um processo ele herdará o ambiente do shell em que foi chamado. A linha `ulimit -c 0` no início do script ou num arquivo de configuração do shell bloqueia todos os arquivos core gerados a partir desse shell.

A solução para programas em C e o C++: a função `setrlimit()` tem no Linux a mesma função que o `ulimit` no shell (veja a **listagem 3**, função `disable_core_dumps()`).

umask

Armadilha 2: normalmente o sistema permite que todos os usuários possam ler e escrever em arquivos novos (666 em notação octal). Um usuário local, por exemplo, poderá então ler e escrever arquivos temporários e talvez até descobrir senhas. Há pouco tempo o OpenOffice foi notícia por causa do ajuste inadequado do umask [8]. O umask limita essas permissões. Muitas distribuições têm como ajuste inicial (válido para todos os usuários) `umask 022` – o que resulta em arquivos com as permissões 644. Assim, qualquer pessoa pode ver os novos arquivos, mas apenas o proprietário pode escrever neles.

A solução para shell scripts: Adicione o comando `umask 077` já no inicio do script para salvar o dia.

A solução para programas em C e o C++: pouco depois do início do programa, o programador chama `umask(0077)` (números representados em notação octal tem um 0 no início). Atenção: não é suficiente alterar as permissões de acesso logo depois de criar o arquivo com `chmod()`. Um invasor pode ter aberto o arquivo nesse meio tempo (a chamada *Race Condition*). Esses ataques são facilmente automatizados e surpreendentemente eficazes.

Variáveis de ambiente

Armadilha 3: variáveis de ambiente são de grande valia. Com elas o usuário configura seu shell e outros programas ou evita o uso de programas opcionais e traba-

lho de digitação. Pela mesma razão elas tornam-se perigosas quando um invasor manipula seu conteúdo. São muitas as variáveis que representam um risco: `IFS`, `PATH`, `TZ`, muitas variáveis que começam com `LC_` e `LD_` e outras.

O perigo surge quando um programa confia cegamente nas variáveis de ambiente (talvez até sem saber). Por exemplo, imagine que o invasor manipulou a variável `PATH` e incluiu o diretório `/tmp` com precedência maior que qualquer outro. Depois, ele coloca um programa malicioso qualquer em `/tmp` e o renomeia para o mesmo nome de um outro programa já existente e de uso comum – `rm`, por exemplo. O usuário root, ignorante de tudo isso, pode tentar usar o `rm`, mas como `PATH` está manipulada, vai executar `/tmp/rm` ao invés de `/bin/rm`. Raramente fica claro qual variável está sendo usada, em que momento e de que forma, e o exemplo acima demonstra isso de maneira irrefutável.

O ambiente do processo salva as variáveis como seqüências de caracteres terminadas em zero na forma `nome=valor`. Com a função `execve()` o invasor pode ajustar qualquer coisa; por exemplo, deixando o nome da variável vazio, criar registros sem o `=` ou múltiplas versões das mesmas variáveis.

A solução para shell scripts: o *Bash* e outros shells atuais não oferecem uma solução confiável para o administrador, apenas o *Zsh* (*Z Shell*) **[9]** (veja a **listagem 4**). O programador depende do administrador de sistemas para disponibilizar um ambiente controlado para a execução de seu software. Para isso o programa Super é indicado, já que ele exclui todas as variáveis de ambiente estranhas e ajusta outras importantes com valores padrão e reconhecidamente sadios. O menos indicado é o sudo, porque ele apenas faz a filtragem de variáveis reconhecidamente perigosas, mas não reage às variáveis desconhecidas.

