# Kapitel 1: Sicheres Programmieren in C

# 1: Sicheres Programmieren in C

- Dieses Kapitel beschäftigt sich mit dem sicheren Programmieren mit C
- Die meisten Schwachstellein in Programmen basieren auf einer unzureichende Überprüfung von Benutzereingaben
- Die meisten Programmiere rechnen nicht mit invaliden Nutzereingaben
- Goldene Programmierregel: Misstrauen Sie jeder Eingabe
- Diese Grundregeln schützen Sie weitgehend vor den typischen Schwachstellen
  - Cross Site Scripting
  - SQL-Injection
  - Buffer Overflows
  - Code Injection (z.B. für Betriebssystembefehle)

# 1.1: Buffer Overflow

## Definition 2.1 (Buffer Overflow)

Es werden mehr Daten in einen Buffer (normalerweise char-Array) geschrieben werden als dieser aufnehmen kann.

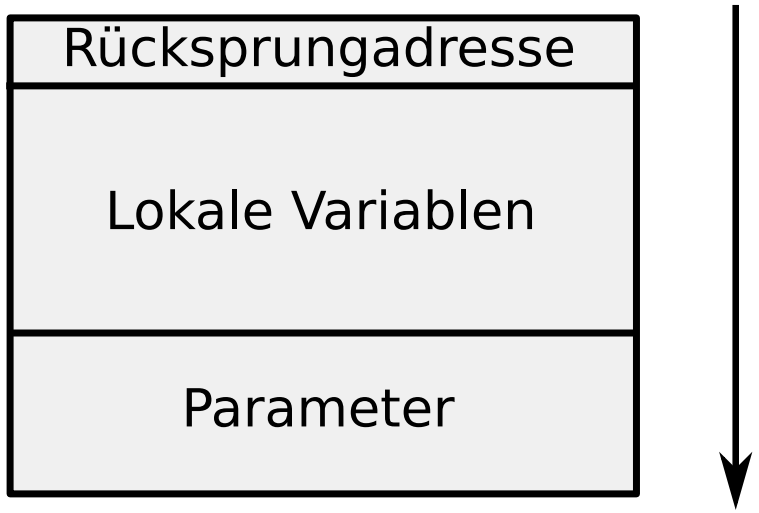
## Beispiel

```
char buf[8];
strcpy(buf, "0123456789");
```

# Auswirkungen eines Buffer Overflows

- Der Prozess wird instabil und stürzt ab
- Der Prozess liefert ein falsches Ergebnis
- Durch das geschickte überschreiben von Speichern kann es dem Angreifer gelingen den Programmfluss des Programmes zu verändern
- Einschleusung und Ausführung von beliebigen Code (Arbitrary code execution)
- Kompromittierung des Rechners (Game Over)

# Der Stack




- In diesem Kapitel widmen wir uns Stack Overflows.
- Neben Stack- gibt es noch Heap Overflows
- Auf dem Stack befinden sich
  - Lokale Variablen
  - Lokale Funktionsparameter
  - Rücksprungadresse der aufrufenden Funktion
- Die Reihenfolge der Einträge hängt vom System ab

# Beispiel: Stack

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int foo(long a, long b) {
5    long x = 3;
6    long y = 4;
7    long z[2] =  { 5, 6 };
8    printf("a:␣␣␣␣␣%lx␣(0x%p)\n", a, &a);
9    printf("b:␣␣␣␣␣%lx␣(0x%p)\n", b, &b);
10   printf("x:␣␣␣␣␣%lx␣(0x%p)\n", x, &x);
11   printf("y:␣␣␣␣␣%lx␣(0x%p)\n", y, &y);
12   printf("z[0]:␣␣%lx␣(0x%p)\n", z[0], &z[0]);
13   printf("z[1]:␣␣%lx␣(0x%p)\n", z[1], &z[1]);
14   printf("z[2]:␣␣%lx␣(0x%p)\n", z[2], &z[2]);
15   printf("z[3]:␣␣%lx␣(0x%p)\n", z[3], &z[3]);
16   printf("z[4]:␣␣%lx␣(0x%p)\n", z[4], &z[4]);
17   printf("z[5]:␣␣%lx␣(0x%p)\n", z[5], &z[5]);
18   printf("z[-1]:␣%lx␣(0x%p)\n", z[-1], &z[-1]);
19   printf("z[-2]:␣%lx␣(0x%p)\n", z[-2], &z[-2]);
20   return a+b;
21 }
22
23 int main() {
24   int x = foo(1,2);
25   printf("%d\n",x);
26   printf("Address␣main:␣%p\n", main);
27 }
```

(Ausgabe des Programms → *Tafel*)

# Stack Overflow Beispiel: Hello World

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

void world() {
  puts("World");
  exit(23);
}


void hello() {
  long z[1] = { 5 };
  if(z[0] == 5) printf("%s","Hello ");
  z[2] = (long) world;
}


int main() {
  hello();
  puts("Goodbye");
}
```

