PR3 - Übungsblatt A.5

(Stand 2021-10-20 15:30)

Prof. Dr. Holger Peine
Hochschule Hannover
Fakultät IV – Abteilung Informatik
Raum 1H.2.60, Tel. 0511-9296-1830
Holger.Peine@hs-hannover.de

Thema

Zeiger

Termin

Ihre Arbeitsergebnisse zu diesem Übungsblatt führen Sie bitte bis zum 12.11.2021 vor.

1 Buffer overflow, scanf (1 Punkt)

basiert auf Vorlesung bis einschl. Abschnitt 5.d

Das folgende Programm fragt ein Passwort ab und endet nur dann erfolgreich, wenn der Benutzer das (im Programmcode eingebettete[1]) korrekte Passwort eingibt (die Idee dabei ist, dass nur bestimmte Benutzer dieses Passwort kennen – der Quelltext des Programms wäre also dem Benutzer nicht bekannt). Wir werden uns das Programm unter Sicherheitsaspekten ansehen.

Das Programm verhält sich wie folgt:

```
$ ./passwort
Bitte Passwort fuer den Hochsicherheitsbereich eingeben:
keine ahnung
Passwort falsch - Zugang verweigert!
$ ./passwort
Bitte Passwort fuer den Hochsicherheitsbereich eingeben:
GeHeIm
Passwort korrekt - Willkommen im Hochsicherheitsbereich!
```

Im folgenden drucken wir das Programm ab; es ist auch als Quelldatei `PR3_U_A.5.1_passwort.c` im gleichen Verzeichnis wie dieses Übungsblatt enthalten.

- Erläuterungsbedürftig ist evtl. die Anweisung `scanf("%[^\n]", pattern)`: In eckigen Klammern kann man bei `scanf` die Menge der gültigen Eingabezeichen angeben. Mit `^` drückt man aus, dass *alle* Zeichen gültig sind, *außer* den angegebenen. Die obige `scanf`-Anweisung liest also alles bis zum Zeilenende in den Speicherbereich `pattern` ein. Insbesondere wird durch diese `scanf`-Anweisung die Eingabe nicht an Leerzeichen aufgetrennt, damit man z.B. auch Leerzeichen im Passwort haben könnte.
- Erläuterung zu `strncmp`: Die in `string.h` deklarierte Funktion `strncmp(char s1[], char s2[], int n)` verhält sich wie `strcmp`, außer dass der Vergleich höchstens `n` Zeichen betrachtet (eine Sicherheitsmaßnahme, um nicht versehentlich über das Array-Ende hinaus zu lesen).

[1] Ein Passwort im Klartext in den Programmcode einzubetten, ist nicht besonders sicher, aber hier machen wir das der Einfachheit halber so. In echtem Code sollte das Passwort z.B. in Form seines Hash-Werts eingebettet werden – das müssen Sie hier nicht verstehen ☺

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char eingabe[32];
    char passwort[32] = "GeHeIm"; /* Dem Programmnutzer unbekannt */

    /* Ggf. muessen Sie die Reihenfolge der beiden vorstehenden
       Array-Definitionen umkehren, um den unten beschriebenen Effekt
       zu beobachten (plattformabhaengig!). Auf den Pool-PCs ist die
       obige Reihenfolge die richtige, um den gewuenschten Effekt zu
       beobachten.

       Sollten Sie auf Ihrem Rechner stattdessen einen
       Programmabsturz (z.B. mit "Speicherzugriffsfehler") erleben,
       versuchen Sie das Programm mit folgenden zwei Optionen zu
       kompilieren (achten Sie auf Leerzeichen und Unterstriche):
       gcc -fno-stack-protector -D_FORTIFY_SOURCE=0
    */

    printf("Bitte Passwort fuer den Hochsicherheitsbereich \
            eingeben:\n");

    /* Alle Zeichen (auch Leerzeichen etc.) bis Zeilenende lesen */
    scanf("%[^\n]", eingabe);

    if (!strncmp(eingabe, passwort, strlen(passwort))) {
      printf("Passwort korrekt - Willkommen im \
              Hochsicherheitsbereich!\n");
      return 0;
    } else {
      printf("Passwort falsch - Zugang verweigert!\n");
      return -1;
    }
}
```

Nun beobachten wir, was passiert, wenn man als Benutzer folgendes eingibt:

```
$ ./passwort
Bitte Passwort fuer den Hochsicherheitsbereich eingeben:
Blabla□□□□□□□□□□□□□□□□□□□□□□□□□□Blabla
```

Hierbei soll das Symbol □ ein eingegebenes Leerzeichen bedeuten (bitte wirklich genau 26 Leerzeichen tippen und nicht auf dem PDF kopieren!). Der Benutzer gibt also `Blabla`, dann 26 Leerzeichen und dann noch einmal `Blabla` ein.

Diese Eingabe ist länger als vom Programm erwartet. Das Programm erwartet maximal 31 Zeichen, denn das `eingabe`-Array kann nur 31 Zeichen und den \0-Terminator speichern. Davon weiß der `scanf`-Aufruf jedoch nichts. Die ersten 32 Zeichen werden nun von `scanf` eingelesen und in das Array `eingabe` (liegt als lokale Variable auf dem Stack) geschrieben. Die überzähligen Zeichen werden ebenfalls durch die Funktion `scanf` eingelesen und weiter auf den Stack geschrieben. Und zwar (je nach Stacklayout) *hinter* `eingabe`, also an den

Anfang des Arrays `passwort`. Somit wird das darin gespeicherte Passwort überschrieben[2] und zwar mit den "überzähligen" Zeichen aus der Benutzereingabe, d.h. mit dem zweiten `Blabla`. Dadurch ist der Stringvergleich erfolgreich (`Blabla` wird mit `Blabla` verglichen), und der Benutzer wird in den Hochsicherheitsbereich gelassen, obwohl er das richtige Passwort gar nicht kennt. Wir haben hier also ein schönes Beispiel, wie ein Pufferüberlauf zu einem unerwünschten Verhalten des Programms führen kann.

Verschaffen Sie sich einen Einblick in die Daten auf dem Stack, indem Sie folgendes Codefragment nach dem `scanf` ins Programm einfügen, das die Daten als Bytes im Hexadezimalformat ausgibt:

```
for (i=0; i<64; i++) {
  printf("%3d (%p): %02X %c\n",
         i, eingabe+i, (unsigned char)eingabe[i], eingabe[i]);
}
```

Ihre Aufgabe:
a) Erklären Sie das Verhalten des Programms und die Ausgabe der Stack-Daten.
b) Korrigieren Sie das Programm, so dass das beschriebene Problem nicht mehr auftreten kann. Wie können Sie den `scanf`-Aufruf sicherer machen? Lesen Sie dazu die Beschreibung von `scanf`, z. B. durch Aufruf der man-page (Suchwort: `WIDTH`).

2 Zeiger auf Strukturen (2 Punkte)

basiert auf Vorlesung bis einschl. Abschnitt 5.e

Erweitern Sie das Programmstück aus der Vorlesung, Kapitel 5.e zu einem vollständigen Programm.

Strukturdefinition:

```
#define NAME_LEN 40
typedef struct {
    char name[NAME_LEN+1];
    int personalnummer;
    float gehalt;
} angestellter;
```

Array von Angestellen-Zeigern:

```
angestellter *array[arraygröße];
```

Einen Angestellten anlegen:

```
array[i]= (angestellter*)malloc(sizeof(angestellter));
array[i]->personalnummer = 1234;
```

Das Programm soll Folgendes leisten:

- Erzeugung eines Speicherbereichs, der zehn Adressen von Angestellteninformationen aufnehmen kann. Initialisierung des Speichers mit Nullzeigern.
- Benutzereingabe eines Indexwerts i (0<=i<=9):
  - Steht an der Stelle i des Speichers der Nullzeiger, so soll das Programm vom Benutzer die Daten eines Angestellten anfordern. Es soll eine neue Struktur

[2] Das Überschreiben funktioniert unter Windows häufig nicht, weil Windows dies mit speziellen Maßnahmen erschwert.

erzeugen, mit den eingegebenen Daten besetzen und an der i-ten Speicherposition ablegen.
- Steht an der Stelle i bereits eine Struktur, so soll sie gelöscht werden.

- Nach der Eingabe oder dem Löschen sollen sämtliche gespeicherten Daten auf den Bildschirm ausgegeben werden.
- Eingabe, Löschen und Ausgabe sollen beliebig oft wiederholt werden können, bis der Benutzer ein Ende der Programmausführung wünscht.
- Fehleingaben soll das Programm nicht berücksichtigen, d. h. das Programm darf von korrekten Benutzereingaben ausgehen. Techniken zur Fehlererkennung bei Benutzereingaben lernen wir später kennen.

Hinweis:

Wenn Sie einen Namen mit Leerzeichen einlesen möchten, können Sie das z. B. so tun:

