Undefiniertes Verhalten

Proseminar Effiziente Programmierung in C

Konstanze Reupert

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

08.Juli.2020

Gliederung (Agenda)

Was ist undefiniertes Verhalten? [15] [17] [22]

- nicht durch den C-Standard definiertes Verhalten
    - Compiler muss nur definierte Fälle prüfen
    - „graue Bereiche“, in denen Compiler Entscheidungsfreiheit hat
    - unvorhersehbares Verhalten
- potentielle Ursache für (schwerwiegende) Fehler

"Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose." [19]

engl.: „undefined behavior“ = UB

C-Standard [15]

> „Verhalten im Fall der Nutzung eines nicht portablen oder fehlerhaften Programms oder fehlerhaften Daten, für das der Standard keine Anforderungen erhebt“ [7] - *C99-Standard*

- zentrales Regelwerk als Orientierung
- „Der Compiler darf annehmen x“
  - UB ist alles, was nicht x ist [23] [24]
  - indirekte Vorlage für UB ⇒ "Lizenz zur Willkür"[17]
- über 100 Arten von UB

Kategorien von Verhalten [12] [5] [28] [29]

- definiert
  - deterministisch, streng konform
  - eindeutiger Ausgang (gemäß des C-Standards)
- implementations-definiert
  - verschiedene dokumentierte Ausgänge
- unspezifiziert
  - beliebig gewählter Ausgang (nicht dokumentiert)
- undefiniert
  - nichtdeterministisch, keine Einschränkungen
  - verschiedene, unvorhersehbare Ausgänge

Mögliche Folgen von UB [17]

- Programmabsturz
- Generierung von zufälligem/willkürlichem Code
- Beschädigung von Daten
- unbemerktes falsches Verhalten
  - UB als „Zeitbombe“ [22]
  - Programm läuft eine Zeit lang wie erwartet

⇒ im Allgemeinen schwer vorherzusagen

Signed Integer Overflow [34]

- Überschreitung des Integer-Wertebereichs
- Unsigned Integer Überlauf ist **definiert** (2-Komplement)
- Signed Integer Überlauf ist **undefiniert**

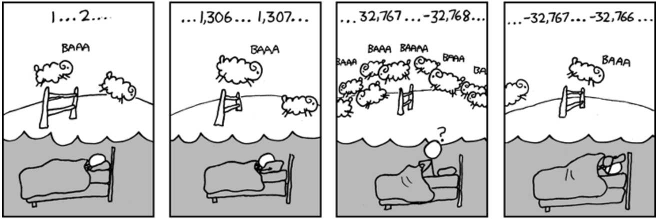
```c
int main(int n) // Ueberlauf, falls n = INT_MAX
{
    for (int i = 1; i <= n; ++i)
    {
        printf("%d. Durchlauf", i);
    }
    return 0;
}
```

- evtl. Endlosschleife, evtl. Beginn bei INT_MIN ("Wraparound")
- unterschiedliche Repräsentation von negativen Werten

Signed Integer Overflow

Betrag&Vorzeichen	1-Komplement	2-Komplement
000 = 0	000 = 0	000 = 0
001 = 1	001 = 1	001 = 1
010 = 2	010 = 2	010 = 2
011 = 3	011 = 3	011 = 3
100 = -**0**	100 = -**3**	100 = -**4**
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

Tabelle: Überlauf bei einer 3-bit-Datenstruktur [33]



Bildquelle: `https://sergioprado.org/undefined-behavior-em-linguagem-c/`

Nicht-initialisierte Variablen [15]

```c
int random()
{
    int x;  // nicht initialisiert
    printf("x = %d\n", x); // undefiniert
    return 0;
}
```

```
In function 'random':
warning: 'x' is used uninitialized in this function [-Wuninitialized]
Compiler returned: 0
```

Nicht-initialisierte Variablen [15]

