LinuxKP.org 28.05.2001

Sicherheitsrelevante Programmierfehler (Teil 1)

Thomas Biege

Inhaltsverzeichnis

- Einleitung/Motivation
- C
    - Speicherüberläufe

Einleitung/Motivation

"Warum sollte ich als Programmierer auf Sicherheit achten?"
"Ich will mein Programm nur schnell fertigstellen und es soll so viele Funktionen wie möglich bieten."

Diese Aussagen spiegeln in der Regel die Einstellung von vielen (natürlich nicht allen) Programmierern wider. Diese pragmatischen und kurzsichtigen Ansichten entstehen aber nicht grundlos. Die Entwicklerteams von kommerzieller, aber auch nichtkommerzieller (s. KDE vs. GNOME, Kernelentwicklung etc.), Software stehen unter einem hohen Erwartungsdruck. Riesige Softwareprodukte müssen in Rekordzeit an den Kunden gebracht werden, nur weil der Manager, der den Vertrag mit den Kunden ausgehandelt hat, aufgrund mangelnder Erfahrung nicht in der Lage war, einen angemessenen Zeitraum festzulegen. Oder es muß ein Konkurrenzkampf gewonnen werden. Aufgrund dieser Tatsache kommt es immer wieder zu Sicherheitsproblemen in großen und kleinen Programmen. Diese Sicherheitsprobleme führen zu Kosten nicht nur innerhalb der Entwicklungsfirma, sondern auch beim Kunden.

Diese Kosten setzen sich aus folgenden Punkten zusammen:

- Programmierer müssen von laufenden Projekten abgezogen werden, um die Sicherheitslücke zu beheben
- Kunden müssen informiert werden
- das neue Produkt muß dem Kunden zur Verfügung gestellt werden
- der Kunde muß das alte Produkt ersetzen, was zu Fehlern führt, die wiederum die eigene Supportabteilung belasten
- das Image des Unternehmens wird angekratzt, was langfristig zum größten Schaden führt (s. Einbrüche in das Microsoft-Netzwerk über Microsoft-Produkte)
- Zudem müssen Produkte für Banken und Versicherungen i.d.R. von einem externen (und sehr teuren) Consultant überprüft werden

Wenn von Anfang an an die Sicherheit (als Teil der Qualitätssicherung) gedacht wird, kann ein Großteil dieser Kosten vermieden werden. Dazu ist es nötig, daß die Entwickler über die Schwächen und Eigenschaften der verwendeten Programmiersprache Bescheid wissen und sie vermeiden/umgehen können.

In Sonderfällen, bei Finanz-/Versicherungsprodukten, privilegierten Serverprogrammen usw. sollte zusätzlich ein Quellcodereview von zwei erfahrenen Personen, i.d.R. Programmierer, die nicht an dem Projekt beteiligt sind, nacheinander durchgeführt werden (Vier-Augen-Prinzip) um ganz sicher zu gehen, daß sich keine Fehler eingeschlichen haben.

Mit dem Wissen aus diesem Text und dem Quellcodereview sollten viele Fehler aus dem Softwareprodukt verschwinden und somit die Folgekosten gering bleiben.

C

Die Sprache C ist eine der am häufigsten verwendeten Programmiersprachen. Die Kernel von Bestriebsystemen werden mit ihr genauso programmiert wie kleine privilegierte Systemtools, Server-Daemonen und grafische Oberflächen (C/C++).

Aus diesem Grund werden die meisten Sicherheitslücken in Programmen gefunden, die mit C geschrieben wurden. Das läßt nicht den Rückschluß zu, daß C eine "unsichere" Sprache ist, sondern nur, daß C sehr verbreitet, kraftvoll und flexibel ist.

