

C-Kurs

Debugging

Stack

Debugging

Techniken

Beispiel Debugger: gdb

Debugging

- ❑ Nötig während der Entwicklung
 - Hinweis: Code sollte Stück für Stück entwickelt, getestet und debugged werden!
- ❑ Nötig, wenn Code nicht compiliert
- ❑ Nötig, wenn Software
 - „sich anders verhält als erwartet“
- ❑ D.h., Resultate unterscheiden sich von Spezifikation
- ❑ Typischerweise
 - Nicht, weil die ganze Software voller Fehler ist
 - Sondern, wegen eines kleinen inkorrekten Codefragments (Bug)
- ❑ **Bug:** Codefragment das nicht seiner Spezifikation entspricht

□ Konsequenzen von Bugs:

- Compiler gibt syntaktischen/semantischen Fehler (syntax/semantic error)
- Programm hält mit Laufzeit Fehler (run-time error)
- Programm hält nie an
- Programm läuft vollständig, aber gibt inkorrekte Resultate
- Programm läuft vollständig, aber gibt manchmal inkorrekte Resultate

Syntaktische Fehler

- ❑ Vorgehen, wenn der Compiler syntaktische Fehler ausgibt
- ❑ Zum **ersten** Fehler gehen (wegen möglicher Folgefehler)
 - In die passende Zeile im Code gehen (Ist in der Fehlermeldung des Compilers angegeben)
 - Fehler verstehen
 - Fehler beheben
 - Compilieren
 - Fehler behoben ?
 - Wenn ja gegebenenfalls nächsten Fehler beheben
 - Wenn nein versuchen, Fehler zu verstehen und zu beheben...

□ Beispiel eines buggy Programms

```
int main ( int argc, char *argv[]) {  
    int n = 1024;  
    int buf[n];  
    unsigned int i;  
    for (i=n-1; i >= 0; i--) {  
        buf[i] = n;  
    }  
}
```

Debugging (2.)

□ Drei Aspekte

- Code finden, der das Problem verursacht
- Verstehen, wieso Code ein Problem verursacht
- Code fixen, sodass das Problem gelöst wird

□ Generell:

- Nach Lokalisierung des Problems ist es relativ einfach das Problem zu verstehen
- Fixen des Bugs ist verhältnismäßig einfach nachdem das Problem gefunden und verstanden ist
(Außer für den Fall das ein substantielles Re-design und/oder Re-Implementation notwendig ist)

Debugging (3.)

❑ Fixen von Bugs (Normalfall):

- Prüfen der Spezifikation
- Modifikation des Codes entsprechend der Spezifikation

❑ Spezialfall:

- Modifikation der Spezifikation
- Modifikation des Codes

❑ Spezialfall 2:

- Modifikation der Spezifikation: Bug = Feature (Merkmal)

Debugging: Verstehen des Fehlers

- ❑ Vorgehen, nachdem eine kleine fehlerhafte Region im Code gefunden wurde
 - Was ist der Zustand des Programms vor Ausführung des Codes
 - Was ist der Zustand des Programms nach der Ausführung
 - Vergleich dieses Zustandes mit dem erwarteten Zustand (nach Spezifikation)
 - Verfolgen des Codes, um zu finden, wo der Zustand nicht wie erwartet verändert wurde
- ❑ Was ist der Zustand eines Programms?
 - Namen und Werte aller aktiven Variablen
 - Z.B. $x == 3, y == 7, \dots$
 - Der Zustand eines Programms kann sehr, sehr groß sein

Debugging: Fehlerlokalisierung

Schwierigste Schritt: Lokalisierung des Bugs

- ❑ Bug == Codesegment mit nicht beabsichtigten Aktionen.
Man muss verstehen:
 - Im Detail, was der Code machen **solte**
 - Im Detail, was der Code wirklich **macht**
- ❑ **Hauptproblem**: Riesige Mengen von Details
(jedenfalls in nicht trivialen Programmen)
- ❑ **Haupttrick** zum effektiven Debuggen:
Einengen des „focus of attention“
- ❑ Ausnutzen von Suchstrategien, die es erlauben auf den Bug zu zoomen („zoom in“)

Debugging: Suchstrategie

Gegeben: Buggy Programm

- ☐ Am Anfang ist alles OK
- ☐ Irgendwann wird ein buggy Statement ausgeführt, danach ist der Programmzustand inkorrekt

Ziel:

- ☐ Identifikation des Codeblocks, in dem der Zustand „korrupt“ wird.

Verlangt:

- ☐ Kenntnis des Zustandes an verschiedenen Punkten während der Ausführung

Debugging: Suchstrategie (2.)

Ausnutzung typischen Strukturen eines Programms

□ Anschauen (Display) aller Hauptdatenstrukturen nach

- Initialisierung
- An „strategischen Punkten“ während der Programmausführung
- Am Ende des Programms

□ Wie findet man strategische Punkte?

- Nach der ersten, zweiten, mittleren Iteration einer wichtigen Schleife
- Nach einem Tastendruck, der ein interaktives Programm zum Absturz bringt
- Nach dem Punkt, wo das Programm die letzte korrekte Ausgabe liefert

Programmzustandsuntersuchung

Endscheidendes Tool: Mechanismus zum Anzeigen des Zustands

☐ Eine Möglichkeit: **printf** von „suspekten“ Variablen

☐ Probleme:

- Ändert das Programm
- Falsches Raten bezüglich der suspekten Variablen
- Ausgabe von viel zu vielen Daten