O comando `env -i` funciona bem, pois “limpa” o ambiente sem dar permissões ao programa. O comando `exec -c` *`programa`* também funciona bem. Nos dois casos o script iniciará um novo programa, aproveitando o ambiente limpo. Mas talvez nesse momento o invasor já tenha atingido seu objetivo. Para um mínimo de segurança, o script ajusta as variáveis `IFS` e `PATH` no início da execução:

```
#!/bin/sh
IFS=" \t\n"
PATH="/bin:/usr/bin:/sbin:/usr/sbin"
export IFS; export PATH
```

A solução para programas em C e C++: o programa substitui o ambiente por uma estrutura que contém apenas valores padrão. Ele apenas copia para o novo ambiente e verifica variáveis de que ele mesmo precisa. O código para exclusão total é parecido com a função `set_minimal_env()` da **listagem 3**. ➟

Descritores abertos

Armadilha 4: o número de arquivos, de *sockets* e de outros descritores abertos é limitado. Por isso, quando estão abertos os descritores – gerados por um processo pai – são deixados de herança para os eventuais processos filhos. Um invasor pode desviar o fluxo de funcionamento de um programa executando-o com vários descritores já abertos. As possibilidades dependem muito de cada programa. Elas vão desde uso não autorizado do próprio programa e ataques de DoS até invasões sofisticadas em que o agressor consegue um shell com permissões de administrador – tudo é possível. Um processo filho também recebe todas as permissões do seu processo pai. Se, por exemplo, um programa abrir o arquivo `/etc/passwd` como usuário root e escrever nele, todo processo filho também terá permissão de escrita.

A solução para shell scripts: o sudo e o Super fecham todos os descritores de arquivos antes de chamar um comando. A chamada `exec` *n* `>` *arquivo* abrirá um arquivo para escrita com descritor "n" e `exec` *n* `> &-` o fechará. Para leitura o `>` deverá ser substituído por `<`.

A solução para programas em C e C++: o programa fecha os descritores já no início e depois de um `fork()` fechará todos os descritores que estiverem inesperadamente abertos (veja a **listagem 3**, função `close_descriptors()`). A função `sysconf()` com o argumento `_SC_OPEN_MAX` retorna o descritor mais alto possível no Linux (exclusivamente no Linux – outros sabores de Unix não oferecem essa possibilidade). Em outros sistemas o programador tem que contentar-se com `getdtablesize()` ou `OPEN_MAX`. Importante: `getrlimit()` não fornece o número procurado, apenas o limite para novos descritores.

A função `popen()` também chama internamente `fork()`. Como o programador não tem influência neste processo, é melhor não usar o `popen()`. A função `execve()` fecha os descritores apenas a pedido. Uma seqüência adequada é, logo depois do `open()`, o comando `fcntl (`*descritor*`, F_SETFD, F_CLOEXEC);`.

Entrada e saída padrão

Armadilha 5: na maioria dos casos os descritores de arquivos 0, 1 e 2 são programados com a entrada e saída padrão, exceto se eles forem fechadas previamente pelo solicitante. Então, quando um programa abrir um arquivo para escrita, relacionará a ele o primeiro descritor que estiver livre, por exemplo o 2 (saída de erro padrão). A partir daí todas as mensagens de erro do sistema serão registradas – automagicamente – nesse arquivo. Como muitas vezes os invasores podem manipular essas mensagens de erro, podem usar esse artifício para alterar o conteúdo de quaisquer arquivos a seu bel-prazer – arquivos que, em condições normais, não teriam permissão para alterar.

A solução para shell scripts: os shells não escolhem automaticamente descritores adequados. No *Bash* e no *Zsh* as linhas a seguir abrem os descritores padrão quando necessário e os ligam ao dispositivo `/dev/null`:

```
test -e /dev/fd/0 || exec < /dev/null
test -e /dev/fd/1 || exec 1>&/dev/null
test -e /dev/fd/2 || exec 2>&/dev/null
```

A solução para programas em C e C++: o desenvolvedor verifica se o `stdin`, `stdout` e `stderr` estão abertos e, se necessário, faz a conexão com o `/dev/null`. Na **listagem 3**, a função `open_stdfiles()` mostra como isso funciona.

Identificadores de usuários

Armadilha 6: sistemas Unix modernos (BSD 4.4, Posix) relacionam três identificadores de usuários (UID) a cada processo. O UID efetivo (EUID) fornece as permissões de acesso, o UID real (RUID) é o identificador do usuário que iniciou o programa. Internamente, o processo ainda possui um UID "de backup" (*Saved UID* ou SVUID). Em programas que ajustam o UID do usuário, o RUID e o EUID são diferentes. Em processos normais todos os UIDs são idênticos. O sistema permite, entre outras coisas, que um programa escolha qual dos três identificadores quer usar. Assim, um programa SUID une as permissões do usuário que chamou o programa com as permissões do proprietário do programa. O mesmo vale para os identificadores de grupos (RGID, EGID e SVGID).

