Informationssicherheit

3. Schwachstellen

Prof. Dr. Christoph Skornia
christoph.skornia@oth-regensburg.de

Programmierfehler

Nicht korrekt überprüfte Eingaben:

- Jede interaktive Software verarbeitet Benutzereingaben
- Die Eingaben werden dabei von der Software interpretiert
- Tätigt der Benutzer erwartete Eingaben funktioniert alles reibungslos
- Annahme: Der Benutzer ist bösartig! Wird dieser erwartete Eingaben tätigen? Nein!
- Sichere Software muss also auch mit unerwarteten Eingaben umgehen können, das ist aber häufig nicht der Fall

Buffer Overflows

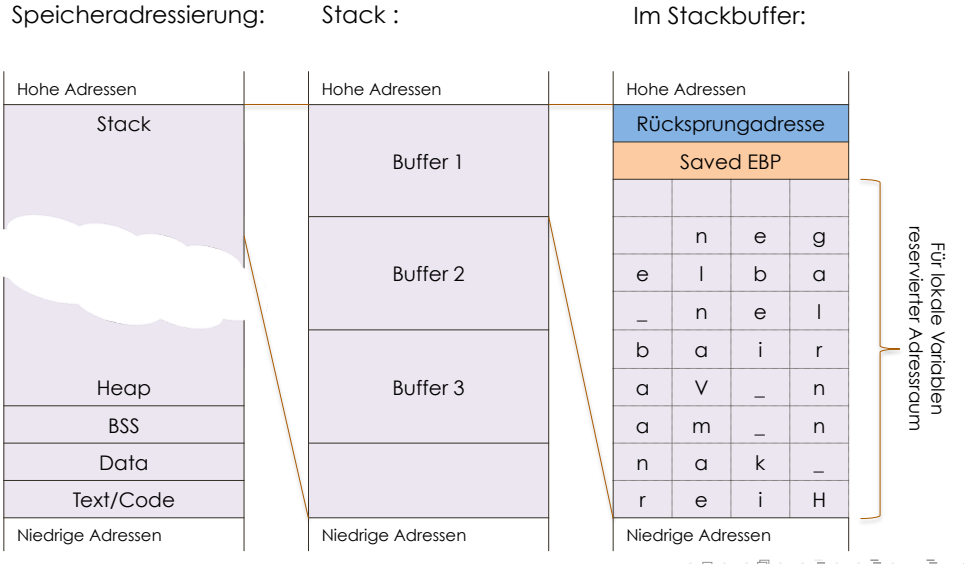
- Erste Berichte über Buffer-Overflow Angriffe 1973
- Erste größere Schadensereignisse regelmäßig seit 1988
- SANS No. 3 der gefährlichsten Software-Fehler

> „Buffer overflows are Mother Nature's little reminder of that law of physics that says: if you try to put more stuff into a container than it can hold, you're going to make a mess."

- Bekannte Angriffe:
  - Morris (1988, Unix finger)
  - Witty (2004, BlackICE)
  - Slammer (2003, MS-SQL)
  - Blaster (2003, MS-DCOM)
  - Twilight (2008, Wii)

Verbreitung:	Hoch
Folgen:	Code Execution, DoS
Vermeidung:	Einfach
Erkennung:	Einfach
Angriffshäufigkeit:	Hoch
Bekanntheit:	Hoch

Speicheradressierung in C und C++

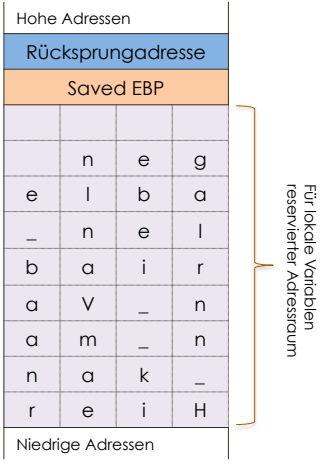


Speicheradressierung in C und C++:Stack Buffer Overflow

- Kann man damit mehr Daten in einen Speicherbereich schreiben, als dafür vorgesehen ist?

