Sicheres Programmieren in C

Ausblick

Heute lernen wir in Hinblick auf sichere Programmierung ...

- wie wir uns vor *spatialen Fehlern* schützen können
- wie wir uns vor *temporalen Fehlern* schützen können
- worauf man beim API-Design achten sollte
- wie wir Routineaufgaben mit der libc besser bewältigen

Grundlegende Überlegungen

Solltet ihr sicherheitskritischen Code in C schreiben?

Grundlegende Überlegungen

Solltet ihr sicherheitskritischen Code in C schreiben? **Nein.**

Grundlegende Überlegungen

Solltet ihr sicherheitskritischen Code in C schreiben? **Nein.**
Aber manchmal muss es halt sein.

Grundlegende Überlegungen

Solltet ihr sicherheitskritischen Code in C schreiben? **Nein.**
Aber manchmal muss es halt sein.
Und wenn ihr es machen müsst, dann wenigstens richtig.

Spatiale Probleme: Überläufe

```
1  int konto, kreditrahmen = -1000;
2  enum { ABHEBUNG_FEHLSCHLAG, ABHEBUNG_ERFOLG };
3
4  int abheben(int betrag) {
5      if (betrag < 0)
6          return (ABHEBUNG_FEHLSCHLAG);
7
8      if (konto - betrag < kreditrahmen)
9          return (ABHEBUNG_FEHLSCHLAG);
10
11     konto -= betrag;
12     return (ABHEBUNG_ERFOLG);
13 }
```

Spatiale Probleme: Überläufe

Was ist das Problem?

Spatiale Probleme: Überläufe

Was ist das Problem?

- `konto - betrag` kann überlaufen, wenn `betrag` nah an `INT_MAX` ist.

Spatiale Probleme: Überläufe

Was ist das Problem?

- `konto - betrag` kann überlaufen, wenn `betrag` nah an `INT_MAX` ist.
- dann wird `konto - betrag` eine sehr große Zahl
- und der Kredit-Check lässt uns Geld abheben

Was tun wir jetzt?

Überläufe: C23 hilft!

```c
#include <stdckdint.h>

int abheben(int betrag) {
    int konto_neu;

    if (betrag < 0 || ckd_sub(&konto_neu, konto, betrag)
        || konto_neu < kreditrahmen)
        return (ABHEBUNG_FEHLSCHLAG);

    konto = konto_neu;
    return (ABHEBUNG_ERFOLG);
}
```

Spatiale Probleme: Zugriff außerhalb der Array-Grenzen

```
1 enum {ITEM_COUNT = 100};
2 struct item items[ITEM_COUNT];
3
4 struct item *get_item(int id) {
5     return (&items[id]);
6 }
```

Spatiale Probleme: Zugriff außerhalb der Array-Grenzen

```
1 enum {ITEM_COUNT = 100};
2 struct item items[ITEM_COUNT];
3
4 struct item *get_item(int id) {
5     if (id < 0 || ITEM_COUNT <= id)
6         return (NULL);
7
8     return (&items[id]);
9 }
```

Spatiale Probleme: Typverwirrung

```
1 static int str_predicate(void *a, void *b) {
2     return strcmp((const char *)a, (const char *)b);
3 }
4
5 void sort_string(const char *A[], size_t n) {
6     qsort(A, n, sizeof *A, str_predicate);
7 }
```

Spatiale Probleme: Typverwirrung

```
1 static int str_predicate(void *a, void *b) {
2     return strcmp(*(const char **)a, *(const char **)b);
3 }
4
5 void sort_string(const char *A[], size_t n) {
6     qsort(A, n, sizeof *A, str_predicate);
7 }
```

Falscher Typ: Argumente des Prädikats sind Zeiger auf Array-Elemente, nicht Array-Elemente selbst. Bei Arbeit mit `void`-Zeigern immer genau auf korrekte Typen achten! Kompiler hilft nicht!

Spatiale Probleme: falsche Allokationsgröße

```
1  struct item **allocate_items(size_t n) {
2      struct item **array = malloc(n * sizeof *array);
3      if (array == NULL) { ... }
4
5      for (size_t i = 0; i < n; i++) {
6          array[i] = malloc(sizeof array[i]);
7          if (array[i] == NULL) { ... }
8      }
9
10     return (array);
11 }
```

Spatiale Probleme: falsche Allokationsgröße

Was ist da alles falsch?