Programmzustandsuntersuchung

❑ Zweite Möglichkeit: **Debugger**

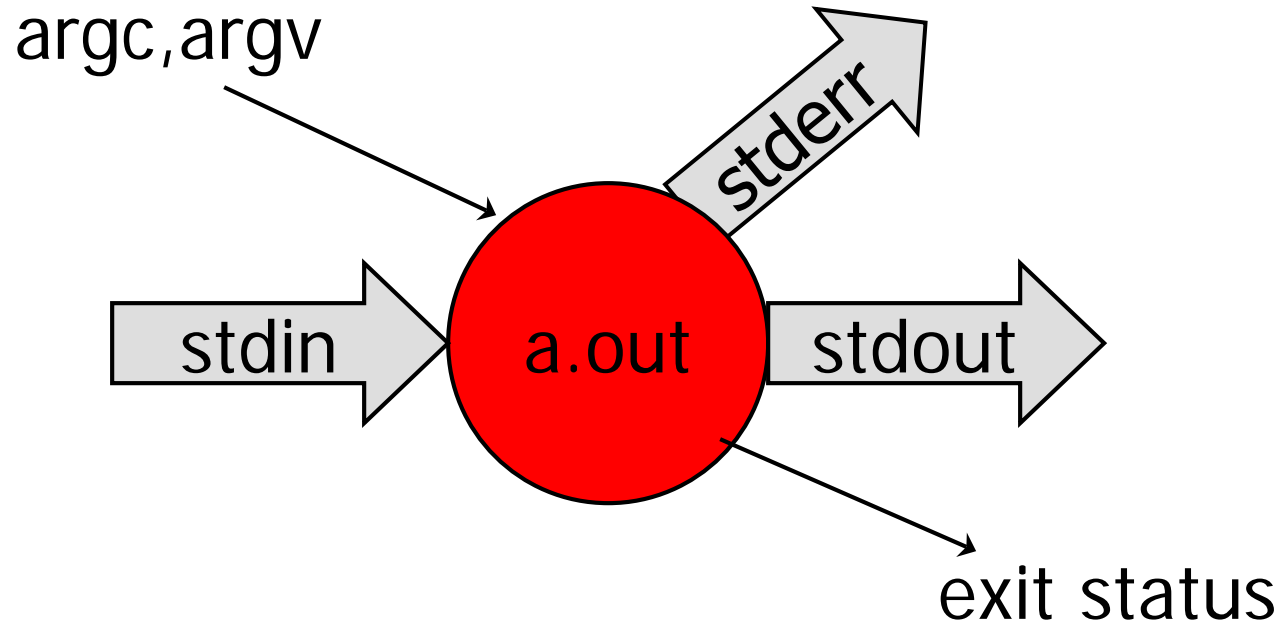
- Kann ein Programm an bestimmten Punkt unterbrechen
- Kann bei Unterbrechung den Zustand anzeigen

❑ Zusätzlich ermöglichen Debugger

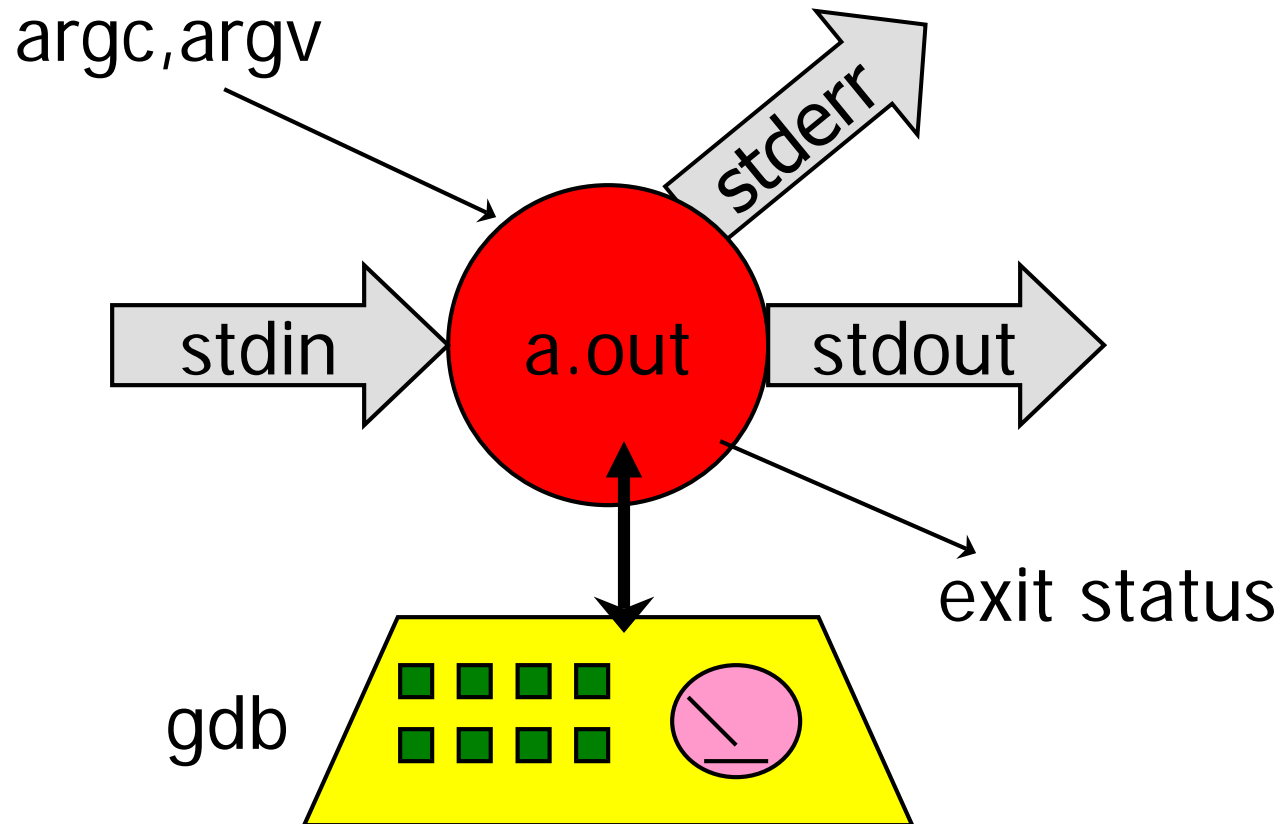
- Den Zustand von Programmen, die durch „run-time“ Fehler (Laufzeitfehler) abstürzen anzuzeigen
- Führt oft zu dem Punkt, wo der Fehler auftritt

- ❑ Ein C, C++ Programm läuft bis:
 - Zum Ende, und produziert ein Resultat
 - Ist das Resultat korrekt?
 - Ist das Resultat nicht das Erwartete?
 - Bis das Programm einen Fehler findet und `exit()` aufruft
 - Bis ein Laufzeitfehler zum Stoppen führt
(z.B. Segmentation fault – core dumped)
(z.B. Laufzeitfehler – Speicherabzug geschrieben)

Normale Programmausführung



Programmausführung mit Debugger



- ❑ Über Debugger kann die Programmausführung kontrolliert werden
 - Normale Ausführung (**run**, **cont**)
 - An einem gewissen Punkt zu halten (**break**)
 - Ein Statement pro Schritt (**step**, **next**)
 - Über Subroutine Aufrufe hinweg (**next**)
 - In Subroutinen anhalten (**step**)
 - Den Programmzustand zu inspizieren (**print**, **eXamine**)
 - Variablen (**print**)
 - Speicher (**eXamine**)

Debugger (2.)

