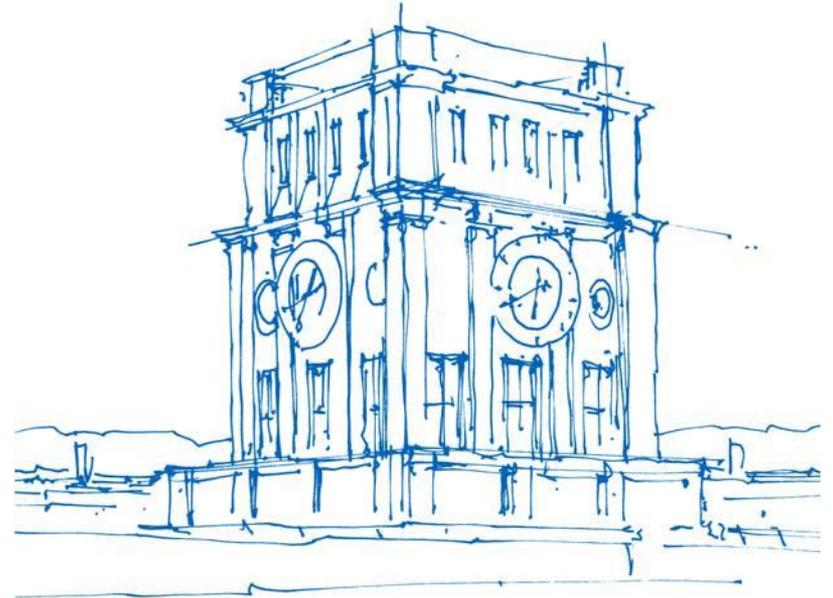


Grundlagenpraktikum: Rechnerarchitektur

SoSe 2024

~ *Danial Arbabi*

danial.arbabi@tum.de



Uhrenturm der TUM

Zulip-Gruppen

<https://zulip.in.tum.de/#narrow/stream/2276-GRA-Tutorium---Gruppe-29>

Gruppe 29

FR 12:00



Gruppe 32

FR 15:00



<https://zulip.in.tum.de/#narrow/stream/2279-GRA-Tutorium---Gruppe-32>

Tutoriums-Website



<https://home.in.tum.de/~arb/>

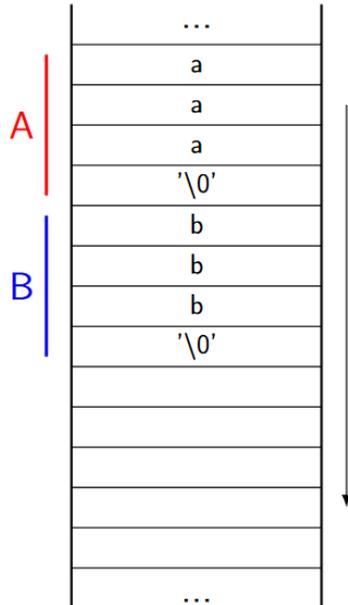
Disclaimer:

*Dies sind keine offiziellen
Materialien, somit besteht keine
Garantie auf Korrektheit und
Vollständigkeit.
Falls euch Fehler auffallen, bitte
gerne melden.*

Wiederholung

Buffer Overflows

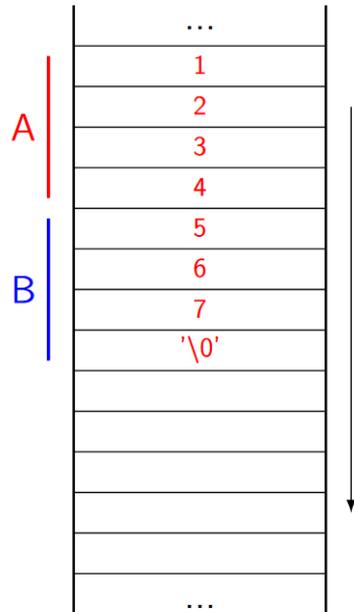
- Lese- oder Schreibzugriffe auf Speicher jenseits des eigentlichen Buffers (Speicherbereichs)



```
1 char A[] = "aaa";  
2 char B[] = "bbb";  
3 strcpy(A, "1234567");
```