**Frage:** Was ist die Ausgabe des Programms?

# Überschreiben der Rücksprungadresse

- Bei Verlassen einer Funktion wird der Instruction Pointer (IP) bzw. Programm Counter (PC) auf die Rücksprungadresse gesetzt.
- Die CPU führt als nächstes die Instruktion an der Rücksprungadresse aus.
- Mit dem Überschreiben der Rücksprungadresse lässt sich daher beliebiger Code im Adressraum des Programmes anspringen.
- Angreifer können mit Hilfe eines Stack Overflows die Rücksprungadresse einer Funktion überschreiben und so ein Programm unter vollkommener Kontrolle bringen.
- Bei dem Beispiel auf der vorherigen Folie wird die Rücksprungadresse von der Fuktion `hello()` auf die Adresse der Funktion `world()` gesetzt. Dadurch wird diese beim Verlassen von `hello()` aufgerufen.

# Beispiel: Überschreiben der Rücksprungadresse

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

void foobar() {
  puts("Aufwiedersehen");
  exit(23);
}


void hello(long *z) {
  if(z[0] == 5) puts("Hello");
  //z[2] = (long) foobar;
  z[-2] = (long) foobar;
}


int main() {
  long z[1] = { 5 };
  hello(z);
  puts("Goodbye");
}
```

**Frage:** Was ist die Ausgabe des Programms?

# Verwundbarer Code: Passwort Vergleich (1/4)

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

#define SECRET_PASSWORD "strong_password"
#define BUF_SIZE 80
#define EQUAL 0

int main() {
  bool correct_password = false;
  char pwd[BUF_SIZE];
  printf("%s", "Please␣enter␣Password:␣");
  scanf("%s",pwd);

  if(strcmp(pwd, SECRET_PASSWORD) == EQUAL) correct_password=true;

  if(correct_password) puts("Correct␣password");
  else puts("Wrong␣password");
}
```

```
$ python3 -c "print('A'*50)" | ./password_overflow
Please enter Password: Wrong password
```

# Verwundbarer Code: Passwort Vergleich (2/4)

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

#define SECRET_PASSWORD "strong_password"
#define BUF_SIZE 80
#define EQUAL 0

int main() {
  bool correct_password = false;
  char pwd[BUF_SIZE];
  printf("%s", "Please␣enter␣Password:␣");
  scanf("%s",pwd);

  if(strcmp(pwd, SECRET_PASSWORD) == EQUAL) correct_password=true;

  if(correct_password) puts("Correct␣password");
  else puts("Wrong␣password");
}
```

```
$ python3 -c "print('A'*9000)" | ./
    password_overflow
Speicherzugriffsfehler
```

# Verwundbarer Code: Passwort Vergleich (3/4)

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

#define SECRET_PASSWORD "strong_password"
#define BUF_SIZE 80
#define EQUAL 0

int main() {
  bool correct_password = false;
  char pwd[BUF_SIZE];
  printf("%s", "Please␣enter␣Password:␣");
  scanf("%s",pwd);

  if(strcmp(pwd, SECRET_PASSWORD) == EQUAL) correct_password=true;

  if(correct_password) puts("Correct␣password");
  else puts("Wrong␣password");
}
```

```
(gdb) p &correct_password
$1 = (_Bool *) 0x7fffffffdf6b
(gdb) p &pwd
$2 = (char (*)[80]) 0x7fffffffdf10

$ python3 -c "print(0x7fffffffdf6b-0x7fffffffdf10)"
91
```