Speicheradressierung in C und C++:Stack Buffer Overflow

- Kann man damit mehr Daten in einen Speicherbereich schreiben, als dafür vorgesehen ist?
  - Ja, eine Reihe von Befehlen in C und C++ überprüfen standardmäßig nicht die Eingabelänge!
    Beispiele: strcpy(), strcat(), sprintf(), scanf(), gets().....
  - Ja, fehlerhafte Angaben über den benötigten Speicher (etwa „off-by-one" bei Schleifen) haben ebenso zur Folge, dass mehr als vorgesehen geschrieben wird.
- Welche Folgen hat das?

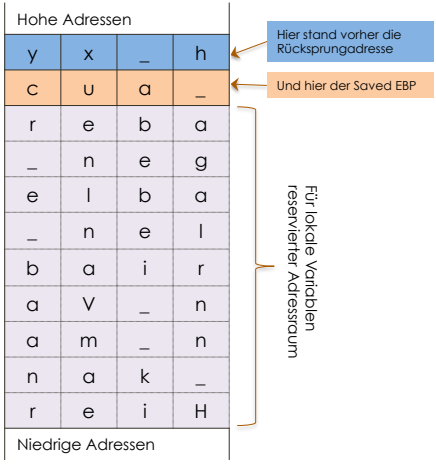


Speicheradressierung in C und C++:Stack Buffer Overflow

- Kann man damit mehr Daten in einen Speicherbereich schreiben, als dafür vorgesehen ist?
  - Ja, eine Reihe von Befehlen in C und C++ überprüfen standardmäßig nicht die Eingabelänge!
    Beispiele: strcpy(), strcat(), sprintf(), scanf(), gets().....
  - Ja, fehlerhafte Angaben über den benötigten Speicher (etwa „off-by-one" bei Schleifen) haben ebenso zur Folge, dass mehr als vorgesehen geschrieben wird.
- Welche Folgen hat das?

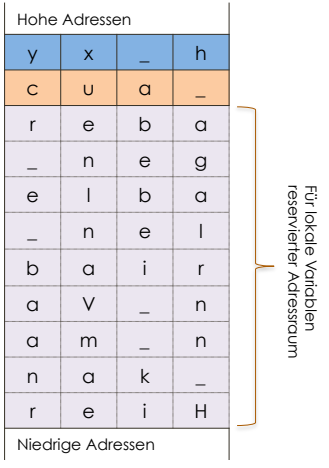
Entscheidend ist die Rücksprungadresse

- Steht dort ein ungültiger Wert, stürzt das Programm ab (erfolgreicher Angriff: DoS)
- Steht dort ein gültiger Wert, so wird springt das Programm zu diesem gültigen Wert und wird u.U. anders weiter geführt als vorgesehen



Angriff durch Buffer Overflow

Geht noch mehr?



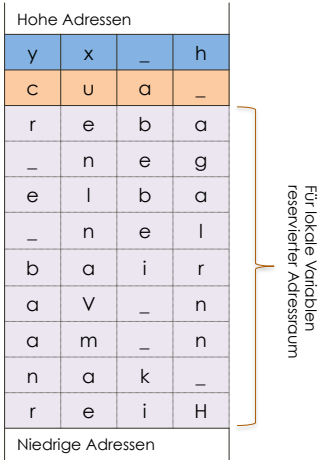
Angriff durch Buffer Overflow

Geht noch mehr? Ja! Gezieltes Überscheiben der Rücksprungadresse!

- Alternativen:

1. Die Rücksprungadresse zeigt auf andere Funktionen des ausgeführten Programms oder dessen Bibliotheken. (Bsp: ret2libc Angriffe)

   Konsequenz: Das Programm kann dem Benutzer Information geben die nicht für Ihn bestimmt sind. (Bsp: Datenbanken)



Angriff durch Buffer Overflow

Geht noch mehr? Ja! Gezieltes Überscheiben der Rücksprungadresse!