Para evitar riscos desnecessários, cada programa deveria devolver os direitos adicionais o quanto antes. O programa `passwd` necessita de permissão de root apenas para escrever uma nova senha em `/etc/shadow`, mas não quando chama ou verifica uma senha do usuário. Um bom programador cuidará de desativar irreversivelmente esses privilégios o mais cedo possível.

A solução para shell scripts: a maioria dos shells não têm a função para o script mudar sua identificação de usuário. Nesse caso o programador deve separar a tarefa em duas partes. O script privilegiado conclui seu trabalho e chama um parceiro não-privilegiado com o Super ou o sudo; o inverso também é possível.

Um script *Zsh* pode relacionar valores diretamente para as variáveis UID e EUID. Mas em função das diferenças na base entre os vários sistemas operacionais, não se deve confiar apenas em um determinado procedimento.

A solução para programas em C e C++: desativar as permissões adicionais não é fácil quando o programa tem que ser portátil. O Linux e outros sistemas Posix usam a função `setreuid()`, que altera todos os três identificadores de uma só vez (veja a **listagem 3**, função `set_credentials()`). Mas em sistemas BSD o procedimento do `setreuid()` é diferente.

Figura 2: O pequeno programa em C no editor (à direita) serve como ajuda na fuga. Com ele o invasor (à esquerda) consegue fugir de uma jaula *chroot*

Processamento de sinais

Armadilha 7: os programas comunicam-se mais facilmente através de sinais. Quando um sinal (codificado em um valor de 5 bits) chega a um processo, o sistema operacional desencadeia uma ação. A maioria dos sinais pode ser ignorada pelo programa, se esse for o objetivo. De acordo com o caso, poderá ser mantida a reação padrão (terminar o programa, criar um *core dump*, etc) ou instalar um manipulador de sinais próprio.

O Linux distingue 31 sinais (veja a lista completa com o comando `man 7 signal`) que um processo recebe do sistema operacional, de um processo que pertence ao mesmo usuário ou ao root. Com o SVUID a situação fica ainda mais complexa (veja `man 2 kill`). O processo herda de seu pai o modo de reagir a um sinal. Se o processo pai tinha implementado um manipulador de sinais próprio, o processo filho retornará à reação padrão. A função `execve()`, porém, é um caso especial; veja a página de manual com: `man 2 execve`.

Utilizar os sinais de forma correta é um tanto complicado, porque durante o processamento podem surgir novas condições para os sinais. A função `signal()` também se comporta de forma diferente em variantes do Unix. Desenvolvedores e administradores pouco experientes não deveriam utilizar esta função. A probabilidade de fazer tudo da maneira certa é muito pouca. Para o invasor abrem-se três caminhos:

- O desenvolvedor de um programa com bit SUID ativado não sobregrava o UID real do solicitante. O invasor inicia então o programa e envia os sinais. O programa não esperava receber sinais de um usuário comum. O programador reage sobregravando o UID real com o UID efetivo o quanto antes (ver armadilha 6).
- O invasor solicita que o sistema operacional mande sinais que o programa não espera.