# Verwundbarer Code: Passwort Vergleich (4/4)

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

#define SECRET_PASSWORD "strong_password"
#define BUF_SIZE 80
#define EQUAL 0

int main() {
  bool correct_password = false;
  char pwd[BUF_SIZE];
  printf("%s", "Please␣enter␣Password:␣");
  scanf("%s",pwd);

  if(strcmp(pwd, SECRET_PASSWORD) == EQUAL) correct_password=true;

  if(correct_password) puts("Correct␣password");
  else puts("Wrong␣password");
}
```

```
$ python3 -c "print('A'*92)" | ./password_overflow
  Please enter Password: Correct password
```

# Lektion

## Die Funktion `scanf` ist böse

Verwenden Sie kein `scanf()` mit der Option `%s`

### Lösungsvorschläge

- `scanf("%79s",pwd);`
- `fgets(pwd, BUF_SIZE-1,stdin);`
- `char *pwd = readline("Please enter Password: ")`

# Password Vergleich 2.0

```
1  #include <readline/readline.h>
2  #include <string.h>
3  #include <stdio.h>
4  #include <stdbool.h>
5  #include <stdlib.h>
6
7  #define SECRET_PASSWORD "strong password"
8
9  int main() {
10   bool correct_password = false;
11   char *pwd = readline("Please enter Password: ");
12
13   if(!strcmp(pwd, SECRET_PASSWORD)) correct_password=true;
14
15   if(correct_password) puts("Correct password");
16   else puts("Wrong password");
17
18   free(pwd);
19 }
```

```
$ python3 -c "print('A'*9000)" | ./password_check
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
....
Wrong password
```

# Verwundbarer Code: Nicht druckbare Zeichen (1/3)

```
1  #include <stdio.h>
2
3  int main () {
4    int foo = 0;
5    char buf[80];
6
7    scanf("%s",buf);
8
9    if(foo == 0x00010203) puts("you win!");
10 }
```

- Der Angreifer gewinnt, wenn er die lokale Variable `foo` den Wert `0x00010203` zuweist.
- Beobachtung `0x00` `0x01`, `0x02` und `0x03` sind nicht druckbare Zeichen.
- **Frage:** Wie gibt man nicht druckbarer Zeichen ein?
- Lösung: Escapezeichen ($\rightarrow$ `man echo`)

# Verwundbarer Code: Nicht druckbare Zeichen (2/3)

```
1  #include <stdio.h>
2
3  int main () {
4    int foo = 0;
5    char buf[80];
6
7    scanf("%s",buf);
8
9    if(foo == 0x00010203) puts("you win!");
10 }
```

```
...
(gdb) p &foo
$1 = (int *) 0x7fffffffdf78
(gdb) p &buf
$2 = (char (*)[80]) 0x7fffffffdf20

python3 -c "print(0x7fffffffdf78-0x7fffffffdf20)"
88
```

# Verwundbarer Code: Nicht druckbare Zeichen (3/3)

```
1  #include <stdio.h>
2
3  int main () {
4    int foo = 0;
5    char buf[80];
6
7    scanf("%s",buf);
8
9    if(foo == 0x00010203) puts("you win!");
10 }
```

## Anmerkung

Die x86 Architektur ist Little-Endian, d.h. bei Ganzzahlen steht das niederwertigste Byte an erster Stelle.