- ❑ Das Tool gdb ist ein Kommandozeilen basierter Debugger für C, C++, ...
- ❑ Es gibt GUI front-ends (z.B. ddd, kdbg, xxd, ...)
- ❑ Funktionalität:
 - Kontrollierte Programmausführung
 - Anzeige von Zustand
 - Zusätzlich: Änderung von Variablen, Speicher, ...

Debugger (3.)

- ❑ Um gdb nutzen zu können: Compilieren mit `-g` Argument
- ❑ Gdb nimmt zwei Argumente:
 - `gdb executable core`
- ❑ Z.B.:
 - `gdb a.out core`
 - `gdb myprog`
- ❑ Das Argument `core` ist optional

- ❑ gdb ist wie eine Shell zum Kontrollieren und Beobachten eines Programms

gdb Kommandos: Basis

- ❑ **quit** – verlässt gdb
- ❑ **help [CMD]** – on-line Hilfe für Kommando CMD
- ❑ **run ARGS** – Ausführen des Programms mit Argumenten
ARGS, z.B.:
 - **a.out datei.in**
- ❑ **wird**
 - **run datei.in**

gdb Kommandos: Status

- ❑ **where** – gibt Aufrufkette (stack trace) aus
 - Mit core dump: Finden welche Funktion das Programm ausgeführt hat als es abgestürzt ist
 - Bei Unterbrechung: Ausgabe der Aufrufkette

Ausflug: Aufrufketten

Aufrufkette: Beispiel

Code Struktur

```
who (...)  
{  
    •  
    •  
    amI ( ) ;  
    •  
    •  
}
```

- Funktion **amI**
rekursiv

```
amI (...)  
{  
    •  
    •  
    amI ( ) ;  
    •  
    •  
}
```

Aufrufkette

```
who  
↓  
amI  
↓  
amI
```

Stack-Frames

□ Inhalt

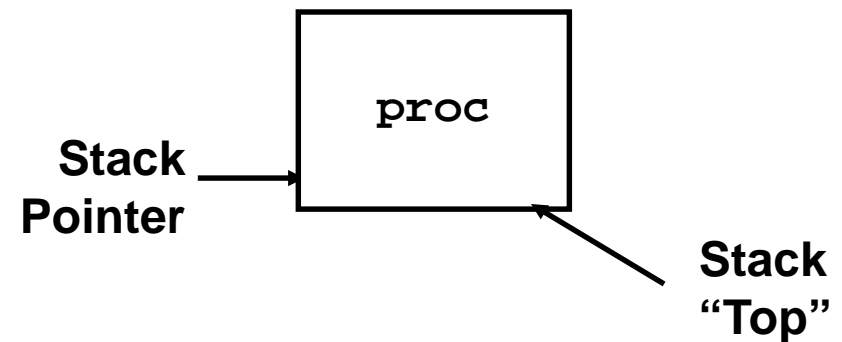
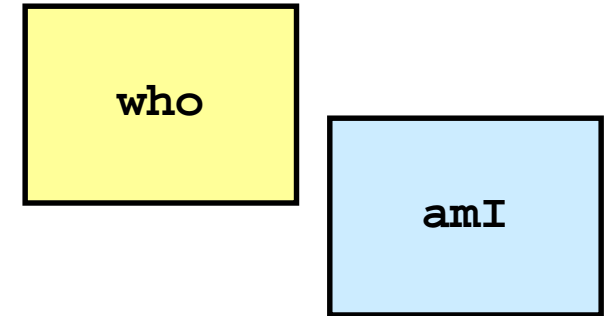
- Lokale Variablen
- Rückkehrinformation
- Parameter

□ Speicherverwaltung

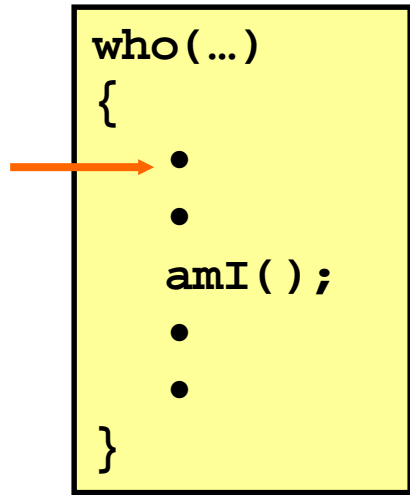
- Speicher alloziert beim Eintritt in die Funktion
- Freigegeben bei der Rückkehr

□ Hilfsmittel

- Stack-Pointer

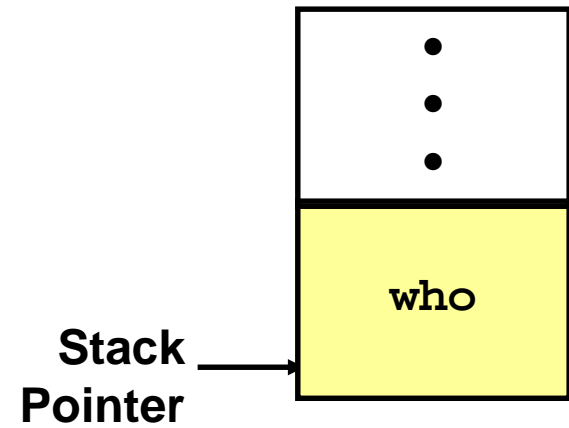


Stack-Operationen

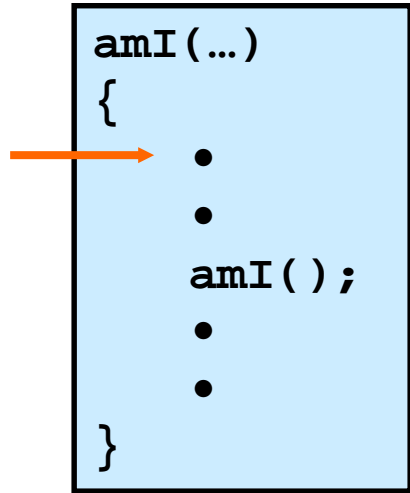


Aufrufkette

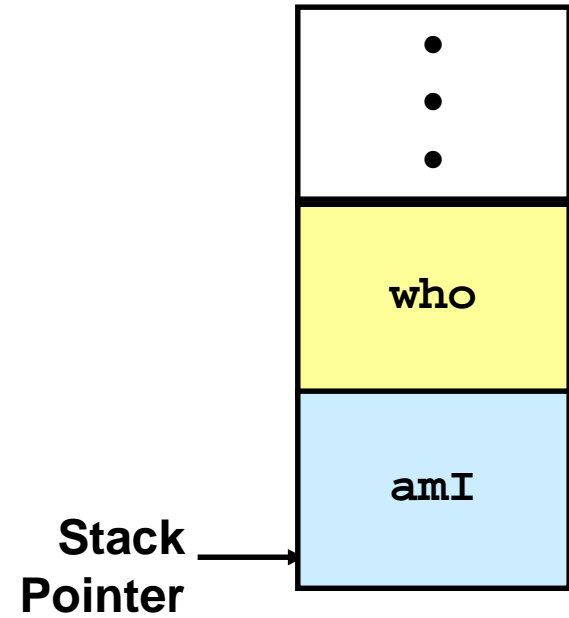
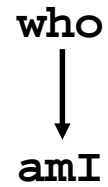
who