Spatiale Probleme: falsche Allokationsgröße

Was ist da alles falsch?

- Erste Allokation: `n * sizeof *array` kann überlaufen!
- dann wird viel zu wenig Speicher alloziert
- aber das Programm glaubt, das Array hätte die richtige Größe

Spatiale Probleme: falsche Allokationsgröße

Was ist da alles falsch?

- Erste Allokation: `n * sizeof *array` kann überlaufen!
- dann wird viel zu wenig Speicher alloziert
- aber das Programm glaubt, das Array hätte die richtige Größe
- Zweite Allokation: es wird Speicher der falschen Größe alloziert
- effektiv `sizeof(struct item *)` statt `sizeof(struct item)`
- Kompiler sollte dich hier warnen, trotzdem wachsam sein!

Spatiale Probleme: falsche Allokationsgröße

```
1  struct item **allocate_items(size_t n) {
2      struct item **array = calloc(n, sizeof *array);
3      if (array == NULL) { ... }
4
5      for (size_t i = 0; i < n; i++) {
6          array[i] = malloc(sizeof *array[i]);
7          if (array[i] == NULL) { ... }
8      }
9
10     return (array);
11 }
```

Spatiale Probleme: verlorene Terminatoren

```c
void zeilenumbruch_dranhaengen(char *str) {
    /* gehe zum Ende des Strings */
    for (size_t i = 0; str[i] != '\0'; i++)
        ;

    str[i] = '\n';   /* haenge Zeilenumbruch an */
}
```

Spatiale Probleme: verlorene Terminatoren

```c
void zeilenumbruch_dranhaengen(char *str) {
    /* gehe zum Ende des Strings */
    for (size_t i = 0; str[i] != '\0'; i++)
        ;

    str[i] = '\n';   /* haenge Zeilenumbruch an */
    str[i+1] = '\0'; /* restauriere Null-Terminator */
}
```

Mögliche Lösung, falls `str` genug Platz bietet.
Das zu garantieren ist schwierig, und macht euren Code fragil.

Spatiale Probleme: verlorene Terminatoren

```c
char *zeilenumbruch_dranhaengen(const char *str) {
    char *res;

    asprintf(&res, "%s\n", str);

    return (res);
}
```

Bessere Lösung: Kopie des Strings mit Zeilenumbruch erzeugen. String muss nachher freigegeben werden.
Mehr `asprintf()` wagen!

Temporale Probleme: uninitialisierter Speicher

...

Temporale Probleme: erloschener Speicher

```
1 struct liste { struct liste *nachfolger; int wert; };
2
3 int glied_entfernen(struct liste **l) {
4     free(*l);
5     *l = (*l)->nachfolger;
6     return ((*l)->wert);
7 }
```

Temporale Probleme: erloschener Speicher

```
1  struct liste { struct liste *nachfolger; int wert; };
2
3  int glied_entfernen(struct liste **l) {
4      struct liste *alt = *l;
5      int wert = (*l)->wert;
6
7      *l = (*l)->nachfolger;
8      free(alt);
9      return (wert);
10 }
```

Temporale Probleme: doppelt freigegebener Speicher

```
1 struct canvas *canvas = new_canvas(width, height);
2
3 ...
4
5 free_canvas(canvas);
6
7 ...
8
9 free_canvas(canvas);  // autsch
```

Temporale Probleme: geleckter Speicher

```
1 void paint_all(int color) {
2     struct canvas *canvas = new_canvas(width, height);
3
4     for (size_t i = 0; i < width; i++)
5         for (size_t j = 0; j < height; j++)
6             canvas->pixels[i*width + j] = color;
7
8     show_canvas(canvas);
9 }
```

Temporale Probleme: geleckter Speicher

```
1  void paint_all(int color) {
2      struct canvas *canvas = new_canvas(width, height);
3
4      for (size_t i = 0; i < width; i++)
5          for (size_t j = 0; j < width; j++)
6              canvas->pixels[i*width + j] = color;
7
8      show_canvas(canvas);
9      free_canvas(canvas);
10 }
```

Allen Speicher den ihr alloziert müsst ihr wieder freigeben.