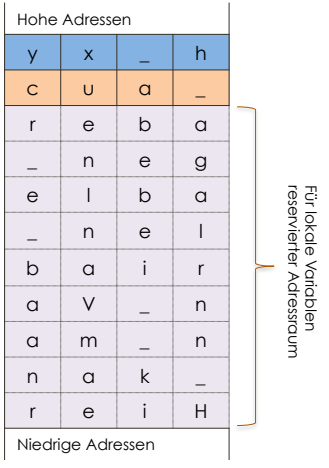
- Alternativen:

1. Die Rücksprungadresse zeigt auf andere Funktionen des ausgeführten Programms oder dessen Bibliotheken. (Bsp: ret2libc Angriffe)

   Konsequenz: Das Programm kann dem Benutzer Information geben die nicht für Ihn bestimmt sind. (Bsp: Datenbanken)

2. Die Rücksprungadresse zeigt zurück in den vorher überschriebenen Stack!

   Konsequenz: Wurde dort vorher Binärcode platziert, kommt dieser zur Ausführung. (Code Injection)

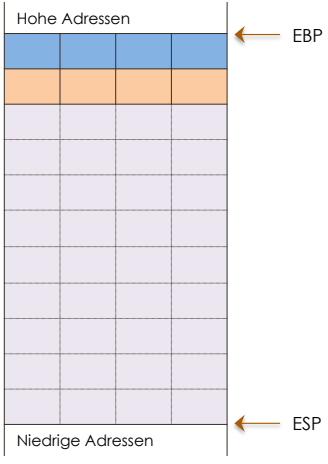


Angriff durch Buffer Overflow: Beispiel

Verwundbares Programm:

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv){
4      char buffer [512];
5      strcpy(buffer, argv[1]);
6      printf("You entered: %s\n",buffer);
7      return 0;
8  }
```

Vorgehensweise:



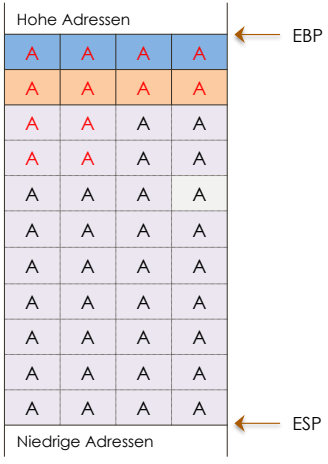
Angriff durch Buffer Overflow: Beispiel

Verwundbares Programm:

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv){
4      char buffer [512];
5      strcpy(buffer, argv[1]);
6      printf("You entered: %s\n",buffer);
7      return 0;
8  }
```

Vorgehensweise:

1. Schwachstelle identifizieren (z.B. durch gezieltes Herbeiführen von Abstürzen mit zu langen Eingaben)



Angriff durch Buffer Overflow: Beispiel

Verwundbares Programm:

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv){
4     char buffer [512];
5     strcpy(buffer, argv[1]);
6     printf("You␣entered:␣%s\n",buffer);
7     return 0;
8 }
```

Vorgehensweise:

1. Schwachstelle identifizieren (z.B. durch gezieltes Herbeiführen von Abstürzen mit zu langen Eingaben)
2. Schadhafte Eingabe konstruieren aus:
   1. NOPs (\x90) (bekommt der Prozessor diese in ausführbarem Code, springt er einfach eins weiter)

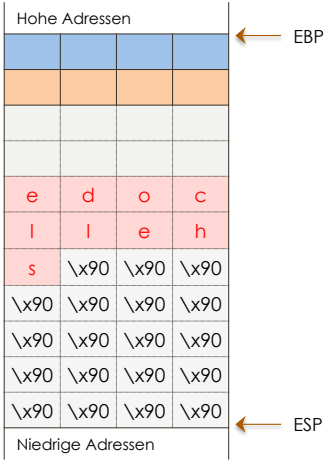
Angriff durch Buffer Overflow: Beispiel

Verwundbares Programm:

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv){
4      char buffer [512];
5      strcpy(buffer, argv[1]);
6      printf("You␣entered:␣%s\n",buffer);
7      return 0;
8  }
```

Vorgehensweise:

1. Schwachstelle identifizieren (z.B. durch gezieltes Herbeiführen von Abstürzen mit zu langen Eingaben)
2. Schadhafte Eingabe konstruieren aus:
   1. NOPs (\x90) (bekommt der Prozessor diese in ausführbarem Code, springt er einfach eins weiter)
   2. ausführbarer Code (z.B. /bin/sh)