Listagem 5: sudo-clone com Gate Guardian

```
#include <stdio.h>
02  #include <stdlib.h>
03  #include <unistd.h>
04  #include "gateguardian.c"
05
06  int main(void) {
07    char pwd[9], busca[99], linha[99];
08    uid_t uid;
09    FILE *df;
10    gateg_cfg_t cfg;
11    int op;
12
13    /* Configuração padrão */
14    gateg_init_cfg(&cfg, GATEG_INIT_DFLT);
15    /* Isso só acontece na segunda chamada: */
16    op = GATEG_OP_SET | GATEG_OP_DELAY_1;
17    cfg.ops.priv_set = op;
18    cfg.ops.priv_drop = op;
19    cfg.ops.sighand_verify = op;
20    /* A primeira chamada resolve a maioria dos casos */
21    gateg_safe_init(&cfg);
22
23    /* Abre o banco de dados */
24    df = fopen("/etc/Passworte", "r+");
25    /* Por garantia, desliga os buffers */
26    setvbuf(df, 0, _IONBF, 0);
27
28    /* Lê UID e senha */
29    printf("Por faovr, informe o user ID: ");
30    fscanf(stdin, "%d", &uid);
31    printf("Por favor, informe a senha: ");
32    fscanf(stdin, "%8s", pwd);
33
34    /* Busca registro no banco de dados */
35    sprintf(busca, "%d:%s", uid, pwd);
36    while (1)
37    { /* Busca por linha */
38      if (fscanf(df, "%98s", linha) != 1)
39        exit(1); /* Fim do arquivo */
40      if (strcmp(linha, busca) == 0)
41        break; /* Encontrado */
42    }
43
44    /* Segunda chamada, substitui o setreuid */
45    cfg.ops.fds_close = op;
46    cfg.priv_uid = uid;
47    gateg_safe_init(&cfg);
48
49    execl("/bin/Skript", 0); /* Início */
50    return 255;
51  }
```

```
~
$ cat /tmp/ExploitSkript   # Angriffs-Skript:
echo
cat 0<&3

$ cat /bin/Skript   # Eigentliches Skript:
#!/bin/bash
echo -e "\n\nIch bin das bash Skript!\n"

$ # Mit BASH_ENV dem Programm unterjubeln
$ BASH_ENV=/tmp/ExploitSkript /bin/sudo-clone

Bitte User-Id eingeben : 1000
Bitte Passwort eingeben: geheim

1001:Anna
1002:qwert123
1003:W_2d@0x_
1004:password
1005:Waldi

Ich bin das bash Skript!

$
```

Figura 3: É fácil enganar o nosso clone do sudo da listagem 2. O invasor ajusta o `BASH_ENV` para dar passagem a seu próprio script.

- O invasor transmite ao processo filho uma forma de reação não esperada.

A solução para shell scripts: os próprios shells normalmente ajustam seus manipuladores de sinais. Mas não há garantia. Através do script, o programador não pode mudar muita coisa. O sudo e o Super ajustam o UID real correto e voltam ao processamento (e reação) padrão aos sinais.

A solução para programas em C e C++: o programa logo ignora o UID real (ver acima) e restaura os manipuladores de sinais (veja a **listagem 3**, função `reset_sighandlers()`).

Diretório de trabalho

Armadilha 8: em programas de ajuste do UID sugere-se definir um diretório de trabalho seguro; pode ser até o diretório pessoal do root (na maioria das distribuições, `/root`). Isso será importante se o diretório de trabalho estiver em uma mídia removível (ou remota, que pode estar montada ou não) ou num sistema de arquivos no qual as opções de montagem permitam acesso a outros usuários (por exemplo com `gid -option` em sistemas de arquivos FAT). Um core dump não é seguro num meio desse tipo.

Solução para shell scripts: `cd diretório || exit 1`

A solução para programas em C e C++ é o comando: `if (chdir("diretório") != 0) exit(1);`

Trocando a raiz

Armadilha 9: a função `chroot()` prende programas num diretório qualquer e "mente" a ele, dizendo que esse diretório é, na verdade, o sistema de arquivos raiz. Teoricamente isso evita o acesso ao sistema restante, já que o programa "enjaulado" tem poderes ilimitados dentro da jaula, mas permissões quase nulas fora dela. Mas a teoria, na prática, é diferente. A **figura 2** mostra como o invasor com permissões de root aproveita uma das várias brechas para sair da jaula *chroot* sem tomar conhecimento das restrições. Basta criar um arquivo e executá-lo. O usuário comum também sai sem problemas quando seu diretório de trabalho ou um outro diretório aberto está fora da jaula chroot.