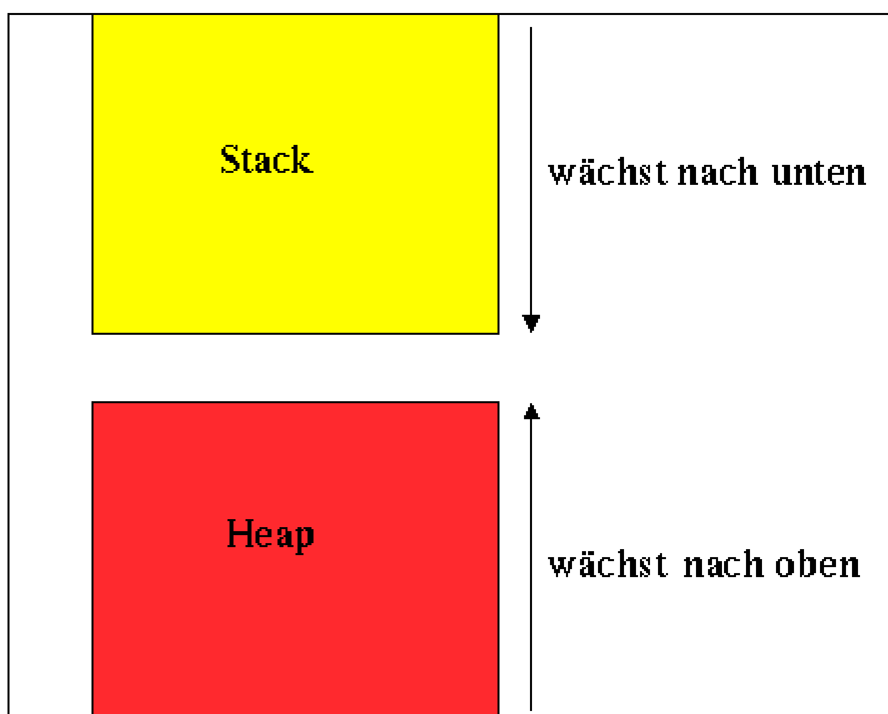
Viele der in diesem Abschnitt beschriebenen Sicherheitsrisiken treffen auch auf andere Programmiersprachen zu, da sie abhängig von der Betriebsumgebung sind.

Speicherüberläufe

Speicherüberläufe (engl. **Buffer Overflows**) sind einer der bekanntesten Gründe für Sicherheitsprobleme und Programmabstürze. Sie entstehen überall da, wo Daten aus nicht vertrauenswürdigen Quellen, wie Tastatur, Netzwerk oder Benutzerdateien, in einen Speicherbereich mit statischer Größe ohne Längenprüfung geschrieben werden.

Die Auswirkungen von Speicherüberläufen hängen stark von dem Speichertyp ab. Bevor wir jedoch dazu kommen, müssen wir noch einen kleinen Blick auf die Speicherverwaltung von Programmen auf intelbasierten Prozessoren werfen, um ein besseres Verständnis für die Vorgänge bei den folgenden Attacken zu bekommen.

Lokale Variablen werden auf dem Stack und globale Variablen auf dem Heap gespeichert. Stack und Heap teilen sich denselben Speicherbereich, wobei der Heap von unten nach oben und der Stack von oben nach unten wächst.

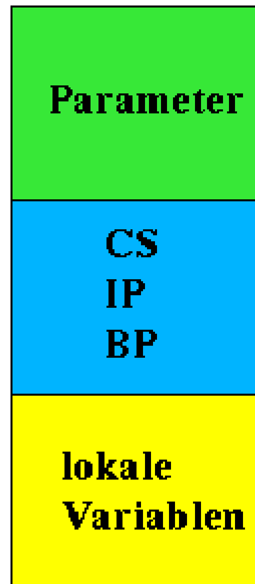


Jede Funktion hat ihren eigenen **Stack-Frame**, auf dem sie ihre lokale Variablen speichert. Dasselbe gilt für Programmblöcke, in C/C++ durch die geschweiften Klammern { und } gekennzeichnet. Neben lokalen Daten müssen vor dem Sprung zum Maschinencode der Funktion auf dem Stack noch Prozessorregisterinhalte gesichert werden, um nach Beendigung der Funktion einen Rücksprung an die aufrufende Funktion zu ermöglichen.

Erst werden die Funktionsparamter vor dem Aufruf des Unterprogramms mit *PUSH* auf dem Stack geschrieben.

Der Assembler Befehl *CALL* sichert die aktuelle **Position im Maschinencode**, die durch *CS:IP* dargestellt wird, und das *BP* Register, das für den eigenen Stackframe, der aufrufenden Funktion, benötigt wird.

Letztendlich richtet die Funktion ihren **Stack-Frame** für die lokalen Variablen ein.