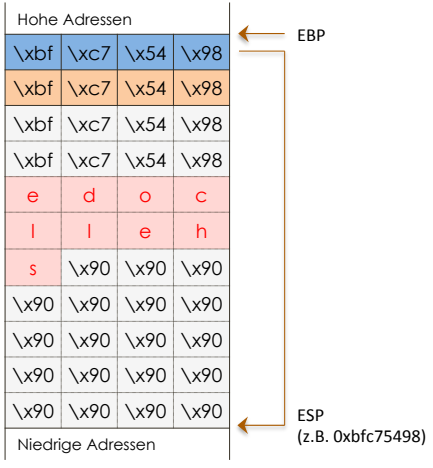
Angriff durch Buffer Overflow: Beispiel

Verwundbares Programm:

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv){
4      char buffer [512];
5      strcpy(buffer, argv[1]);
6      printf("You␣entered:␣%s\n",buffer);
7      return 0;
8  }
```

Vorgehensweise:

1. Schwachstelle identifizieren (z.B. durch gezieltes Herbeiführen von Abstürzen mit zu langen Eingaben)
2. Schadhafte Eingabe konstruieren aus:
   1. NOPs (\x90) (bekommt der Prozessor diese in ausführbarem Code, springt er einfach eins weiter)
   2. ausführbarer Code (z.B. /bin/sh)
   3. Geeignete Rücksprungadressen, die in den Bereich der NOPs zeigen



Angriffsszenarien

- Angriff von Innen:
  - Ausnutzung von Schwachstellen in Programmen die mit mehr Rechten laufen, als der Angreifer hat.
  - Ziel: Systemabsturz oder Erlangung von Information, die dem Angreifer regulär nicht zugänglich sind
  - Beispiele für erfolgreich angegriffene Programme:

    `sendmail, XFree86, xterm, WinAmp, Oracle9i, chpass, MS Media Player, MySQL`

- Angriff von Aussen:
  - Ausnutzung von Schwachstellen in Serverdiensten.
  - Vorteil: Der Angreifer, benötigt zum Angriff keinen Account auf dem anzugreifendem System
  - Ziel: Systemabsturz oder Erlangung von Information, die dem Angreifer regulär nicht zugänglich sind
  - Beispiele für erfolgreich angegriffene Programme:

    `sshd, ftpd, MS SQL Server, telnetd, pppd, MS IIS, MySQL`
  - Kombinierte Angriffsform: Der Angreifer schleust über eine Schwachstelle in eien Serverdienst Code ein, welcher dann von Innen weiter Systemprogramme attackiert.
  - Auch kombinierbar mit Social-Engineering wie etwa „Anna Kournikova“ etc.

Angriffsszenarien

- Angriff von Innen:
  - Ausnutzung von Schwachstellen in Programmen die mit mehr Rechten laufen, als der Angreifer hat.
  - Ziel: Systemabsturz oder Erlangung von Information, die dem Angreifer regulär nicht zugänglich sind
  - Beispiele für erfolgreich angegriffene Programme:

    `sendmail, XFree86, xterm, WinAmp, Oracle9i, chpass, MS Media Player, MySQL`
- Angriff von Aussen:
  - Ausnutzung von Schwachstellen in Serverdiensten.
  - Vorteil: Der Angreifer, benötigt zum Angriff keinen Account auf dem anzugreifendem System
  - Ziel: Systemabsturz oder Erlangung von Information, die dem Angreifer regulär nicht zugänglich sind
  - Beispiele für erfolgreich angegriffene Programme:

    `sshd, ftpd, MS SQL Server, telnetd, pppd, MS IIS, MySQL`
  - Kombinierte Angriffsform: Der Angreifer schleust über eine Schwachstelle in eien Serverdienst Code ein, welcher dann von Innen weiter Systemprogramme attackiert.
  - Auch kombinierbar mit Social-Engineering wie etwa „Anna Kournikova“ etc.

Modernere Angriffe

- Heap Overflow:
  - Betrifft dynamisch allokierte Speicherblöcke (z.B. malloc(3))
  - Überschreiben von Programmvariablen durch Usereingabe möglich.
  - Beispiel: Standardausgabefile eines Systemprogramms wird auf /etc/passwd gesetzt und dort ein zweiter root-User angelegt

ID	Date	PublicTitle
