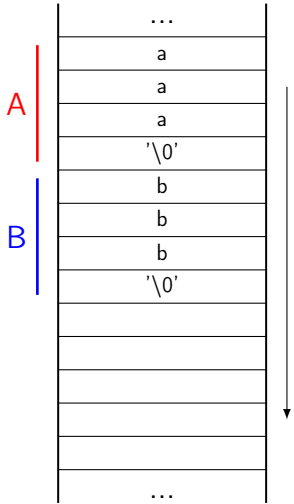
Buffer Overflows

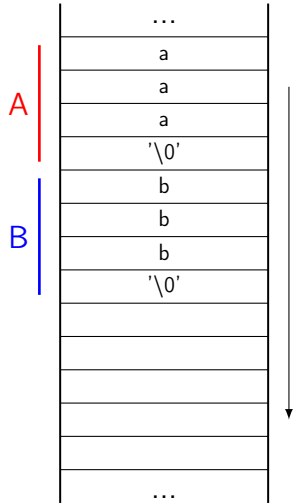
- Buffer sind in C oft notwendig
  - Beispiel: Nutzereingaben entgegennehmen
- Buffer Overflow: Lesende bzw. schreibende Zugriffe über die Grenzen des Buffers hinaus
- Häufig auftretende Sicherheitslücke
  - Beispielhafter Angriff: Überschreiben der Rücksprungadresse
  - Ermöglicht Sprünge zu beliebigen Funktionen im Programm
- Overflows von nur einem Byte können die Sicherheit bereits beeinträchtigen

Buffer Overflows



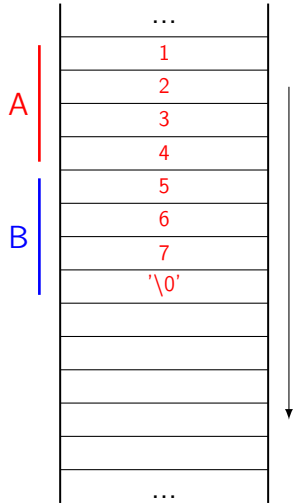

```
1    char A[] = "aaa";
2    char B[] = "bbb";
```

Buffer Overflows




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

Buffer Overflows




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

Quiz: Buffer Overflows

Welches dieser Code-Snippets kann bzw. wird einen Buffer Overflow auslösen?

☐
```
char a[10];
strcpy(a, "123456789");
```

☐
```
char b[4];
strcpy(b, "erap");
```

☐
```
char c[10];
scanf("%10s", c);
```

☐
```
char d[] = {'a'};
strcpy(d, "a");
```

Segmentation Faults

- Verletzung des Speicherschutzes, zum Beispiel durch..
  - Schreibenden Zugriff auf read-only Daten
  - Zugriff auf Daten mit fehlenden Berechtigungen
    - Beispiel: Kernel-Daten als Benutzer
  - Null-Pointer Dereferenzierung
  - Fehlerhafte Speicherzugriffe durch Verletzung der Calling Convention
  - (Viele weitere Möglichkeiten)

Segmentation Faults

- Mögliche Ansätze zur Vermeidung:
  - Pointer vor Benutzung überprüfen
    - Beispiel: `fopen` gibt bei Fehler `NULL` zurück
  - Fehler bei Einhaltung der Array-Grenzen überprüfen
    - Off-By-One Fehler
    - Hardcoding von Arraygrößen
  - Beachtung des Nullterminals bei C-Strings überprüfen
    - Vergessen des Nullterminals
    - Speicher zu klein für Nullterminal

Inhärent Unsichere Funktionen

- Wenn möglich Funktionen verwenden, die Schutzmechanismen erzwingen
  - Beispiel: `memcpy(void* dest, const void* src, size_t n);`
  - `n` gibt explizit Größe der Daten an
- C-Standardbibliothek beherbergt viele potenziell gefährliche Funktionen

Unsichere Funktionen

gets

- Paradebeispiel: `gets(char* buf);`
  - (Ab C11 nicht mehr Teil des Standards)
  - Liest Daten in `buf`, bis `EOF` oder `'\n'` erkannt wird
  - Zitat `man gets`: "*Never use gets().*"
- Auch fehleranfällig: `scanf()`
  - Angabe eines Format-Strings kann maximale Zeichen einschränken
    - Beispiel: `scanf("%5s", buf)`
  - ..aber `scanf("%s", buf)` ist weiterhin erlaubt
- Stattdessen immer verwenden: `fgets(char* dest, int n, FILE* stream)`
  - Höchstens `(n-1)` Zeichen werden eingelesen
  - String in `dest` wird nullterminiert