```
python3 -c "print('A' * 88 + '\x03\x02\x01\x00')" | ./
    buf_overflow
you win!
```

# Spaß mit scanf

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5    char buf[100];
6
7    int len =  scanf("%s",buf);
8    printf("len: %d\n, ",len);
9    printf("strlen: %lu\n", strlen(buf));
10   puts(buf);
11
12   for(int i=0; i<10;i++) printf("%c", buf[i]);
13   puts("");
14 }
```

**Frage:** Was ist die Ausgabe bei dem folgenden Aufruf?

```
echo -e "Hallo\x00Welt" | ./fun_scanf
```

# Auswirkungen von Buffer Overflows

- Absturz eines Prozesses
- Überschreiben von Variablen
  $\Longrightarrow$ Änderung des Kontrollflusses
- Überschreiben der Rücksprungadresse
  $\Longrightarrow$ Aufrufen einer beliebigen Funktion
- Komplette Kontrolle über den Stack
  $\Longrightarrow$ Ausführung *beliebigen* Codes
- `https://www.youtube.com/watch?v=pVBnjSQ4Fjk`

# 1.2: Maßnahmen gegen Buffer Overflows

Im folgenden werden drei Gegenmaßnahmen betrachtet, welche Buffer Overflows erschwere oder deren Auswirkungen minimieren.

1. Umsortierung der Stack Variablen
2. Kanarienvögel (Canaries)
3. Address Space Layout Randomization (ASLR)

# Umsortierung der Stack Variablen

- Durch Umsortierung der Stackvariablen können die Auswirkungen von Buffer Overflows abgeschwächt werden.
- Erst werden alle Buffer auf den Stack gelegt.
- Danach erst die Ganzzahlen.
- Dadurch kann ein Bufferoverflow nicht andere Variablen überschreiben, um den Kontrollfluss des Programms zu manipulieren.

# Beispiel: Unsortierte Stack Variablen

```
1  #include <stdio.h>
2
3  int main() {
4    int win = 0;
5    char buf[5];
6    scanf("%s",buf);
7
8    if(win) puts("You Win");
9    else    puts("You Lose");
10 }
```

Bei diesem Programm ist es einfach möglich den Wert der Variable `win` zu manipulieren.

```
python3 -c "print('A'*6)"
You Win
```

# Beispiel: Sortierte Stack Variablen

```
1  #include <stdio.h>
2
3  int main() {
4    char buf[5];
5    int win = 0;
6    scanf("%s",buf);
7
8    if(win) puts("You Win");
9    else    puts("You Lose");
10 }
```

Durch Umsortieren der Stack Variablen ist es nicht mehr einfach möglich den Wert der Variable `win` zu manipulieren.

```
$ python3 -c "print('A'*6)" | ./sort
You Lose
python3 -c "print('A'*20)" | ./sort
Segmentation fault
```

# Canaries (Kanarienvögel)

Quelle: https://thisnessofathat.blogspot.com/2012/05/canary-in-coal-mine.html

- Früher wurden Kanarienvögel im Kohlebergbau eingesetzt, um vor toxischen Grubengase und Sauerstoffmangel (Kohlenmonoxid) zu warnen.
- In der IT-Sicherheit warnen Canaries vor Buffer Overflows.
- Sobald ein Buffer Overflow durch ein Canary entdeckt wurde wird der Prozess frühzeitig beendet (z.B. durch SIGABRT).

# Funktionsweise

- Bei Canaries handelt es sich um Wörter mit bekanntem Inhalt.
- Die Canaries werden im Stack verteilt.
- Nach dem Füllen eines Buffers wird überprüft ob sich der Wert eines Canaries geändert hat.
- Ein Canary Wert hat sich geändert $\Longrightarrow$ Bufferoverflow.
- **Frage:** Warum sollten es sich bei Canaries um Zufallswerte handeln?
- **Frage:** Wo werden Canaries plaziert?

# GCC: Stack Protector

Das Compiler Flag `-fstack-protector-all`

- Sortiert Stack-Variablen um
- Plaziert immer zwischen lokalen Variablen und Parametern einen Canary.
- Sendet das Signal SIGABRT an sich selbst, falls sich der Wert des Canary verändert hat.