Wenn die Funktion beendet ist, springt sie an ihr Oberprogramm zurück, indem sie mit *RET* die auf dem Stack gesichertern Register *CS*, *IP* und *BP* restauriert. Das bedeutet, daß die Ausführung des Maschinencodes an der Stelle weitergeht, auf die das gesicherte *CS:IP* zeigt, also nach dem *CALL* Befehl.

Nun aber zu den Auswirkungen.

Stack. Speicherüberläufe auf dem Stack können auf drei verschiedene Arten ausgenutzt werden.

- Inhalte von Variablen, die auf dem Stack über der betroffenen Variable liegen, können mit beliebigen Daten vom Angreifer überschrieben werden. Ein klassisches Beispiel für ein ausnutzbares Sicherheitsloch ist die Authentifikation mit Hilfe eines Paßworts. Dabei wird erst das Paßwort aus einer lokalen Datenbank gelesen und in eine Variable gespeichert. Im späteren Verlauf des Programms wird das Paßwort vom Benutzer abgefragt und die beiden Zeichenketten verglichen.

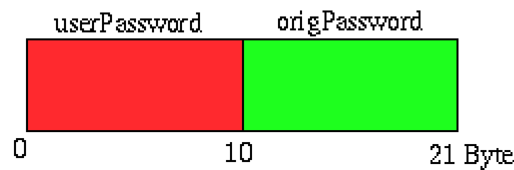
  Beispiel:

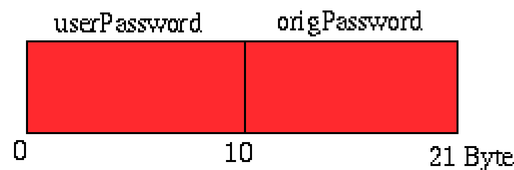
```
[...]
/* that's our secret phrase */
char origPassword[12] = "Geheim\0";
char userPassword[12];
[...]

gets(userPassword);    /* read user input */
[...]

if(strncmp(origPassword, userPassword, 12) != 0)
{
        printf("Password doesn't match!\n");
exit(-1);
}
[...]
/* give user access to everything */
[...]
```

Wenn der Benutzer jetzt mehr als 12 Zeichen (32-Bit Alignment) eingibt, dann überschreibt er den Inhalt von `origPassword[]`. Gibt er also `sesamoeffne!sesamoeffne`' ein, so enthält `userPassword[]` und `origPassword[]` denselben String und zwar `sesamoeffne!`; ein Vergleich wird also positiv ausfallen.





- Es können natürlich nicht nur Variableninhalte, sondern auch die **gesicherten Register** auf dem Stack überschrieben werden. Gibt der Angreifer also noch mehr Zeichen ein und überschreibt den **Instruction Pointer** (*IP*), dann kann er den Programmcode durch *RET* am Ende der Funktion an einer beliebiger Stelle des Programms ausführen. In der Regel wird aber nicht auf programmeigenen Code gesprungen, sondern die CPU wird mit eigenem Maschinencode gefüttert. Dazu wird der Maschinencode in die Variable, also den Stack, geschrieben und zusätzlich die gesicherte IP-Adresse auf die Startadresse des fremden Programmcodes gesetzt. Wenn die Variable zu klein ist, um den Maschinencode aufzunehmen, kann er auch in dem Programmenvironment, auf dem Heap oder sonstwo im adressierbaren Raum abgelegt werden. Wenn nun die Funktion an ihrem Ende angelangt ist, wird durch *RET* das *IP* Register der CPU mit dem IP-Wert von unserem Stack, der durch den Angreifer gesetzt wurde, belegt und der Computer führt nun treu die Codesequenzen des Angreifers aus.

- Desweiteren können **Zeiger auf Funktionen** überschrieben werden, um bei Verwendung dieses Zeigers fremden Code auszuführen. Es läuft wieder nach dem alten Schema ab. Der Angreifer speichert seinen Maschinencode in eine globale oder lokale Variable oder auch im Programmenvironment - hierzu ist kein Overflow nötig, es wird lediglich ein Aufbewahrungsort für den Code benötigt - und läst den Funktionszeiger auf seinen Programmcode zeigen.

  Wird der Funktionszeiger benutzt, um die Funktion aufzurufen, wird nicht etwa der Funktionscode, sondern der Code des Angreifers ausgeführt.