Unsichere Funktionen

strcpy

- `strcpy(char* dest, const char* src);`
  - Kopiert `src` nach `dest`
  - Inkludiert auch das Nullterminal von `src`
  - Problem: `size(src) > size(dest)`
- Bessere Alternative: `strncpy(char* dest, const char* src, size_t n);`
  - Parameter `n` bestimmt, wie viele Bytes maximal kopiert werden
  - Resultat ist nur nullterminiert, wenn ein Nullterminal in den `n` Bytes von src existiert!
    - Beispiel: `char d[3]; char* s = {'1', '2', '3'};`
    - `strncpy(d, s, 3)` ergibt `d = {'1', '2', '3'}`

Unsichere Funktionen

strcat

- `strcat(char* dest, const char* src);`
  - Hängt `src` an `dest`
  - Überschreibt Nullterminal von `dest`, Resultat ist *immer* nullterminiert
  - Problem: `strlen(dest) + strlen(src) >= size(dest)`
- Bessere Alternative: `strncat(char* dest, const char* src, size_t n);`
  - Parameter `n` bestimmt, wie viele Bytes von `src` maximal verwendet werden
  - Sonderregel: ist `size(src)` $\geq$ `n`, muss `src` kein Nullterminal enthalten
  - Resultat ist auch hier immer nullterminiert
    - Wenn also `size(src)` $\geq$ `n`, werden `n+1` Bytes in `dest` geschrieben
  - Vorsicht auch hier: Es muss immer $size(dest) > strlen(dest) + n$ gelten, sonst vielleicht Bufferoverflow!

Codebeispiel

```c
int main() {
    char *buf = malloc(40 * sizeof(char));
    printf("Please enter your name:\n");
    scanf("%40s", buf);
    printf("Hello "); printf(buf); printf("!\n");

    if (!strlen(buf)) {
        printf("You didn't enter your name!\n");
        return 1;
    } else if (strlen(buf) > 20) {
        printf("You have a really long name, %s!\n", buf);
        free(buf);
    }
    printf("Thank you for introducing yourself, %s!\n", buf);
    free(buf);
    return 0;
}
```

Quiz: Fehler im Code I

Gesamter Code in Beschreibung unten

Welche Fehler verbergen sich in Zeile 1–5 im gegebenen Code?

```
1 int main() {
2     char *buf = malloc(40 * sizeof(char));
3     printf("Please enter your name:\n");
4     scanf("%40s", buf);
5     printf("Hello "); printf(buf); printf("!\n");
```

- ☐ Die `main`-Funktion nimmt keine Parameter entgegen
- ☐ `malloc` alloziert durch `sizeof(char)` zu viel Speicher
- ☐ Das Ergebnis von `malloc` wird nicht gecastet
- ☐ Es wird nicht geprüft, ob `malloc` erfolgreich war

Überprüfung von `malloc()`

- Anfordern von Speicher mit `malloc` kann fehlschlagen
- Rückgabewert ist dann `NULL`
- Benutzung dieses Pointers ist undefiniert!

- Verhindern dieses Fehlers:
  - Überprüfen, ob `malloc` null zurückgibt

Quiz: Fehler im Code II

Gesamter Code in Beschreibung unten

Welche Fehler verbergen sich in Zeile 1–5 im gegebenen Code?

```
1 int main() {
2     char *buf = malloc(40 * sizeof(char));
3     printf("Please enter your name:\n");
4     scanf("%40s", buf);
5     printf("Hello "); printf(buf); printf("!\n");
```

- ☐ `printf` darf nicht mehrmals pro Zeile verwendet werden
- ☐ Strings für `printf` müssen zwingend mit `\n` enden
- ☐ `scanf("%40s", buf)` schreibt evtl. zu viele Zeichen in `buf`
- ☐ Bei `printf(buf)` kann man mit bestimmten Eingaben Speicher lesen und schreiben

Buffer Overflow

Off-By-One

- Im Codebeispiel: `scanf("%40s", buf)` ist Off-By-One Buffer Overflow
  - `scanf("%40s", buf)` liest bis zu 40 Zeichen und hängt ein Nullbyte an
  - $\Rightarrow$ genau einen Wert über den Buffer hinaus geschrieben
- Um den Fehler zu vermeiden kann man:
  - Ein Byte weniger als die Länge des Buffers in an `scanf()` übergeben
  - Oder deutlich besser: `fgets()` verwenden

Format String Injection

- `printf` benutzt pro Format Specifier einen Parameter
- Parameter sind automatisch immer die Register und danach der Stack
- Bei Einlesen von Format Specifiern werden Register/Stack als Parameter interpretiert!
  - User kann Speicher mit `%x`, `%s` etc. leaken oder mit `%n` schreiben
- Vermeidung: Kombinieren der printf-Aufrufe und Benutzung eines Formatstrings: `printf("Hello %s!\n", buf);`