- mögliches Verhalten
    - Initialisierung mit zufälligem Wert
    - optimierende Entfernung der Variable

Beispiel 2008 (OpenSSL, Debian):
berechenbarer Zufallsgenerator zur Verschlüsselung
`https://www.debian.org/security/2008/dsa-1571`

Index-Out-Of-Bounds [27]

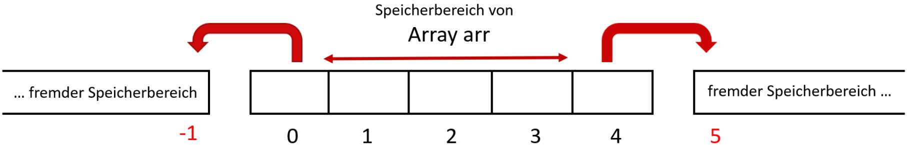


Abbildung: Zugriff auf Index < 0 oder $\geq$ Array-Länge

Index Out Of Bounds

```c
int main()
{
    int arr[5]= {0,1,2,3,4};

    // Zugriff auf arr[5] im letzten Durchlauf
    for (int i=0; i<=5; i++)
        printf("%d \n", arr[i]);
}
```

2020 CWE TOP 25 der gefährlichsten Software-Schwachstellen[8]

- Platz 2: Out-Of-Bounds-Write
- Platz 4: Out-Of-Bounds-Read

Buffer Overflow ("Pufferüberlauf") [13]

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

void func(char *name)
{
    char buf[100];
    strcpy(buf, name); // Overflow, falls |name|>100
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
   func(argv[1]); // Uebergabe eines Namens
   return 0;
}
```

Buffer Overflow [13]

$|A^{100}B^4C^4| = 108 > 100$

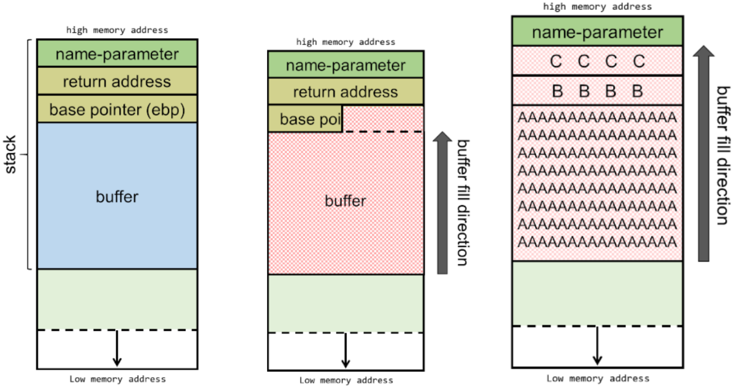


Abbildung: Überschreibung der Rücksprungadresse

Dangling Pointer ("Hängender Zeiger") [6] [26]

- Zeiger verweist auf ein bereits freigegebenes Objekt
- Verwendung verursacht Use-After-Free-Problem

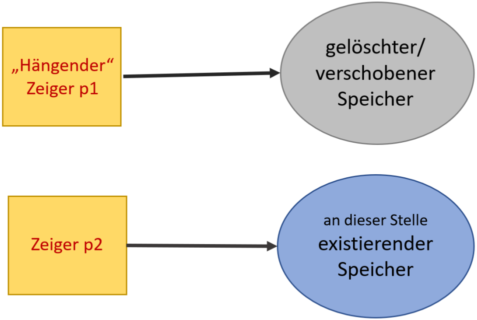


Abbildung: Dangling Pointer

Use-After-Free & Double-Free [30] [31] [9] [10]

- Use-After-Free
  - Verwendung eines Hängenden Zeigers
  - Bsp.: veraltete Hyperlinks
  - 70% der Probleme bzgl. Speichersicherheit bei Microsoft und Chrome wegen Use-After-Free [2] [1]

- Double-Free
  - doppelte Freigabe eines Speicherbereiches
  - Aufruf von free() mit demselben Argument

Use-After-Free & Double-Free [30]