  Beispiel:

```
[...]
long (* funcptr) () = atol();
[...]
/*
** irgendwo im erreichbaren Speicher schreibt der
** Angreifer seinen Code
*/
[...]

/*
** durch einen Overflow im Programmcode überschreibt
** der Angreifer den Wert, den (*funcptr) enthält,
** mit der Startadresse seines Codes
*/
[...]
```

```
/*
** die Funktion wird über den Zeiger aufgrufen und
** somit der Fremdcode ausgeführt
*/
(*funcptr)(string);
[...]
```

Heap. Genau wie beim Stack können auch durch Überläufe auf dem Heap Daten und Funktionszeiger verändert werden, die das Verhalten des Programms zugunsten des Angreifers verändern. Der Heap bietet noch die Möglichkeit, die *jmp_buf* Variable der *setjmp(3)* Funktion zu überschreiben. Der Jumpbuffer enthält unter anderem die Adresse der Programmcodestelle beim Aufruf von *setjmp(3)*. Wird dieser Wert mit der Startadresse des eigenen Maschinencodes überschrieben und *longjmp(3)* aufgerufen, dann wird das *IP*-Register auf den Beginn des fremden Codes gesetzt und damit zur Ausführung gebracht.

Wertebereich. Fehler, die man nicht sehr leicht findet, entstehen durch Wertebereichsüberschreitung bei Zahlenvariablen. Einige Codebeispiele sollen die Gefahr verdeutlichen.

Beispiel:

1.)

```
[...]
unsigned int uintAnzahl = GetZahlFromUser();
unsigned int uintGroesse = uintAnzahl * sizeof(struct
                                            myStructure);

/*
** Wenn der Benutzer das Maximum für den
** Wertebereich unsigned int (UINT_MAX definiert in
** limits.h) als Zahl eingibt, dann wird
** durch die Multiplikation ein Wert größer als
** UINT_MAX errechnet.
** Die Variable erfährt nun einen Überlauf, was zur
** Folge hat, daß
** uintGroesse einen kleineren Wert zugewiesen
** bekommt.
*/

myStructureArray[i] = malloc(uintGroesse);

/*
** durch den Aufruf von malloc(3) wird also ein
** kleiner Speicherbereich alloziert als
** UINT_MAX * sizeof(struct myStructure);
** Der kleinere Buffer führt zwangsläufig zu einen
** Bufferoverflow.
*/
[...]
```

2.)

```
[...]
unsigned int uintAnzahl = GetZahlFromUser();
myArray[i] = malloc(uintAnzahl + strlen("oops!"));

/*
** Bei Addition passiert natürlich dasselbe,
** der von malloc(3) allozierte Buffer ist zu klein
*/
```

3.)

```
[...]
char Buffer[1024];
[...]

int intAnzahl = GetZahlFromUser();
[...]

if(intAnzahl > sizeof(Buffer))
{
fprintf(stderr, "Buffer zu klein!\n");
exit(-1);
}

/*
** Wenn wir für intAnzahl -1 angeben, dann ist der
** Ausdruck in der if-Bedingung FALSE, aber wenn
** wir später ein malloc(3), memcpy(3) oder
** ähnliches machen, dann wird -1 als positiver
** Wert von den Funktionen gehandelt. Der Wert -1
** entspricht ca. 4 GB.
*/
[...]
```

Am Ende dieses Abschnittes werden einige **System-/Bibliotheksaufrufe** und **Programmsegmente** aufgelistet, die die **häufigste Ursache für Speicherüberläufe** sind.

- gets(3)

  Daten werden von stdin in einen statischen Buffer gelesen. Der bekannteste Bug dieser Art wurde vom **Moris Internet Wurm** im *fingerd* ausgenutzt, um über das Netzwerk auf einem Rechner Kommandos auszuführen.