A solução para shell scripts: usar o programa `chroot` para realizar tarefas de segurança não é recomendado. Como argumento ele recebe o comando que deve ser executado. Como isso pode ser executado apenas pelo root, o comando roda sempre com permissões de root. Consulte [9] para saber a maneira correta de fazer as coisas.

Solução para programas em C e C++: a função `safe_chroot()` da **listagem 3** mostra como o programador pode alcançar o objetivo em quatro passos.

Ataque dos clones

Voltando ao programa sudo-clone. Como foi dito no início, a **listagem 2** cai em pelo menos três das armadilhas que já mencionamos:

- O nosso clone do sudo abre `/etc/password` como root, mas não fecha o arquivo antes de chamar `execl`. O `/bin/script` funciona como usuário comum, mas herda o descritor de arquivo, incluindo as permissões de leitura e escrita.
- Supondo que o usuário solicitante tenha fechado a saída padrão e chame sudo-clone. A linha 15 atribuirá então o descritor 1 ao manipulador de arquivo `df`. O `printf()` na linha 20 vai querer abrir a entrada padrão, para isso, escreverá 1 no descritor (stdout). Porém, lá estará escondido o arquivo `/etc/password`.
- Nosso clone do sudo chama o script Bash `/bin/script`, mas não exclui previamente as variáveis de ambiente. Se um invasor ajustar a variável `BASH_ENV` em um outro script, o Bash vai executá-lo antes que `/bin/script` funcione.

Quem duvidar dessas brechas encontrará a prova da primeira e da terceira falhas na **figura 3**. O invasor usa uma variável de ambiente com a qual força o Bash a executar seu próprio script `/tmp/ExploitSkript` antes de executar o `/bin/script` solicitado. O script do exploit anexa o descritor de arquivo 3 à saída padrão (descritor 0). O descritor 3 herda o script do clone do sudo (a terceira lacuna) e recebe o arquivo da senha que chega corretamente no *stdout*.

Guardiões

Quem quiser proteger seu programa em C ou C++ destas armadilhas, sem a necessidade de cuidar de todos os detalhes,

```
~/src/fvwm
$ cat /tmp/ExploitSkript   # Angriffs-Skript:
echo
cat 0<&3

$ cat /bin/Skript   # Eigentliches Skript:
#!/bin/bash
echo -e "\n\nIch bin das bash Skript!\n"
cat 0<&3

$ # Mit BASH_ENV dem Programm unterjubeln
$ BASH_ENV=/tmp/ExploitSkript /bin/sudo-clone_gg

Bitte User-Id eingeben : 1000
Bitte Passwort eingeben: geheim


Ich bin das bash Skript!

/bin/Skript: 0: Bad file descriptor
$
```

Figura 4: O Gate Guardian protege o programa *sudo_clone* (**listagem 5**). O invasor não pode mais inserir seu script de exploit e o `/bin/Skript` também não herda mais o descritor de arquivo sabotado.

pode usar bibliotecas prontas. O autor deste artigo usou o *Gate Guardian* ([5], distribuído sob a licença BSD). Essa biblioteca em C conhece os truques e evita a entrada dos invasores, como um discreto guardião. Só consegue entrar no sistema quem é comprovadamente inofensivo ou por um acaso encontra uma entrada desprotegida.

Para usar o Gate Guardian no próprio projeto basta incluir o cabeçalho `src/gateguardian.c` em seu programa. A **listagem 5** mostra como o desenvolvedor fecha as lacunas da **listagem 2** com a ajuda do Gate Guardian. Em casos mais simples, ele apenas anexará quatro linhas ao código-fonte. A linha inclui `gateguardian.c` em seu programa. Isso não é muito elegante, mas evita qualquer modificação nos *Makefiles*.

A linha 10 cria uma estrutura de configuração que ajusta a função obrigatória `gateg_init_cfg()` na linha 14 com os ajustes iniciais padrão através do parâmetro `GATEG_INIT_DFLT`. Além disso existem os valores `GATEG_INIT_DFLT_PARANOID` com ajustes mais restritos e `GATEG_INIT_NOP` se o próprio programador fizer todos os ajustes. Na linha 21, em `gateg_safe_init()`, a configuração desejada é ativada. Para muitos programas isso já é o suficiente.