Lösungsstrategien: allgemeine Sorgfalt

- überlegt was bei jeder Operation passieren kann.
- lest die Doku, bevor ihr eine Funktion aufruft.
- macht den Code einfach und offensichtlich
- denkt ihr zu kompliziert, wird der Code schwer zu findende Fehler haben
- stupider Code ist einfacher zu verstehen und wahrscheinlich korrekter

Lösungsstrategien: Invarianten definieren

Definiert und dokumentiert *Invarianten*. Das sind Eigenschaften, die eure Daten zu einem gewissen Zeitpunkt erfüllen müssen.

Lösungsstrategien: Invarianten definieren

Definiert und dokumentiert *Invarianten*. Das sind Eigenschaften, die eure Daten zu einem gewissen Zeitpunkt erfüllen müssen.
Typische Arten von Invarianten:

Vorbedingung Muss erfüllt sein, wenn die Funktion aufgerufen wird

Nachbedingung Wird von eurer Funktion erfüllt, wenn sie zurückkehrt

Schleifen-Invariante Gilt am Anfang jedes Schleifendurchlaufs

Lösungsstrategien: Invarianten definieren

```
1  // Vorbed.: A hat *len Elemente, kann cap Elemente halten
2  // Nachbed.: A hat *len Elemente.  Falls Rückgabe wahr, ist
3  //     x eines von diesen, sonst schlug Einfügung fehl.
4  bool menge_einfuegen(int *A, size_t *len, size_t cap, int x) {
5      for (size_t i = 0; i < *len; i++) // Invar.: 0 <= i < *len
6          if (A[i] == x) return (true);
7
8      if (*len >= cap) return (false);
9
10     A[i] = elem;
11     *len += 1;
12     return (true);
13 }
```

Lösungsstrategien: Invarianten definieren (für Paranoide)

```
1 bool menge_einfuegen(int *A, size_t *len, size_t cap, int n) {
2     assert(A != NULL);
3     assert(*len <= cap);
4
5     ...
6 }
```

Prüft Invariante mit *Zusicherungen*. Euer Programm stürzt sofort ab, wenn diese nicht erfüllt sind. Mit Kompileroption `-DNDEBUG` können diese abgeschaltet werden (bessere Performanz).

Lösungsstrategien: Allokation und Deallokation zusammen

```
1 struct A *a = new_A();
2 if (a == NULL)
3     /* Fehlerbehandlung ... */
4
5 mach_was_mit(a);
6 free_A(a);
```

Diese Sequenz sollte so knapp hintereinander sein.

Lösungsstrategien: Allokation und Deallokation zusammen

```
1          struct A *a = new_A();
2          if (a == NULL)
3              goto fail1;
4
5          struct B *b = new_B();
6          if (b == NULL)
7              goto fail2;
8
9          /* irgendwas mit a und b hier machen */
10
11         free_B(b);
12 fail2:  free_A(a);
13 fail1:  return ...; /* Fehler-Return hier */
```

Lösungsstrategien: existierende Dinge nutzen

- Sanitizer für undefiniertes Verhalten (Vorlesung 8)
- sanitizer für Speicherprobleme (Vorlesung 8)
- Option `-ftrapv` (Absturz bei Überlauf)
- schreibt komplexe Datenstrukturen nicht selbst
- benutzt fertige Bibliotheken
- ggf. möglich: Boehm-Garbage-Collector

Sichere Stringverarbeitung: asprintf

```c
#include <stdio.h>

int asprintf(char **ret, const char *format, ...);

...
char *name;
asprintf(&name, "%s %s", vorname, nachname);
```

Wie `printf`, aber es wird ein Puffer mit `malloc` alloziert und mit dem erzeugten String gefüllt.

Sichere Stringverarbeitung: open_memstream

```c
#include <stdio.h>

FILE *open_memstream(char **buf, size_t *len);

...
char *buf = NULL;
size_t len = 0;
FILE *stream = open_memstream(&buf, &len);
fprintf(stream, "%s %s", vorname, nachname);
fclose(stream);
```

Erzeugt ein `FILE`, das in einen Puffer schreibt. Der Puffer wird beim Schließen finalisiert und ist dann ein String. Muss nach Nutzung mit `free()` freigegeben werden.