```c
int main()
{
    int* p1 = (int*) malloc(sizeof(int));
    int* p2 = (int*) malloc(sizeof(int));

    *p1 = 8;
    *p2 = 7;

    // Ueberschreibung von p1 mit p2
    p1 = p2; // zeigen auf denselben Speicherbereich

    // doppelte Freigabe des Speichers wegen p1 = p2
    free(p1); // Use-After-Free
    free(p2); // Double-Free
}
```

Dereferenzierung eines NULL-Zeigers [32]

- Zugriff auf den Wert eines NULL-Zeigers
- Problem: Zeiger verweist auf nicht existierenden Speicher

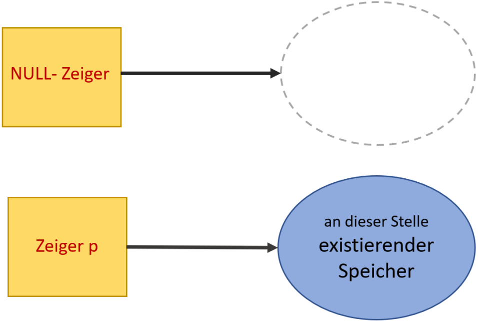


Abbildung: NULL-Zeiger

Dereferenzierung eines NULL-Zeigers [35]

- Beispiel 2009: Linux Kernel [35]
  - Entfernung einer NULL-Prüfung durch den Compiler

```c
int main()
{
    int *p = NULL;
    int x = *p;     // NULL-Zeiger Dereferenzierung

    if(!x)
        printf("%d", *p); // nicht ausgefuehrt
    return 0;
}
```

- Lösung: NULL-Prüfung vor Dereferenzierung

Auslöser von UB

- Annahme, dass UB nicht auftreten kann
- Portierungen
- Compilerwechsel
- Versionsupdates
- Optimierungen
  - Ausnutzung von UB

Signed Integer Overflow [7]

```c
int vergleiche(int x)
{
    return (x+1) > x;
}
int main()
{
    printf("%d\n", (INT_MAX+1) > INT_MAX); // 0
    printf("%d\n", vergleiche(INT_MAX));   // 1