  Fehler:

  ```
  [...]
  char HopeItFits[12];
  [...]

  while(gets(HopeItFits) != NULL)
  {
      puts(HopeItFits);
      memset(HopeItFits, 0, sizeof(HopeItFits));

  }
  [...]
  ```

  Korrektur:

  Mit *fgets(3)* können Daten durch Größenbegrenzung sicher eingelesen werden. Wir geben die Menge der einzulesenden Daten mit *sizeof(HopeItFits)*, also 12 Bytes, an. *fgets(3)* liest dann nur 12-1 Bytes und fügt zudem am Ende des Strings das *NUL* Zeichen ein, dadurch entstehen keine Probleme bei der Weiterverarbeitung des Strings mit den *str*-Funktionen* aus *string.h*.

  ```
  [...]
  char HopeItFits[12];
  [...]
  ```

```
while(fgets(HopeItFits, sizeof(HopeItFits), stdin) !=
                                                    NULL)

{
     puts(HopeItFits);
     memset(HopeItFits, 0, sizeof(HopeItFits));
}
[...]
```

- scanf(3)

  Auch von den *scanf-Funktionen* werden Daten i.d.R. ohne Längenprüfung eingelesen.

  Fehler:

```
[...]
char HopeItFits[12];
[...]

while(scanf("%s", HopeItFits) != NULL)
{
     puts(HopeItFits);
     memset(HopeItFits, 0, sizeof(HopeItFits));
}
[...]
```

  Korrektur:

  Bei **scanf(3)* kann im *Format-String* bei den *Format-Tags* eine Größenbegrenzung eingestellt werden. Für Strings geht das mit %.<Größe>s.

```
[...]
char HopeItFits[12];
[...]

while(scanf("%.11s", HopeItFits) != NULL)
{

     HopeItFits[11] = `\0`;
     puts(HopeItFits);
     memset(HopeItFits, 0, sizeof(HopeItFits));
}
[...]
```

- *sprintf(3)

  Im Grunde besteht hier dasselbe Problem wie mit den *scanf-Funktionen*. Die Größenbegrenzungen können hier entweder auch in den *Format-Tags* definiert werden oder über die Funktion *snprintf(3)* bzw. *vsnprintf(3)*, die als zweiten Parameter die Größe des Zielpuffers enthält.

  Fehler:

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]

while(scanf("%.120s", BigBadBuffer) != NULL)
{
```

```
        BigBadBuffer[111] = `\0`;
        sprintf(HopeItFits, "%s", BigBadBuffer);
        [...]
        memset(HopeItFits, 0, sizeof(HopeItFits));
        memset(BigBadBuffers, 0, sizeof(BigBadBuffers));
}
[...]
```

Korrektur:

- *Format-Tags:*

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]

while(scanf("%.120s", BigBadBuffer) != NULL)
{

     BigBadBuffer[111] = `\0`;
     sprintf(HopeItFits, "%.11s", BigBadBuffer);
     [...]

     memset(HopeItFits, 0, sizeof(HopeItFits));
     memset(BigBadBuffers, 0,
                         sizeof(BigBadBuffers));
}
[...]
```

- *snprintf(3):*

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]

while(scanf("%.120s", BigBadBuffer) != NULL)
{
     BigBadBuffer[111] = `\0`;
     snprintf(HopeItFits, sizeof(HopeItFits), "%s",
                                     BigBadBuffer);
    [...]

memset(HopeItFits, 0, sizeof(HopeItFits));
memset(BigBadBuffers, 0,
                    sizeof(BigBadBuffers));
}
[...]
```

- strcpy(3)/strcat(3)

  Auch bei *strcpy(3)* und *strcat(3)* muß auf die Größe des Zielpuffers geachtet werden.

  Fehler:

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]
```

```
while(scanf("%.120s", BigBadBuffer) != NULL)
{
     BigBadBuffer[111] = `\0`;
     strcpy(HopeItFits, BigBadBuffer);
     [...]

     memset(HopeItFits, 0, sizeof(HopeItFits));
     memset(BigBadBuffers, 0, sizeof(BigBadBuffers));
}
[...]
```

Korrektur:

Mit *strncpy(3)* bzw. *strncat(3)* kann die Anzahl der zu kopierenden Bytes angegeben werden. Man muß jedoch aufpassen, da *strncpy(3)/strncat(3)* genau die Anzahl der Bytes kopiert, die man als drittes Argument beim Funktionsaufruf angibt, und *(strncpy(3)/strncat(3))* kein *NUL*-Byte dem String anhängt. Diese besondere Eigenschaft muß berücksichtigt werden. *strncpy(3)/strncat(3)* funktioniert also nicht wie *fgets(3)*!