Quiz: Fehler im Code III

Gesamter Code in Beschreibung unten

Welche Fehler verbergen sich in Zeile 7–17 im gegebenen Code?

```
7       if (!strlen(buf)) {
8           printf("You didn't enter your name!\n");
9           return 1;
10      } else if (strlen(buf) > 20) {
11          printf("You have a really long name, %s!\n", buf);
12          free(buf);
13      }
14      printf("Thank you for introducing yourself, %s!\n", buf);
15      free(buf);
16      return 0;
17  }
```

- ☐ Wenn Zeile 9 erreicht wird, wird `buf` nicht freigegeben
- ☐ `strlen(buf)` ist immer ≥ 1 aufgrund des Nullbytes
- ☐ Das Ergebnis von `malloc` wird nicht gecastet
- ☐ Bei `free` fehlt die Länge des freizugebenden Speichers

Memory Leak

- Angeforderter Speicher im Heap wird nicht wieder freigegeben
- Hier nicht ganz so problematisch, da Programm direkt beendet wird
- Führt in größeren Programmen allerdings zu extremem Speicherbedarf!
- Vermeidung von Memory Leaks durch Freigabe von unbenötigtem Speicher
  - Vor `return`-Statements immer alle durch `malloc`, `calloc` etc. belegten Speicherbereiche freigeben!
  - Ausnahme: Rückgabe von heapalloziertem Speicher

Quiz: Fehler im Code IV

Gesamter Code in Beschreibung unten

Welche Fehler verbergen sich in Zeile 7–17 im gegebenen Code?

```
7       if (!strlen(buf)) {
8           printf("You didn't enter your name!\n");
9           return 1;
10      } else if (strlen(buf) > 20) {
11          printf("You have a really long name, %s!\n", buf);
12          free(buf);
13      }
14      printf("Thank you for introducing yourself, %s!\n", buf);
15      free(buf);
16      return 0;
17  }
```

- ☐ `free(buf)` setzt `buf` auf `NULL`
- ☐ Ein `return` aus `main` muss immer mit `return 0` erfolgen
- ☐ `buf` wird evtl. zweimal freigegeben
- ☐ `buf` wird benutzt, nachdem `free` darauf aufgerufen wurde

Use after Free, Double Free

- Problem 1:
  - User gibt 20 Zeichen ein
  - `buf` wird freigegeben
  - `buf` wird in `printf` benutzt $\Rightarrow$ Use after Free
- Problem 2:
  - Nach obigem Programmablauf: `buf` wird erneut freigegeben $\Rightarrow$ Double Free
- Diese Fehler fallen in die Kategorie **undefiniertes Verhalten**.

Undefiniertes Verhalten (Undefined Behavior, UB)

$\Rightarrow$ Programm weicht vom C-Standard ab

- Beispiele für UB:
  - Dereferenzierung von Nullpointer
  - Double Free
  - Use after free
  - Lesen uninitialisierter Variablen
  - Signed Integer Overflow
  - Shift um Länge eines Integerwerts (oder mehr oder negativ)
  - Flushen eines Inputstreams, z.B. `fflush(stdin)`
  - Fehler bei Pointercasts (meist unnötig/obsolet)
  - ...
- Bugfix hier: Entfernen des ersten `free` oder Beenden der Ausführung vor Benutzung von `buf` (je nach Sinn des Programms)

Vermeidung von Fehlern – Sanitizer

- Verschiedene Sanitizer können über Compilerflags aktiviert werden:
  - `-fsanitize=address` für Buffer Overflows und Dangling Pointer
  - `-fsanitize=leak` für Memory Leaks
  - `-fsanitize=undefined` für Undefined Behavior
- Nachteile der Verwendung der Sanitizer
  - Erschwert Debugging mit anderen Tools
  - Performanz des Programms wird deutlich verringert
  - Erkennen bei Weitem nicht alle Fehler
    - Testen, Testen, Testen!
  - Funktioniert nicht bei handgeschriebenem Assembly

Codebeispiel (korrigiert)

```c
int main() {
    size_t bufsize = 40;
    char *buf = malloc(bufsize * sizeof(char));
    if (!buf) { // Ueberpruefung von malloc
        printf("Malloc failed. Exiting.\n");
        exit(1);
    }
    printf("Please enter your name:\n");
    fgets(buf, bufsize, stdin); // fgets statt scanf (never use scanf)
    printf("Hello %s!\n", buf); // Keine Formatstringluecke

    if (!strlen(buf)) {
        printf("You didn't enter your name!\n");
        free(buf); // kein Memory Leak
        return 1;
    } else if (strlen(buf) > 20) {
        printf("You have a really long name, %s!\n", buf);
    }
    printf("Thank you for introducing yourself, %s!\n", buf); // kein UAF
    free(buf); // kein Double Free
    return 0;
}
```