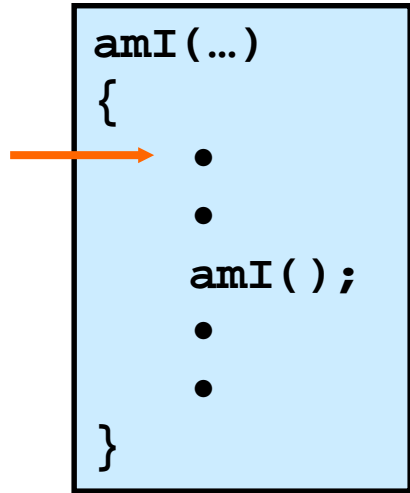
Stack-Operationen



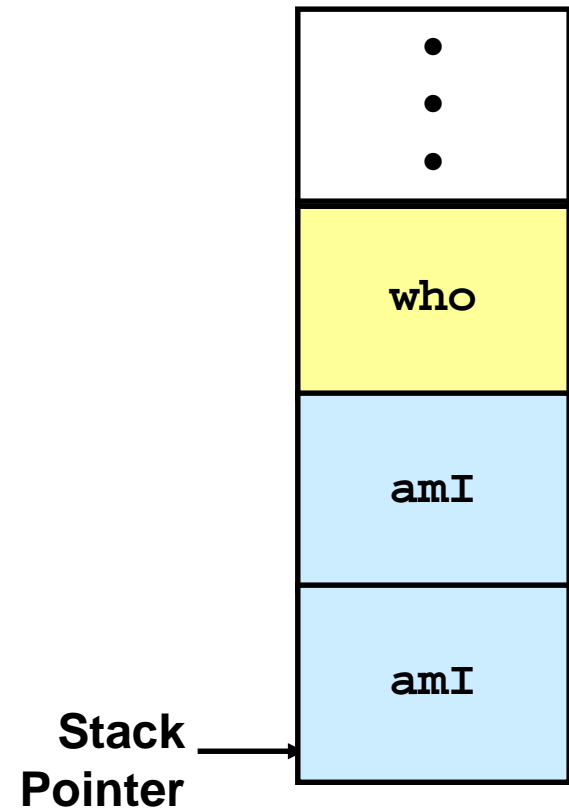
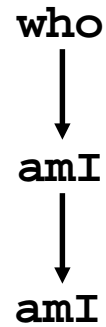
Aufrufkette



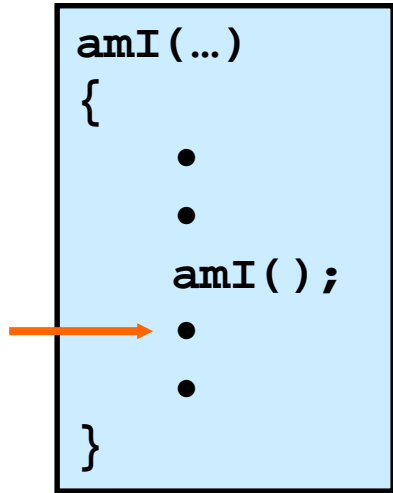
Stack-Operationen



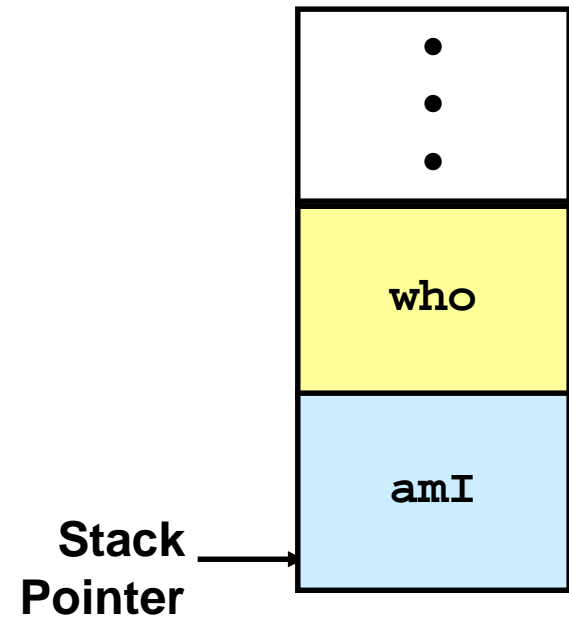
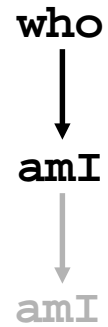
Aufrufkette



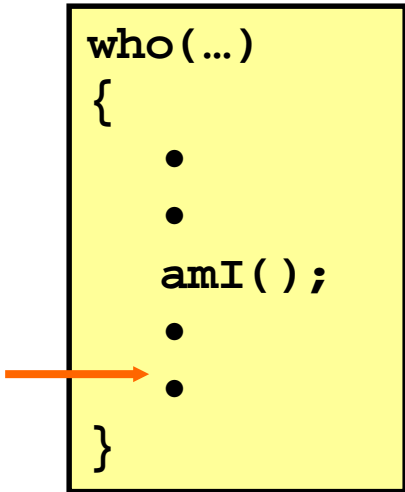
Stack-Operationen



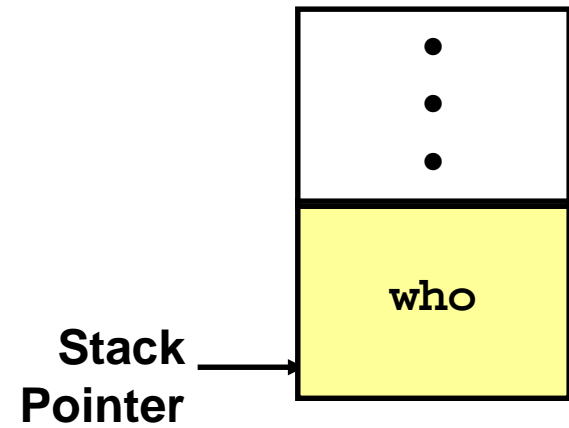
Aufrufkette



Stack-Operationen



Aufrufkette



Zurück zum Debuggen

gdb Kommandos: Status

- ❑ **where** – gibt Aufrufkette (stack trace) aus
 - Mit core dump: Finden welche Funktion das Programm ausgeführt hat als es abgestürzt ist
 - Bei Unterbrechung: Ausgabe der Aufrufkette

gdb Kommandos: Status (2.)