Quase a metade das 1600 linhas de código do Gate Guardian desvia das nove armadilhas descritas anteriormente. O código restante é responsável pelas interfaces. As medidas de segurança falham muitas vezes por falta de tempo do desenvolvedor. É para isso que o Gate Guardian Fail-Safe foi desenvolvido: ele usa os ajustes mais seguros sem medidas especiais. Isso também auxilia os administradores que precisam fazer pequenas modificações em seus servidores. Processos de instalação demorados e experiências com os Makefiles são proibidos. Basta copiar um arquivo e modificar algumas linhas num outro.

Os detalhes

Os comandos nas linhas **16** a **19** da **listagem 5** atribuem o valor `GATEG_OP_SET` para alguns elementos da subestrutura `ops`. Com isso cada elemento ativa uma função. Também existem `GATEG_OP_NOP` (sem ação) e `GATEG_OP_DFLT` (ação padrão, ou *default*). Há também o valor `GATE_OP_DELAY_1`. Ele faz com que a função só opere na segunda chamada de `gate_safe_init()`.

O programa deve fechar o máximo de lacunas o quanto antes e depois fechar o resto rapidamente. Ele desistirá de suas permissões de root quando não necessitar mais delas (`priv_set` e `priv_drop` ajustarão o valor de atraso). O processo `sighand_verify` verifica se um outro usuário ainda pode mandar sinais ao processo. Esse teste também só tem sentido quando o UID final do usuário já estiver definido.

As linhas **45** a **47** dão reforço. A operação `fds_close` fecha novamente todos os descritores abertos. Isso já aconteceu no primeiro passo, mas agora um novo descritor já está aberto (`df`). Então o programa reconhece o UID do usuário procurado e o insere no elemento `priv_uid`. A linha **47** faz o restante.

A **figura 4** prova que o trabalho compensa. Tanto um ataque através do `BASH_ENV` como o ataque por um descritor não têm efeito. Para assegurar que a lacuna está mesmo trancada o comando `cat 0 &3` agora também está escrito em `/bin/script`. Isso não foi necessário na situação da **figura 3**.

Chegando ao final com segurança

Não é fácil escrever programas seguros, mas a recompensa é a paz de espírito mais tarde. Talvez as explicações acima ajudem a fazer o mundo do Software Livre respirar um pouco mais aliviado. O projeto Gate Guardian também quer contribuir com isso. Na próxima edição, a segunda parte desta pequena série descreve o que pode acontecer de errado com arquivos se você não for atento o suficiente ao cuidar deles. ■

Informações

[1] John Viega e Matt Messier, *Secure Programming Cookbook for C an C++*, Editora O'Reilly, 2003: www.secureprogramming.com

[2] David A. Wheeler, *Secure Programming for Linux and Unix HOWTO*: www.dwheeler.com/secure-programs

[3] Dirk P., *Insel-Hüpfer - Sicherheitslücken bei Hosting-Providern*. Linux Magazine Alemãzz, edição 10/03, página 56

[4] Steve Grubb, autor do *env_audit*: www.web-insights.net/env_audit

[5] Página pessoal de Dominik Vogt, autor do programa *Gate Guardian*: sourceforge.net/projects/gateguardian

[6] Página oficial do Super: freshmeat.net/projects/super

[7] Página oficial do sudo: www.courtesan.com/sudo

[8] Mark Vogelsberger, *InSecurity News*, Linux Magazine 11/04, página 22. Mensagem original: www.securitytracker.com/alerts/2004/Sep/1011205.html

[9] Chroot-Login-HOWTO: www.tjw.org/chroot-login-HOWTO

[10] Zsh: zsh.sunsite.dk

[11] Código-fonte dos exemplos deste artigo: ftp://ftp.linux-magazin.de/pub/listings/magazin/2005/02/Sec-Prog

Sobre o autor

O matemático Dominik Vogt é desenvolvedor de software e administrador de sistemas há muitos anos. No momento trabalha como consultor autônomo de informática com foco em segurança de software. Nas suas horas livres gosta de fazer experiências com o gerenciador de janelas *Fvwm*.