```
char name[50+1];
printf("Name: ");
scanf("%50[^\n]", name);
```

Bedeutung der Formatangabe bei `scanf`: Lies alles, außer dem Zeilenumbruch, in den Speicherbereich, auf den `name` zeigt, aber maximal 50 Zeichen, um das Array nicht zum Überlaufen zu bringen. (Leider kann man hier die 50 nicht durch ein #define realisieren, weil sie als Zahlkonstante "50" im ersten Argument des scanf-Aufrufs erscheinen muss – und in Stringkonstanten (hier: `"%50[^\n]"`) wirken #define's nicht.)

Wenn Sie nun aufgabengemäß kurz vorher eine Zahl von der Konsole einlesen, kann es sein, dass das abschließende `\n` hinter der Zahleingabe Ihnen die Eingabe des Namens vermasselt. Leeren Sie daher den Eingabepuffer unmittelbar nach der Eingabe der Zahl. Etwa so:

```
int i ;
char name[50+1];

/* Zahl einlesen */
printf("i: ");
scanf("%d", &i);

/* Puffer leeren */
do {} while (getchar() != '\n');

/* Zeichenkette einlesen */
printf("Name: ");
scanf("%50[^\n]", name);

/* ausgeben */
printf("%d\n", i);
printf("%s\n", name);
```

3 Strukturen mit Zeigern auf Strukturen (0 Punkte)

basiert auf Vorlesung bis einschl. Abschnitt 5.e

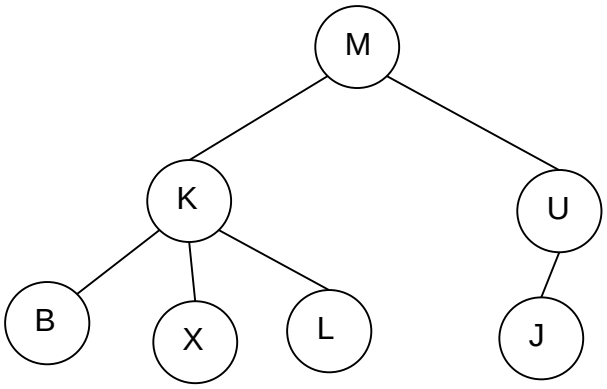
Erstellen Sie eine Datenstruktur in C für einen allgemeinen Baum. In der in dieser Aufgabe beschriebenen Version sollen die Kindknoten jeweils eine Referenz (Zeiger) auf ihren übergeordneten Knoten besitzen. Übergeordnete Knoten kennen ihre Kindknoten nicht.

Die Datenstruktur soll folgendem Pseudocode genügen:

Strukturtyp Knoten
Name des Knotens
Zeiger auf Vorgänger

Die Daten des Baums kommen aus einer Datei, die Ihr Programm lesen soll.
Beispieldatei:

```
B K
X K
K M
M <none>
J U
U M
L K
```



Die Beispieldatei beschreibt den rechts dargestellten Baum. Jede Zeile gehört zu einem Knoten und enthält zuerst den Namen des Knotens (der Name kann im Prinzip auch mehrere Zeichen umfassen) und dann den Namen des übergeordneten Knotens. Die Reihenfolge der Knoten in der Datei ist beliebig. Der Wurzelknoten besitzt keinen übergeordneten Knoten und ist erkennbar an dem Eintrag <none> für den übergeordneten Knoten.

Hinweise:

- Leiten Sie die Datei mit den Baumdaten beim Programmstart auf die Standardeingabe um, etwa so: `./programm < graph.txt`
- Verwenden Sie `scanf`, um die Datei zeilenweise auszulesen (s. auch http://www.cppreference.com/wiki/c/io/start). Der Rückgabewert von `scanf` zeigt Ihnen jeweils an, wieviele Variablen erfolgreich gelesen werden konnten. So ermitteln Sie leicht, wann der Eingabestrom versiegt ist.
- Da Sie nicht wissen, wie viele Zeilen die Datei besitzt, dürfen Sie annehmen, dass die Länge der Datei maximal 100 Zeilen beträgt.
- Da Sie nicht wissen, wie lang die einzelnen Knotennamen sind, dürfen Sie annehmen, dass jeder Knotenname eine Länge von 20 Zeichen nicht überschreitet.
- Zunächst müssen Sie die Datei zeilenweise einlesen. Dabei füllen Sie ein Array der Knoten mit den Knotennamen.
- Da Sie beim ersten Durchlauf eventuelle Vorwärtsreferenzen nicht auflösen können (beispielsweise verweist die erste Zeile der obigen Beispieldatei auf den Vorgängerknoten „K“, den Sie ja noch gar nicht eingelesen haben), schlage ich vor, dass Sie beim Einlesen ein zweites Array mit den Namen der Vorgängerknoten aufbauen. Anschließend durchlaufen Sie die eingelesenen Arrays noch einmal, um die Vernetzung der Datenstruktur zu vervollständigen.
- Um Knotennamen zu vergleichen, können Sie die Funktion `strcmp` benutzen (s. http://www.cppreference.com/wiki/c/string/strcmp).
- Geben Sie am Ende Ihre Datenstruktur aus. Die Ausgabe soll den Wurzelknoten besonders markieren, etwa so:

```
Anzahl Knoten: 7
 B -> K
 X -> K
 K -> M
*M -> -
 J -> U
 U -> M
 L -> K
```

4 Call by reference (1 Punkt)

basiert auf Vorlesung bis einschl. Abschnitt 6.b

Schreiben Sie eine Funktion `free0`, die die Freigabe eines beliebigen Zeigers erledigt und dabei den Zeiger auf den Wert `NULL` setzt. Der Zeiger muss dafür an die Funktion per „call by reference“ übergeben werden. Die Vorlesungsfolien enthalten mehr Informationen dazu, wie man in C call by reference realisiert.
Das folgende Hauptprogramm testet Ihre Funktion:

```
int main(void) {
  char* p = malloc(10);
  printf("%p\n", p); /* Ausgabe z. B. 0x470228 */
  free0( /* Ihr Code, irgend ein Ausdruck mit p */ );
  printf("%p\n", p); /* Ausgabe 0x0 oder 0 oder (nil) o. ä. */
  return 0;
}
```

5 Buffer Overflow minimal (0 Punkte)

basiert auf Vorlesung bis einschl. Abschnitt 5.d

Betrachten Sie folgendes Beispielprogramm (admin.c):

```
#include <stdio.h>

struct user
{
  char name[10];
  int admin;
};

int main(void)
{
  struct user u = { "", 0 };
  printf("Name: ");
  scanf("%s", (char*)&u.name);
  printf("Hallo %s!\n", u.name);
  if (u.admin != 0)
       printf("Gratulation! Sie sind Admin!\n");
  return 0;
}
```

a) Wie können Sie Admin werden, ohne das Programm zu verändern?
b) Ersetzen Sie den Aufruf der Funktion `scanf` durch einen Aufruf der Funktion `fgets`. Recherchieren Sie dafür im Internet, wie die Funktion aufgerufen werden muss. Kann die entdeckte „Sicherheitslücke“ jetzt noch ausgenutzt werden?

6 Zeichenkette als verkettete Liste (0 Punkte)

basiert auf Vorlesung bis einschl. Abschnitt 5.e

Es wäre zwar äußerst ineffizient hinsichtlich Speicherbedarf und Laufzeit, aber man könnte Zeichenketten auch als verkettete Liste aus Einzel-Zeichen repräsentieren. Dabei wird für jedes Zeichen eine Struktur auf dem Heap angelegt, welche das Zeichen und einen Zeiger auf

die nächste Struktur enthält. Der Zeiger der letzten Struktur ist NULL und markiert so das Ende der Zeichenkette. Sie sollen nun die folgenden Funktionen implementieren (eine Vorlage finden Sie in den Anlagen):

a) create_string erhält eine klassische C-Zeichenkette und erzeugt daraus eine verkettete Liste. Jedes Element der Liste soll separat im Heap abgelegt werden (Aufruf von malloc), d. h. es sollen keine Arrays verwendet werden.
b) free_string gibt den Speicher für die Elemente der verketteten Liste wieder frei (Aufruf von free). Achten Sie darauf, dass Sie nicht mehr auf freigegebenen Speicher zugreifen dürfen.
c) print_string gibt die Elemente der verketteten Liste Zeichen für Zeichen aus (Aufruf von printf mit %c).

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

struct character
{
  char c;
  struct character* next;
};
typedef struct character string;

string* create_string(char* str)
{
  string* first = NULL;

  /* Hier sollen Sie Ihren Code einfuegen. */

  return first;
}

void delete_string(string* str)
{
  /* Hier sollen Sie Ihren Code einfuegen. */
}

void print_string(string* str)
{
  /* Hier sollen Sie Ihren Code einfuegen. */
}

int main(void)
{
  string* test = create_string("Hallo String!");
  print_string(test);
  free_string(test);
  return 0;
}
```