- ❑ **up [N] (im stack)** – wechseln des Kontext im Stack eine Ebene höher; Ändert den Rahmen (Scope) einer bestimmten Funktion im Stack
- ❑ **down [N]** – wechseln des Kontext eine Ebene niedriger
- ❑ **list [LINE/PROC]** – anzeigen des Programmcodes; zeigt 5 Zeilen Code vor und nach dem momentanen Statement
- ❑ **print EXPR** – zeige die Werte der Expression EXPR

gdb Kommandos: Ausführung

- ❑ **break [PROC|LINE]** – Setzen eines Haltepunkts (breakpoint). Wenn das Programm die Funktion PROC (oder die Linie LINE) erhält, wird die Ausführung des Programms unterbrochen und die Kontrolle an **gdb** übergeben
- ❑ **next** – single step (over procedures); Ausführen des nächsten Statements. Falls das Statement ein Funktionsaufruf ist, ausführen des gesamten Funktionskörpers
- ❑ **step** – single step (into procedures); Ausführen des nächsten Statements. Falls das Statement ein Funktionsaufruf ist, halte beim ersten Statement in der Funktion

Benutzen eines Debuggers

- ❑ Am häufigsten nach einem Laufzeitfehler
- ❑ Starten von **gdb** mit **core** Datei, anzeigen mit **where**, welche Programmzeile zum Absturz geführt hat
- ❑ Wo das Programm abstürzt ist, ist häufig ein erster Hinweis auf den Ort des Fehlers
- ❑ Allerdings nur ein erster Hinweis: Der Fehler kann viel früher aufgetreten sein.

Benutzen eines Debuggers (2.)

- ❑ Wenn man eine Idee hat, wo der Fehler sein kann
 - Setzen eines **Breakpoints** kurz vorher im Code
 - **Laufen** lassen des Programms (mit den selben Daten)
 - Ausführen des Programms in **Einzelschritten** (single-step) durch die suspekte Region (Nach dem Breakpoint)
 - **Ansehen** der Werte der suspekten Variablen nach jedem Schritt
- ❑ Hierdurch sollte man feststellen können, welche Variable den falschen Wert hat

Benutzen eines Debuggers (3.)

- ❑ Nachdem man herausgefunden hat, dass der Wert einer Variablen (z.B. *x*) falsch ist, muss man herausfinden **warum** der Wert falsch ist.
- ❑ Es gibt zwei Möglichkeiten:
 - Das Statement, das *x* den Wert zuweist, ist falsch
 - Die Werte der anderen Variablen im Statement sind falsch
- ❑ Beispiel
 - `if (c > 0) { x = a + b; }`
 - Falls wir wissen, dass *x* falsch ist und dass die Bedingung und der Ausdruck richtig implementiert sind, dann müssen wir finden, wo *a* und *b* gesetzt werden

Debugging Beispiel: „**wissenschaftliches Vorgehen**“

- Entwicklung einer Hypothese
- Datensammlung zur Verifikation der Hypothese
- Änderung der Hypothese, falls neue Beweismittel vorliegen

□ Die Hypothese ist:

- „Ich denke, der Bug liegt in diesem Statement ...“

Gesetze des Debuggen

(Zoltan Somogyi, Melbourne University)

- ❑ Before you can fix it, you must be able to break it (consistently)
 - Nicht reproduzierbare Bugs ... Heisenbugs ... sind schwierig
- ❑ If you can't find a bug where you're looking, you're looking in the wrong place
 - Eine Pause machen und später mit dem Debuggen weitermachen, ist im allgemeinen eine gute Idee
- ❑ It takes two people to find a subtle bug, but only one of them needs to know the program
 - Die zweite Person stellt Fragen, um die Annahmen des Debuggers in Frage zu stellen

Mögliche falsche Annahmen

- ☐ Der Binärcode entspricht dem Quellcode, den Sie lesen
- ☐ Code der diese Funktion aufruft bekommt niemals unerwartete Argumente
(dieser Pointer wird niemals NULL sein
warum sollte jemand NULL übergeben)
- ☐ Library Funktionen funktionieren immer ohne Fehler
(malloc wird mir immer den angeforderten Speicher geben)
- ☐ Systembibliotheken und -tools sind fehlerfrei

Sichtbarkeit (Scope) von Variablen

Sichtbarkeitseinschränkungen von Variablen: Warum?

- ❑ Ohne Sichtbarkeitseinschränkungen wären alle Variablen immer sichtbar und zugreifbar
- ❑ Unbeabsichtigtes Überschreiben von Variablenwerten
 - Z.B. Zwei Funktionen benutzen zufällig die selbe Variable
- ❑ D.h. Namen sind nicht wiederverwendbar ohne mögliche Seiteneffekte (unbeabsichtigtes Überschreiben von Werten)

Sichtbarkeitseinschränkungen von Variablen: Welche?

- ☐ Global – keine Einschränkung
- ☐ Lokal – Einschränkung auf Block
- ☐ Static – Einschränkung auf „Datei“

□ Globale Variablen

- Beim Start des Programms wird der entsprechende Speicherplatz auf dem Heap allokiert
 - Sind aus allen Modulen des Programms zugreifbar
 - Änderungen der Werte wirken immer global in allen Modulen
-
- Um auf eine globale Variable aus einem anderen Modul zugreifen zu können, braucht es folgende Deklaration:

extern Typname Variablenname;

□ Globale Variablen

- Globaler Zugriff kann mit static auf die Datei in der die Variable definiert ist eingeschränkt werden

static Typname Variablenname;

- Somit ist kein Zugriff außerhalb der Datei möglich

□ Lokale Variablen

- Lokale Variablen eines Blocks werden auf dem Stack alloziert, d.h. Nach dem Verlassen des Blocks wird der Speicherplatz freigegeben
- Wird eine lokale Variable als static definiert, so bleibt sie erhalten und behält ihren Wert (Vorsicht!) – sie werden auf dem Heap alloziert