Buffer Overflows

- Lese- oder Schreibzugriffe auf Speicher jenseits des eigentlichen Buffers (Speicherbereichs)



```
1 char A[] = "aaa";  
2 char B[] = "bbb";  
3 strcpy(A, "1234567");
```

Buffer Overflows

- Sichere Programmierung:

```
char buffer[8];  
  
// strncpy mit Längenüberprüfung  
strncpy(buffer, "1234567", sizeof(buffer) - 1) ;  
// NULL-Terminal setzen  
buffer[sizeof(buffer) - 1] = 0;
```

Memory Leaks

- Speicher wird nicht benutzt, aber auch nicht freigegeben

```
char* buffer = malloc(128);

if (!buffer)
    return; // Fehlerbehandlung

/* buffer benutzen*/

free(buffer);

/* buffer NICHT MEHR BENUTZEN*/
```

Format String Injection

- printf() benutzt pro Format Specifier einen Parameter
- Parameter sind automatisch Register und dann der Stack
- Angreifer kann Format Specifier selbst einfügen und Speicher / Register leaken, da diese als Parameter interpretiert werden

```
// name kann Format String Injection vom Angreifer enthalten
printf("Hello ");printf(name);printf("!\n")

// Richtig:
printf("Hello %s!\n", name);
```

Undefined behavior

- Dereferenzierung eines Nullpointers
- Double free
- Use after free
- Lesen uninitialisierter Variablen
- Signed Integer Overflow
- Pointer cast in strengeres Alignment
- ...

Sanitizer

- Verschieden Sanitizer können über Compilerflags aktiviert werden
 - **-fsanitize=address** für Buffer Overflows und Dangling Pointer
 - **-fsanitize=leak** für Memory Leaks
 - **-fsanitize=undefined** für Undefined Behavior
 - Aber:
 - Performance-Einschränkungen
 - Erkennt nicht alle Fehler
 - Erschwert Debugging mit anderen Tools
- => TESTEN

Zusammenfassung

Wie programmiere ich sicher?

- Arraygrenzen beachten
- Speicherverwaltung beachten
- Keine unsicheren Funktionen benutzen (s. Manpage)
- Vermeidung undefinierter Variablen
- Vermeidung von undefined behavior
- Format-Specifier benutzen

Aufgabe

T3.2 Commandline Parsing
(Artemis)

Commandline Parsing mit getopt()

```
int getopt(int argc, char *argv[], const char *optstring);
```

- argc und argv aus der main-Funktion
- const char *optstring: Alle unterstützten Optionen
 - : => Argument **benötigt**
 - :: => Argument **optional**
 - <nichts> => **Kein** Argument
- Beispiele:
 - hta:b:c::
 - -h und -t
 - a 1.0 oder a1.0 und b5
 - c5 oder c

Commandline Parsing mit getopt()

```
int getopt(int argc, char *argv[], const char *optstring);
```

- Rückgabewert:
 - Nächster options-Character
 - -1 bei Ende der Optionen
 - ?: falls Fehler

Umwandlung von Strings zu Zahlen

- strtol()
- strtoul()
- strtoull()
- strtod()
- strtod()
- KEIN atoi() und atof() benutzen, da keine Fehlererkennung;

```
long strtol(const char *ptr, char **endptr, int base);  
double strtod(const char *ptr, char **endptr);
```

- const char *ptr: zu konvertierender String
- char **endptr: Adresse eines Pointers, der auf den fehlerhaften Teil des ptr Strings zeigen wird
- int base: Zahlenbasis

Umwandlung von Strings zu Zahlen

Beispiel:

```
int a; // Zu ändernde Variable
errno = 0; // Errno für Fehlerbehandlung
char* endptr; // Endptr für Fehlerbehandlung

a = strtol(currentOpt, &endptr, 10);

if (*endptr != '\0' || errno) {
    fprintf(stderr, "Invalid number\n");
    /* Weitere Fehlerbehandlung bzgl errno Wert*/
    return EXIT_FAILURE;
}
```

Quellen

- Video: „Sichere Programmierung“ auf Artemis