VU#60254004	May 2015	ICU Project ICU4C library contains multiple overflow vulnerabilities
VU#81057208	Jun 2015	CUPS print service is vulnerable to privilege escalation and cross-site scripting
VU#69594004	Feb 2015	Henry Spencer regular expressions (regex) library contains a heap overflow vulnerability
VU#73096419	Aug 2014	FortiNet FortiGate and FortiWiFi appliances contain multiple vulnerabilities

Modernere Angriffe

- Heap Overflow:
  - Betrifft dynamisch allokierte Speicherblöcke (z.B. malloc(3))
  - Überschreiben von Programmvariablen durch Usereingabe möglich.
  - Beispiel: Standardausgabefile eines Systemprogramms wird auf /etc/passwd gesetzt und dort ein zweiter root-User angelegt

ID	Date	PublicTitle
VU#60254004	May 2015	ICU Project ICU4C library contains multiple overflow vulnerabilities
VU#81057208	Jun 2015	CUPS print service is vulnerable to privilege escalation and cross-site scripting
VU#69594004	Feb 2015	Henry Spencer regular expressions (regex) library contains a heap overflow vulnerability
VU#73096419	Aug 2014	FortiNet FortiGate and FortiWiFi appliances contain multiple vulnerabilities

- BSS-Overflow
  - Betrifft globale Systemvariablen
  - Beispiel: Überschreiben von Variablen für aufgerufene Unterprogramme

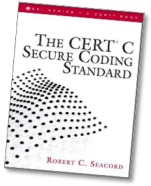
Vulnerability Note VU#773548

gzip contains a .bss buffer overflow in its LZH handling

Original Release date: 19 Sep 2006 — Last revised: 22 Jul 2011

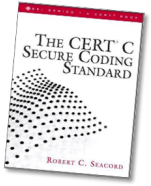
Abwehrmechanismen: Code und Audit

- Secure Programming (wichtigste Reglen zur Vermeidung von Buffer Overflows)
  - „Hardwarenahe" Sprachen wie C oder C++ nur einsetzen wo nötig.
    Alternativen: Java, C#, Perl, Python
  - Vermeidung von Befehlen, welche die Eingabelänge nicht verifizieren.
    (z.B. strncpy(3) statt strcpy(3), fgets(3) statt gets(3))
  - Manuelle Verifikation von Benutzereingaben, Schleifenkonstruktionen etc.
  - Kurz: Sicherheitsorientiertes Programmdesign



Abwehrmechanismen: Code und Audit

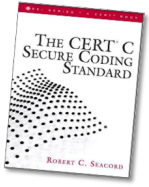
- Secure Programming (wichtigste Reglen zur Vermeidung von Buffer Overflows)
  - „Hardwarenahe" Sprachen wie C oder C++ nur einsetzen wo nötig.
    Alternativen: Java, C#, Perl, Python
  - Vermeidung von Befehlen, welche die Eingabelänge nicht verifizieren.
    (z.B. strncpy(3) statt strcpy(3), fgets(3) statt gets(3))
  - Manuelle Verifikation von Benutzereingaben, Schleifenkonstruktionen etc.
  - Kurz: Sicherheitsorientiertes Programmdesign



- Code Audit
  - Manuelle Audit durch vom Programmierer unabhängigem Auditor (Qualitätssicherung)
  - Automatisierte Code Audits durch Tools wie: flawfinder, Rats, Purify, ElectricFence etc.

Abwehrmechanismen: Code und Audit

- Secure Programming (wichtigste Reglen zur Vermeidung von Buffer Overflows)
  - „Hardwarenahe" Sprachen wie C oder C++ nur einsetzen wo nötig.
    Alternativen: Java, C#, Perl, Python
  - Vermeidung von Befehlen, welche die Eingabelänge nicht verifizieren.
    (z.B. strncpy(3) statt strcpy(3), fgets(3) statt gets(3))
  - Manuelle Verifikation von Benutzereingaben, Schleifenkonstruktionen etc.