Dynamische Speicherallokation

- ❑ Speicher wird mittels `malloc/calloc` angefordert
- ❑ Allokation erfolgt auf dem Heap
- ❑ Inhalte bleiben erhalten, bis Werte überschrieben werden
- ❑ Explizites Freigeben mittels `free()`

Überschreiben von Werten ... Bufferoverflows

❑ Implementation der Unix Funktion gets

- Keine Möglichkeit die Anzahl, der zu lesenden Zeichen anzugeben

```
/* Get string from stdin */  
char *gets(char *dest) {  
    int c = getc();  
    char *p = dest;  
    while (c != EOF && c != '\n') {  
        *p++ = c;  
        c = getc();  
    }  
    *p = '\0';  
    return dest;  
}
```

- Ähnliche Probleme auch bei anderen Unix Funktionen
 - strcpy: kopiert einen String beliebiger Länge
 - scanf, fscanf, sscanf, mit %s Konvertierungsspezifikation

Angreifbarer Buffer-Code

```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Way too small! */  
    gets(buf);  
    puts(buf);  
}
```

```
int main()  
{  
    printf("Type a string:");  
    echo();  
    return 0;  
}
```

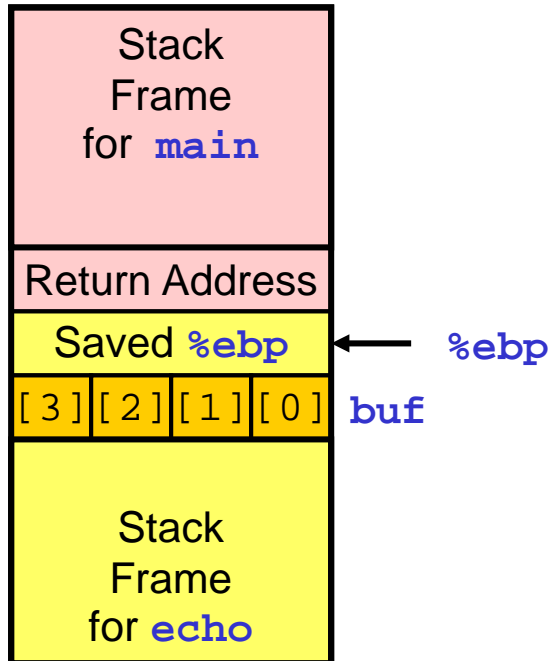
Beispiel eines Bufferoverflows

```
unix>./bufdemo  
Type a string:123  
123
```

```
unix>./bufdemo  
Type a string:12345  
Segmentation Fault
```

```
unix>./bufdemo  
Type a string:12345678  
Segmentation Fault
```

Stack während des Bufferoverflow

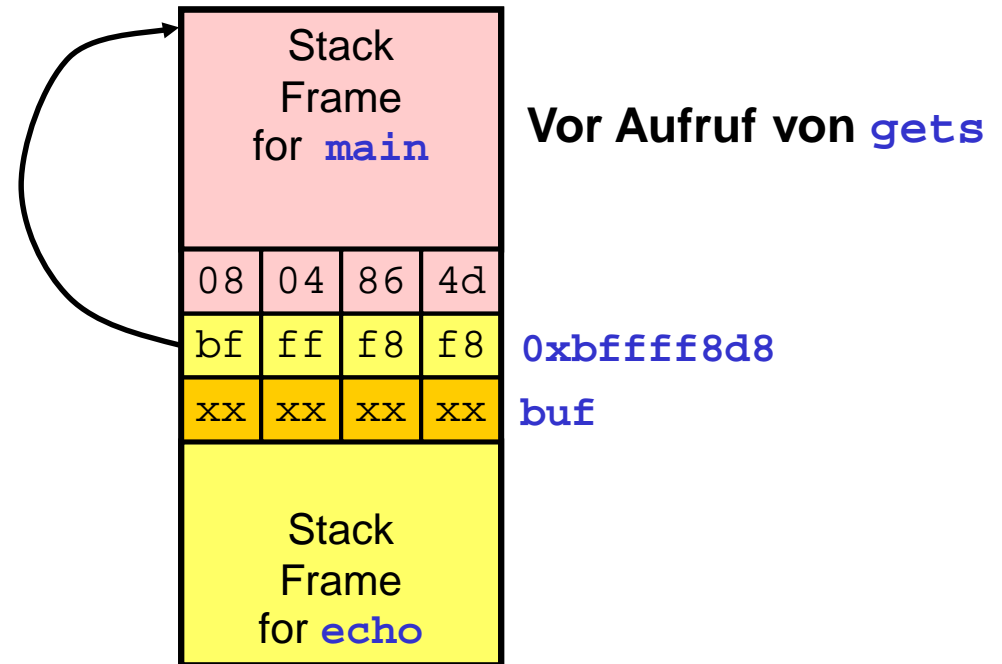
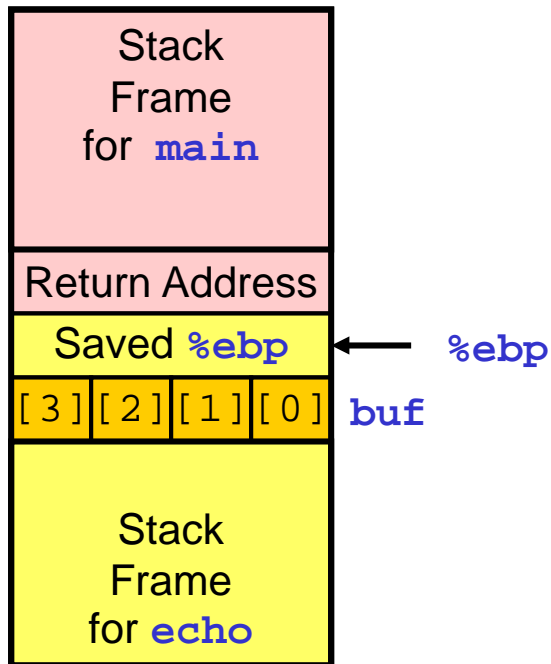


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Stack während des Buffer-overflow