    return 0;
}
```

widersprüchliche Auswertungen auf Optimierungslevel -O2

Speichersicherheit (Memory Safety) [14] [26] [18]

- Auswirkung von Fehlern in Bezug auf den Speicher
  - Lese- und Schreibzugriffe auf sensible Daten
  - Störung/Veränderung/Steuerung des Kontrollflusses

- Unterteilung in ...
  - ... räumliche Fehler (spatial errors)
  - ... zeitliche Fehler (temporal errors)

Speichersicherheit [2]

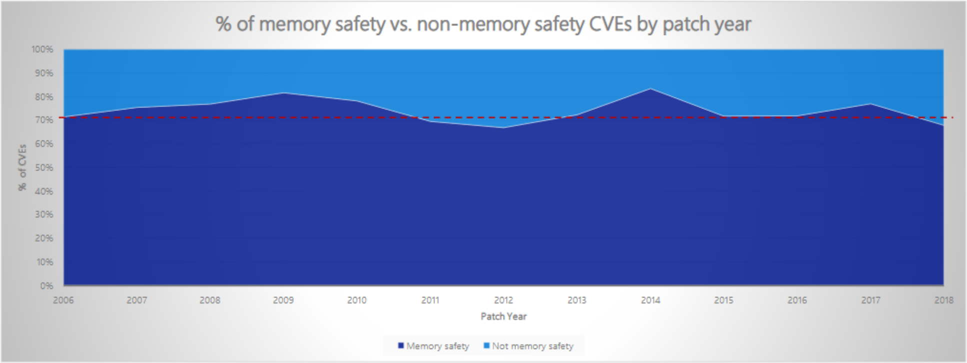
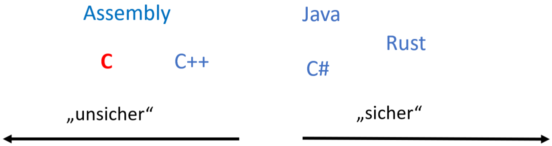


Abbildung: Microsoft: ca. 70% der Probleme betreffen Speichersicherheit

Vergleich zu anderen Sprachen [12] [11] [21]

- „sichere“ Sprachen
  - wegen Exceptions kaum UB, aber auch weniger Freiheiten
  - Fehler werden i.d.R. sofort abgefangen
  - automatische Beachtung der Speichersicherheit
- „unsichere“ Sprachen
  - keine Exceptions ⇒ UB
  - Programm läuft trotz Fehler(n) weiter
  - manuelle Speicherverwaltung mit malloc() und free()
  - mehr Effizienz möglich als bei sicheren Sprachen



Wie kann man UB erkennen? [16] [25] [3] [12] [4]

- Compiler-Warnmeldungen
- statische Analyse
  - z.B. Clang Static Analyzer, Coverty
  - tiefergehende Untersuchung des Quelltextes
- dynamische Analyse
  - z.B. Valgrind
  - Verhalten des Codes zur Laufzeit
  - -fcatch-undefined-behavior (Clang)
  - -fsanitize=undefined (GCC)

⇒ Verlangsamung des Programms durch Analysen

Sanitizer [20] [12] [25]

- Werkzeuge für dynamische Analyse
- Aktivierung von Diagnosen über Compilerflags
    - kein Programmabbruch, lediglich Erkennung von UB
- Arten (je nach Compiler & Zweck)
    - UBSan: „Undefined Behavior Sanitizer“ (Clang & GCC)
    - ASan: „Adress Sanitizer“ (Clang & GCC)
    - MSan: „Memory Sanitizer“ (Clang)
    - TSan: „Thread Sanitizer“ (Clang)

⇒ Auswahlkriterien: Funktion, Kompatibilität, Overhead

Problematiken [16]

- UB vollständig zu eliminieren wäre nicht praktikabel
    - je mehr Flags, desto ineffizienter
    - zu stark veränderter Code ist wenig portabel
- Erstellung von hilfreichen Warnungen
    - Anzahl der Meldungen ⇒ „Wichtiges“ vs. „Unwichtiges“
    - nachvollziehbare Darlegung des Sachverhalts
    - Warnungen „an der richtigen Stelle“
- Zurückverfolgung von Änderungen im Code
    - Nichtdeterminismus, schnelle Verbreitung von UB
    - Optimierungen (evtl. mehrstufig)
    - evtl. falsche positive Rückmeldungen

Warum gibt es UB in C?

- kein Fehler, sondern potentielle Ursache für Fehler [12]
- Schutzziel: Kontextabhängigkeit und Diversität
- mehr Leistung und Effizienz
  - „extreme Designentscheidung“ [24]
  - 30% bis 50% Beschleunigung für Schleifen mit UB [22]

Sollte UB beibehalten werden? [22] [17] [24] [16]

Contra

- unvorhersehbare Gefahren
- Sicherheitslücken
- schwer erkennbar/kontrollierbar
- geringere Portabilität

Pro

- Optimierungsmöglichkeiten
- geringerer Overhead
- geringere Compilerkomplexität
- Diversität & Kompatibilität

⇒ Effizienz oder Sicherheit?

Empfehlungen [22] [24]

- C-Standard kennen und verstehen
- stets nach UB Ausschau halten
- situationabhängige bewusste Entscheidungen treffen
- geeignete Werkzeuge/Hilfsmittel verwenden
- Compiler-Warnungen aktivieren und beherzigen
- Vor- und Nachbedingungen definieren und prüfen