**Verwenden Sie, falls möglich, das Compiler Flag `-fstack-protector-all`.**

# Stack Protector ist kein Allheilmittel

```
1  #include<stdio.h>
2  #include <string.h>
3
4  int main() {
5    char buf1[9];
6    char buf2[9];
7
8    scanf("%s",buf1);
9
10   if( !strcmp(buf2,"password") ) puts("You win!");
11   else puts("You Lose");
12 }
```

```
$ cc -fstack-protector-all -o antispa antispa.c
$ python3 -c "print('A'*9 + 'password')" | ./
    antispa
You win!
```

# Address Space Layout Randomization (ASLR)

- Klassisch sind die Funktionen eines Prozesess immer an der gleichen Addresse.
- Dies ist einfach zu implementieren und erleictert das Debugging.
- Dies erleichtert es aber auch einen Angreifer bei einem Buffer Overflow die Rücksprungadresse des Stack mit einer gewünschten Funktion wie `system()` oder `execve()` zu überschreiben, um beliebigen Schadcode auszuführen.
- Bei ALSR wird der Addressraum ausgewürfelt, d.h, alle Funktionen des Programms befinden sich an einer zufälligen Addresse.
- Dies erschwert das Schreiben eines Buffer Overflows.

# ASLR und GCC

- Bibliotheken (shared library, shared object)
  - PIC (Position-Independent Code)
  - **Verwenden Sie, falls möglich, die Compiler Flags `-fPIC`.**

- Ausführbare Programme (executable)
  - PIE (Position-Independent Executable)
  - **Verwenden Sie, falls möglich, die Compiler Flags `-fPIE`.**

# Best Practics: Schutz vor Buffer Overflows

- Testen Sie alle Funktionen die Buffer verarbeiten ausgiebig
- Simulieren Sie Nutzereingaben immer mit sehr langen Strings
- Verwenden Sie bei `scanf()` niemals den Formatstring `"%s"`
- Lesen Sie Benutzereingaben mit `fgets()` ein.
- Verwenden Sie die folgenden Compilerflags
  - `-fstack-protector-all`
  - `-fPIE` bzw. `-fPIC`

# AddressSanitizer (ASan)

- Werkzeug von Google um fehlerhafte Speicherzugriffe zu erkennen.
- Erkennt unter anderem auch Buffer Overflows.
- Ausführungszeit des Programms erhöht sich um ca. 70%.
- Speicherbedarf des Programms ist ca. 3,5 mal so hoch.
- Sollte bei der Entwicklung immer aktiviert sein.
- Integration in clang und gcc: `-fsanitize=address`
- Hat in in dem Chromium Webbrowser über 300 Fehler gefunden.

# AddressSanitizer: Demo

```c
#include <stdlib.h>

int main() {
  int *a = malloc(sizeof(int)*100);
  a[100] = 0;
  int res = a[100];
  return res;
}
```

```
make
cc -ggdb3 -W -Wall -Wextra -Werror -fsanitize=address
    asdemo.c -o asdemo
./asdemo
...
SUMMARY: AddressSanitizer: heap-buffer-overflow
/home/cforler/git/beuth/it-sec/vorlesung/examples/insecure
    /asdemo.c:6
in main
...
```

# Makefile mit Entwicklung- und Release-Version

```
CFLAGS += -W -Wall -Wextra -Werror
DFLAGS += -ggdb3 -fsanitize=address        # Dev flags
RFLAGS += -fPIE -fstack-protector-all -O2 # Release flags

ifdef RELEASE
CFLAGS += $(RFLAGS)
else
CFLAGS += $(DFLAGS)
endif

all: program

program:

clean:
        $(RM) program
```

```
make
make RELEASE=1
```

# 1.3: Systemcall Filtern

- Der Linux Kernel hat einen eingebauten Filter für Systemaufrufe.
- Mit der Bibliothek `libseccomp` kann *einfach* darauf zugegriffen werden.
- Für sichere Programme können hier alle nicht benötigten Systemaufrufe gefiltert werden.
- Gefilterte Systemaufrufe werden nicht aufgerufen,
- stattdessen wird dem aufrufende Prozess das Signal `SIGSYS` gesendet.
- Diese Technik erschwert das Ausnutzen von Schwachstellen erheblich.