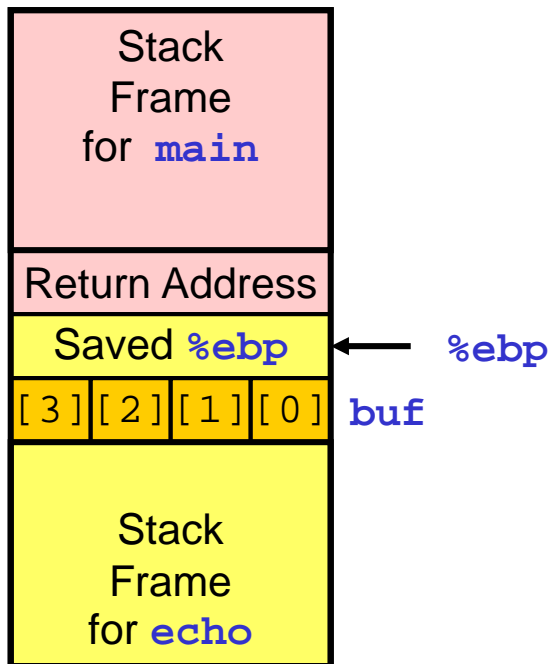
```

unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
    
```

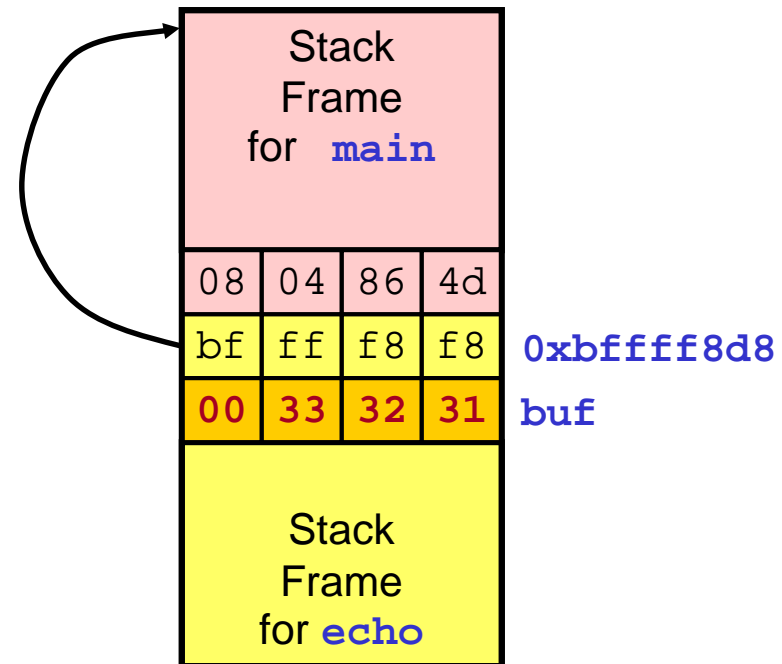


Stack während Bufferoverflow: #1

Vor dem Aufruf von `gets`



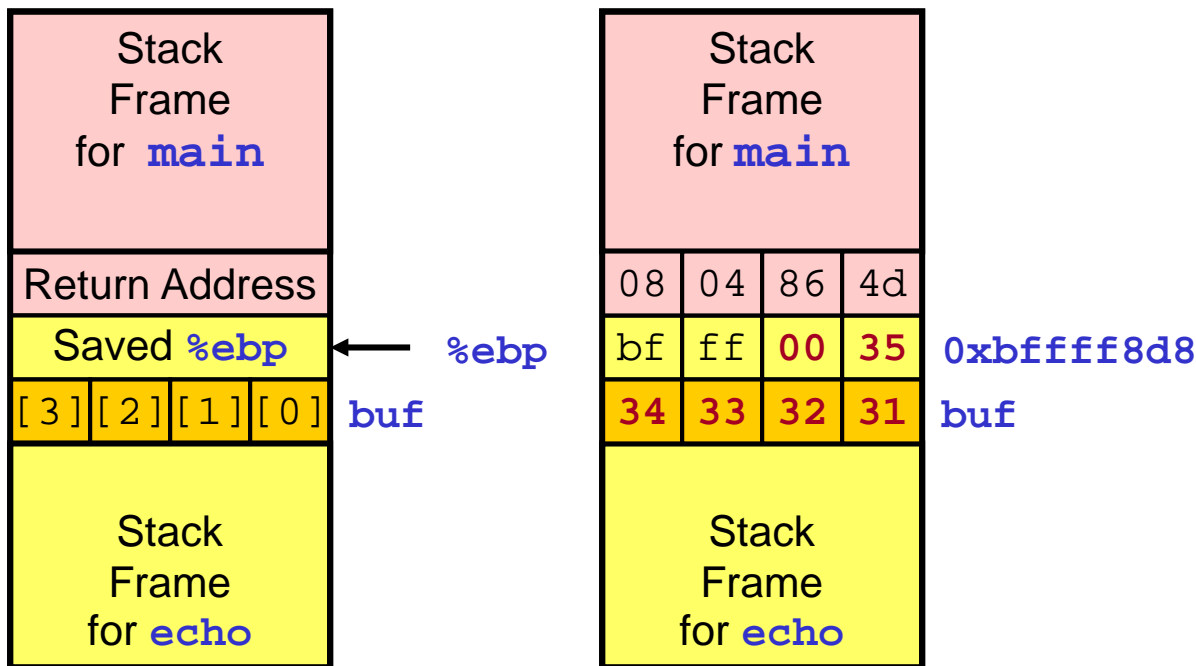
Eingabe: "123"