Zusammenfassung

- UB als Folge der Abweichung von den Regeln des C-Standards
- zahlreiche Arten von UB
  - z.B. Overflows, Out-Of-Bounds, Use-After-Free
- häufige Ursache für (schwerwiegende) Fehler
  - Gefährung der Speichersicherheit
  - Unvorhersehbarkeit ⇒ Risiko
- Chance auf mehr Effizienz
  - systemspezifische Optimierungsmöglichkeiten
  - geringerer Overhead
- Sanitizer zur Erkennung von UB

Literatur I

[1] Catalin Cimpanu. "Chrome: 70% of all security bugs are memory safety issues". In: (Mai 2020). eingesehen am 12.06.2021. URL: https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/.

[2] Catalin Cimpanu. "Microsoft: 70 percent of all security bugs are memory safety issues". In: (Nov. 2019). eingesehen am 12.06.2021. URL: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[3] *Clang Static Analyzer*. eingesehen am 26.06.2021. URL: https://clang-analyzer.llvm.org/.

[4] *Coverity Static Analysis*. eingesehen am 26.06.2021. URL: https://de.wikipedia.org/wiki/Coverity_Static_Analysis.

[5] cppreference. "Undefined behavior". In: (Juli 2020). eingesehen am 22.05.2021. URL: https://en.cppreference.com/w/c/language/behavior.

[6] *Dangling pointer*. eingesehen am 05.05.2021. URL: https://en.wikipedia.org/wiki/Dangling_pointer.

[7] Will Dietz u. a. *Understanding Integer Overflow in C/C++*. eingesehen am 13.06.2021. Juni 2012. URL: https://www.cs.utah.edu/~regehr/papers/overflow12.pdf.

[8] Commom Weekness Enumeration. *2020 CWE Top 25 Most Dangerous Software Weaknesses*. eingesehen am 13.06.2021. Aug. 2020. URL: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.

[9] Commom Weekness Enumeration. *CWE-415: Double Free*. eingesehen am 25.06.2021. Juli 2006. URL: https://cwe.mitre.org/data/definitions/415.html.

[10] Commom Weekness Enumeration. *CWE-416: Use After Free*. eingesehen am 25.06.2021. Juli 2006. URL: https://cwe.mitre.org/data/definitions/416.html.

Literatur II

[11] Alex Gaynor. “What is memory safety and why does it matter?” In: (2020). eingesehen am 13.06.2021. URL: `https://www.abetterinternet.org/docs/memory-safety/`.

[12] Barbara Geller und Ansel Sermersheim. “Undefined Behavior is Not an Error”. In: (Sep. 2018). eingesehen am 12.06.2021. URL: `https://www.copperspice.com/pdf/Undefined-Behavior-CppCon-2018.pdf`.

[13] Coen Goedegebure. “Buffer overflow attacks explained”. In: (Nov. 2020). eingesehen am 12.06.2021. URL: `https://www.coengoedegebure.com/buffer-overflow-attacks-explained/`.

[14] Diane Hosfelt. “Fearless Security: Memory Safety”. In: (Jan. 2019). eingesehen am 12.06.2021. URL: `https://hacks.mozilla.org/2019/01/fearless-security-memory-safety/`.

[15] Tobiáš Kamenický. “Undefined Behaviour in the C Language”. In: (2015). eingesehen am 17.06.2021, S. 17–19. URL: `https://is.muni.cz/th/p62tn/Undefined_behaviour_in_the_C_language.pdf`.

[16] Chris Lattner. “What Every C Programmer Should Know About Undefined Behavior 3/3”. In: (Mai 2011). eingesehen am 22.05.2021. URL: `https://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html`.

[17] Raph Levien. “With Undefined Behavior, Anything is Possible”. In: (Aug. 2018). eingesehen am 25.05.2021. URL: `https://raphlinus.github.io/programming/rust/2018/08/17/undefined-behavior.html`.