  - Kurz: Sicherheitsorientiertes Programmdesign

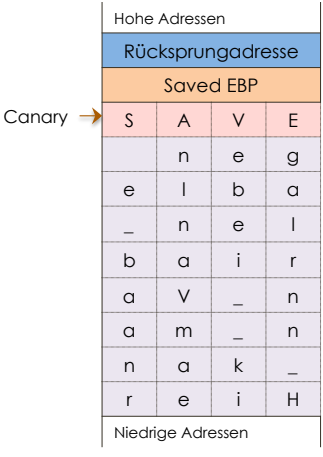


- Code Audit
  - Manuelle Audit durch vom Programmierer unabhängigem Auditor (Qualitätssicherung)
  - Automatisierte Code Audits durch Tools wie: flawfinder, Rats, Purify, ElectricFence etc.
- Binary Audit
  - Im wesentlichen identisch zu Aktionen eines Angreifers, nur mit dem Ziel Schwachstellen zu identifizieren
  - Automatisierte Audits: Netzwerkbasiert (z.B. Hallstorm), Hostbasiert (z.B. BFBTester, Sharefuzz)

Abwehrmechanismen: Compiler und Library

- Canary-Basierter Stack-Schutz
  - Im Stack wird zwischen den gespeicherten Adressen und Variablen ein (Zufalls-)Wert abgespeichert. Wurde dieser vor dem Funktionsende verändert wird das Programm abgebrochen.
  - Implementierungen: StackGuard (Unix), StackCookie (Win)
  - DoS funktioniert immer noch....
- Sicherung der Rücksprungadresse
  - Die Rücksprungadresse wird an einem weiteren Platz gesichert und als letzte Operation vor Beenden der Funktion wieder zurück geschrieben
  - Implementierung: StackShield(Linux)
- Safe Library (z.B. Libsafe)

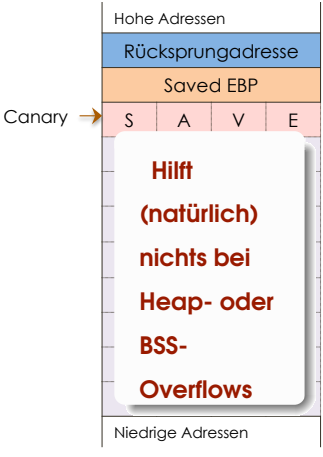
  Normale Aufrufe unsichere Funktionen werden von einem Wrapper durch sichere ersetzt



Abwehrmechanismen: Compiler und Library

- Canary-Basierter Stack-Schutz
  - Im Stack wird zwischen den gespeicherten Adressen und Variablen ein (Zufalls-)Wert abgespeichert. Wurde dieser vor dem Funktionsende verändert wird das Programm abgebrochen.
  - Implementierungen: StackGuard (Unix), StackCookie (Win)
  - DoS funktioniert immer noch....
- Sicherung der Rücksprungadresse
  - Die Rücksprungadresse wird an einem weiteren Platz gesichert und als letzte Operation vor Beenden der Funktion wieder zurück geschrieben
  - Implementierung: StackShield(Linux)
- Safe Library (z.B. Libsafe)

  Normale Aufrufe unsichere Funktionen werden von einem Wrapper durch sichere ersetzt



Abwehrmechanismen: Betriebssystem

- Non-Executable Stack
  - Daten im Stack werden als nicht ausführbar markiert
  - Implementierungen: NX-Bit (Unix), DEP (Win)
  - Auch programmabhängig als Compiler-Flag (executable-flag) einsetzbar
  - Hilft nicht bei Heap-Overflows oder ret2lib-Angriffen
- Address-Space-Layout-Randomisation (ASLR)
  - Adressbereiche werden vom Betriebssystem zufällig vergeben und so die Ermittlung einer geeigneten Rücksprungadresse erschwert
  - Implementierungen: Unix und Win (mittlerweile Kernel Standard)
  - Neuere Angriffe umgehen diesen Mechanismus, z.B. JavaVM-Attack, Spraying

Zusammenfassung

- Buffer-Overflows (Schwachstellen- und Angriffe) sind immer und überall
- Ein erfolgreicher Buffer-Overflow-Angriff kann beliebig verheerende Auswirkungen haben
- Die Entdeckung von Schwachstellen ist verhältnismäßig einfach und die Wahrscheinlichkeit von Angriffen hoch
- 100% Schutz nicht möglich (zumindest nicht jetzt und auf absehbare Zeit)
- Schutzmechanismen sind kontinuierlich weiter zu entwickeln!
- Der Beste Schutz ist „Sicheres Programmieren“

Danke für Ihr Interesse

Fortsetzung folgt