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]

while(scanf("%.120s", BigBadBuffer) != NULL)
{
     BigBadBuffer[111] = `\0`;
     strncpy(HopeItFits, BigBadBuffer,
                                 sizeof(HopeItFits)-1);
     HopeItFits[sizeof(HopeItFits)-1] = '\0';
     [...]

     memset(HopeItFits, 0, sizeof(HopeItFits));
     memset(BigBadBuffers, 0, sizeof(BigBadBuffers));
}
[...]
```

- strncpy(3)/strncat(3), bei falscher Benutzung

  Viele Programmierer benutzen *strncat(3)* oder *strncpy(3)* und denken, sie befänden sich damit auf der sicheren Seite, vergessen jedoch die besondere Eigenschaft von *strncpy(3)* (s.o.). Dadurch entsteht ein 1-Byte Speicherüberlauf, der zum Absturz (Segmentation Fault) führt, aber unter Umständen auch ein Sicherheitsloch darstellt.

  Fehler:

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]

while(scanf("%.120s", BigBadBuffer) != NULL)
{
     BigBadBuffer[111] = `\0`;
     strncpy(HopeItFits, BigBadBuffer,
                                  sizeof(HopeItFits));
     [...]

     memset(HopeItFits, 0, sizeof(HopeItFits));
     memset(BigBadBuffers, 0, sizeof(BigBadBuffers));
}
```

```
[...]
```

Korrektur:

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]

while(scanf("%.120s", BigBadBuffer) != NULL)
{
     BigBadBuffer[111] = `\0`;
     strncpy(HopeItFits, BigBadBuffer,
                              sizeof(HopeItFits)-1);
     HopeItFits[sizeof(HopeItFits)-1] = '\0';
     [...]

     memset(HopeItFits, 0, sizeof(HopeItFits));
     memset(BigBadBuffers, 0, sizeof(BigBadBuffers));
}
[...]
```

- Lesen in einer Schleife ohne Beachtung der Pufferlängen

  Häufig findet man Schleifen, die Benutzereingaben solange einlesen, bis ein bestimmtes Zeichen, z.B. das *Newline-Zeichen '\n'*, im Eingabestream gefunden wurde.

  Fehler:

```
[...]
int Byte, i;
char HopeItFits[12];
[...]

i = 0;
while((Byte = getc(stdin)) != `\n`)
{
     HopeItFits[i] = Byte;
     [...]

     i++;
}
[...]
```

- *Format-Tags:*

  Wenn ein Angreifer einen Speicherüberlauf provozieren möchte, dann muß er lediglich mehr als 12 Byte eingeben, die kein Newline-Zeichen enthalten.

  Korrektur:

```
[...]
int Byte, i;
char HopeItFits[12];
[...]

i = 0;
while((Byte = getc(stdin)) != `\n`)
{
     HopeItFits[i] = Byte;
     [...]
```

```
        if(++i >= sizeof(HopeItFits))
        {
            fprintf(stderr, "Too much data read!\n");
            return(-1);
        }
    }
    [...]
```

Eine Lösung über *strncat(3)* ist natürlich auch möglich.

- getwd(3)

  Die Bibliotheksfunktion *getwd(3)* gibt den Namen des aktuellen Verzeichnisses in dem *char-Array* zurück, das ihr als Argument übergeben wurde. Ist das Array zu klein für den Namen, kommt es zu einem Speicherüberlauf. Neuere Versionen der Implementierung von *getwd(3)* schreiben maximal *PATH_MAX* Zeichen in das Array. Man ist also sicher, wenn das Array *PATH_MAX+1* Byte groß ist.

  Durch die Verwendung von *getcwd(3)* oder *get_current_dir_name(3)* kann man sicher sein, daß man nicht doch zufällig aufgrund von Implementierungsunterschieden einen Speicherüberlauf in seinem Programm hat. Es ist jedoch bei *getcwd(3)* Vorsicht geboten, da es auf alten SunOS Systemen einfach nur *popen("pwd")* aufruft, und das bringt seine eigenen Sicherheitsprobleme mit sich. (s. Abschnitt Programmumgebung)

- und viele andere

  Es gibt noch viele andere Funktionen, die keine Längenprüfung durchführen. Sie hängen von dem Betriebssystem, den vorhandenen Bibliotheken und den Implementierungen ab.

LinuxKP.org 28.05.2001