[18] Rajeev K. Barua Matthew S. Simpson. “MemSafe: ensuring the spatial and temporal memory safety of C at runtime”. In: (Feb. 2012). eingesehen am 25.06.2021. URL: `https://onlinelibrary.wiley.com/doi/full/10.1002/spe.2105`.

[19] *Nasal Demons*. eingesehen am 14.06.2021. Feb. 1992. URL: `https://groups.google.com/g/comp.std.c/c/ycpVKxTZkgw/m/S2hHdTbv4d8J`.

Literatur III

[20] Marek Polacek. “GCC Undefined Behavior Sanitizer - ubsan”. In: (Okt. 2014). eingesehen am 13.06.2021. URL: https://developers.redhat.com/blog/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan.

[21] Erik Poll. *Language-based Security: 'Safe' programming languages*. eingesehen am 11.06.2021. URL: http://www-verimag.imag.fr/~mounier/Enseignement/Software_Security/6_LanguageBasedSecurity%20-%20safety.pdf.

[22] John Regehr. “A Guide to Undefined Behavior in C and C++, Part 1”. In: (Juli 2010). eingesehen am 24.05.2021. URL: https://blog.regehr.org/archives/213.

[23] John Regehr. “Compilers and Termination Revisited”. In: (Juli 2010). eingesehen am 06.05.2021. URL: https://blog.regehr.org/archives/161.

[24] John Regehr. “Undefined Behavior in 2017”. In: (Sep. 2017). eingesehen am 17.05.2021. URL: https://github.com/CppCon/CppCon2017/blob/master/Presentations/Undefined%20Behavior%20in%202017/Undefined%20Behavior%20in%202017%20-%20John%20Regehr%20-%20CppCon%202017.pdf.

[25] John Regehr. “Undefined Behavior in 2017”. In: (Apr. 2017). eingesehen am 05.05.2021. URL: https://blog.regehr.org/archives/1520.

[26] Laszlo Szekeres u. a. “Eternal War in Memory”. In: (2013). eingesehen am 17.06.2021. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6547101.

[27] *Undefined Behavior in C and C++*. eingesehen am 18.06.2021. URL: https://www.geeksforgeeks.org/undefined-behavior-c-cpp/.

[28] *Undefined, unspecified and implementation-defined behavior*. eingesehen am 28.06.2021. URL: https://stackoverflow.com/questions/2397984/undefined-unspecified-and-implementation-defined-behavior.

Literatur IV

[29] *Unspecified behavior*. eingesehen am 28.06.2021. URL: https://en.wikipedia.org/wiki/Unspecified_behavior.

[30] *Use after free error?* eingesehen am 03.06.2021. URL: https://stackoverflow.com/questions/1677850/use-after-free-error.

[31] *Use-After-Free*. eingesehen am 03.06.2021. URL: https://encyclopedia.kaspersky.com/glossary/use-after-free/.

[32] *What exactly is meant by "de-referencing a NULL pointer"?* eingesehen am 18.06.2021. URL: https://stackoverflow.com/questions/4007268/what-exactly-is-meant-by-de-referencing-a-null-pointer.

[33] *Wrap around explanation for signed and unsigned variables in C*. eingesehen am 28.06.2021. Nov. 2013. URL: https://stackoverflow.com/questions/19842215/wrap-around-explanation-for-signed-and-unsigned-variables-in-c.

[34] Zhenpeng Wu, Zixuan Yin und Lucas Zamprogno. *Undefined Behaviour in C*. eingesehen am 12.06.2021. URL: https://www.cs.ubc.ca/~rxg/cpsc509/UndefinedBehaviourInC.pdf.

[35] Bojan Zdrnja. "A new fascinating Linux kernel vulnerability". In: (Juli 2009). eingesehen am 13.06.2021. URL: https://isc.sans.edu/forums/diary/A+new+fascinating+Linux+kernel+vulnerability/6820/.