# Seccomp: Initialisierung

```
#include <seccomp.h>
typedef void * scmp_filter_ctx;

scmp_filter_ctx seccomp_init(uint32_t def_action);
```

- Initialisierung der *Seccomp-Struktur*.
- Im Fehlerfall wird `NULL` zurückgegeben.
- Das Standardverhalten kann mit der Argument `def_action` festgelegt werden:
  - **SCMP_ACT_KILL**: Thread der ein gefilterten Systemaufruf sendet wird vom Kernel mit einem SIGSYS-Signal terminiert.
  - **SCMP_ACT_KILL_PROCESS**: Thread der ein gefilterten Systemaufruf sendet wird vom Kernel mit einem SIGSYS-Signal terminiert.
  - **SCMP_ACT_LOG**: Gefilterte Systemaufrufe werden vom Kernel via `syslog` gelogged.

# Seccomp: Regeln hinzufügen

```
#include <seccomp.h>
int SCMP_SYS(syscall_name);

nt seccomp_rule_add(scmp_filter_ctx ctx, uint32_t action,
                    int syscall, unsigned int arg_cnt, ...);
```

- Es wird eine Regel der *Seccomp-Struktur* `ctx` hinzugefügt,
- Seccomp Regeln können in der BPF (Berkeley Packet Filter) ähnlichen Pseudo-Maschinensprache geschrieben werden.
- Regeln werden via optionale Parameter angegeben. Die Anzahl wird mit dem Paramter `arg_cnt` angegeben
- Hier werden nur elementare Regeln mit `arg_cnt==0` betrachtet.
- Parameter `syscall` definiert den betreffenden Systemaufruf.
- Neben den bekannten Aktionen gibt es noch `SCMP_ACT_ALLOW`. Dieser erlaubt den Aufruf des angegebenen Systemaufrufs.

# Seccomp: Regeln laden und freigeben

```
#include <seccomp.h>

int seccomp_load(scmp_filter_ctx ctx);
int seccomp_release(scmp_filter_ctx ctx);
```

- `seccomp_load()`: Die Regeln der *Seccomp-Struktur* `ctx` werden in den Kernel geladen und aktiviert.
- `seccomp_release()`: Freigabe des durch die *Seccomp-Struktur* `ctx` belegten Heapspeichers.

# Seccomp: Beispiel 1

```
1  #include <stdio.h>
2  #include <seccomp.h>
3
4  void allow_syscall(scmp_filter_ctx ctx, int syscall) {
5    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, syscall, 0);
6  }
7
8  void set_allow_list(scmp_filter_ctx ctx) {
9    allow_syscall(ctx, SCMP_SYS(exit_group));
10   allow_syscall(ctx, SCMP_SYS(newfstatat));
11   allow_syscall(ctx, SCMP_SYS(write));
12 }
13
14
15 int main() {
16   scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);
17   set_allow_list(ctx);
18   seccomp_load(ctx);
19
20   puts("Hello␣World");
21
22   seccomp_release(ctx); // Free ctx from heap
23 }
```

# Seccomp: Beispiel 2

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

void allow_syscall(scmp_filter_ctx ctx, int syscall) {
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, syscall, 0);
}

void set_allow_list(scmp_filter_ctx ctx) {
  allow_syscall(ctx, SCMP_SYS(exit_group));
  allow_syscall(ctx, SCMP_SYS(newfstatat));
  allow_syscall(ctx, SCMP_SYS(write));
}


int main() {
  scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);
  set_allow_list(ctx);
  seccomp_load(ctx);

  puts("Hello␣World");

  seccomp_rule_add(ctx,SCMP_ACT_KILL,SCMP_SYS(write),0);
  seccomp_load(ctx);

  puts("Foobar");

  seccomp_release(ctx); // Free ctx from heap
}
```

# Zusammenfassung

Sie sollten . . .

- . . . verstanden haben was ein Buffer Overflow ist.
- . . . Maßnahmen gegen Buffer Overflows kennen.
- . . . wissen was der Compilerflag -fstack-protector-all bewirkt
- . . . wissen was der Compilerflag -fPIE bewirkt.
- . . . wissen was der Compilerflag -fsanitize=address bewirkt.