Kein Problem

Stack während Bufferoverflow #2

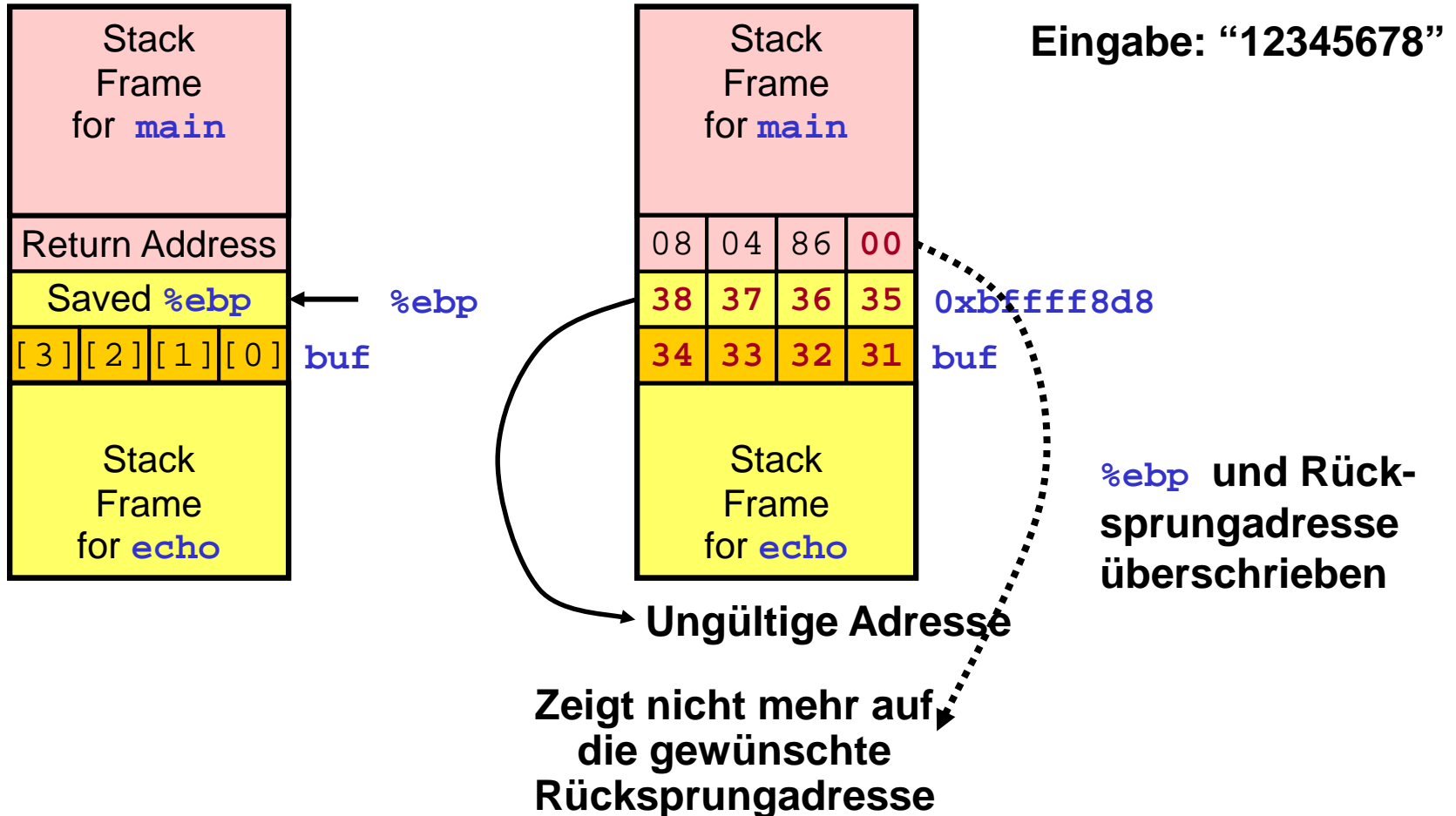
Eingabe: "12345"



Gespeicherter Wert von **%ebp wird auf **0xbffff0035** gesetzt**

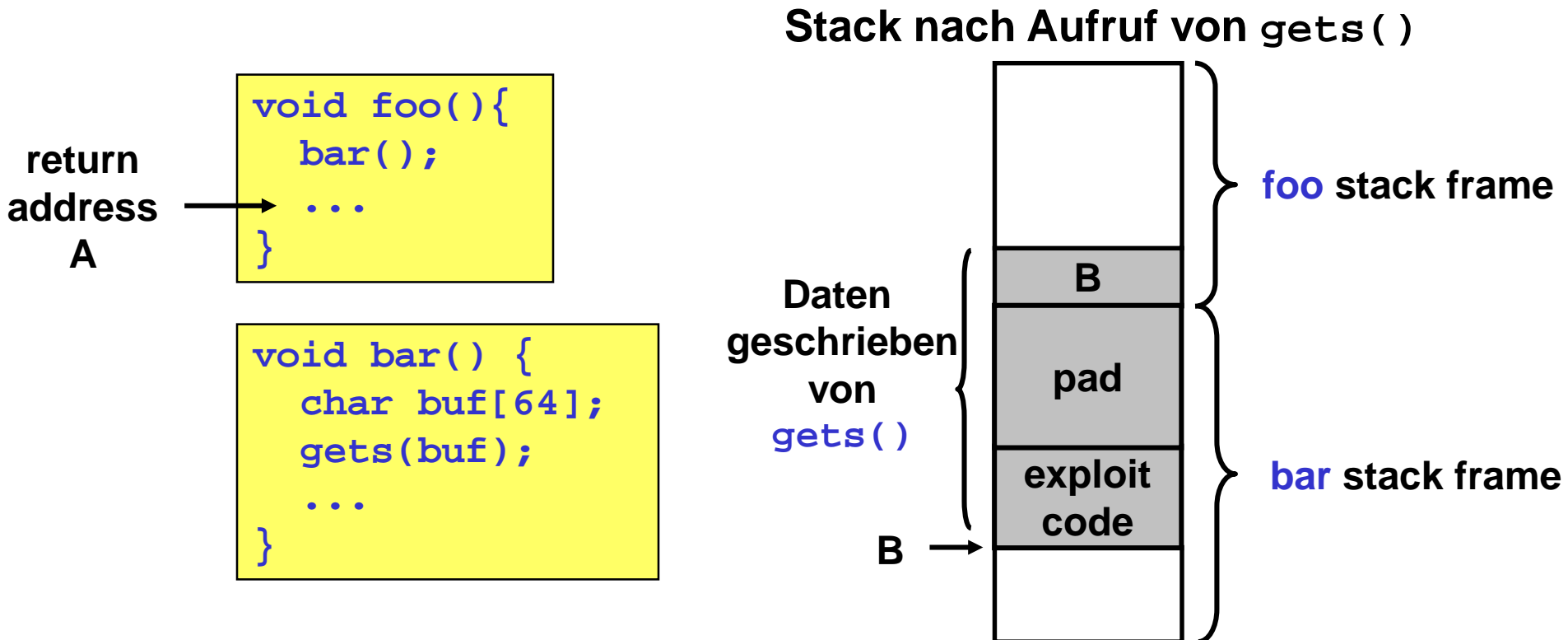
Schlechte Nachricht, wenn später versucht wird den Wert von **%ebp wiederherzustellen**

Stack während Bufferoverflow #3



Boshafte Nutzung von Bufferoverflow

- ❑ Eingabe String enthält binär Codierung des ausführbaren Codes
- ❑ Überschreibt Rücksprungadresse mit Adresse des Buffers
- ❑ Wenn bar() ret ausführt, erfolgt der Sprung zum Exploit (exploit code)



Exploits auf Bufferoverflowbasis

- ❑ Bufferoverflow-Bugs erlauben es, entfernten Maschinen, beliebigen Code auf der Maschine des Opfers auszuführen.
- ❑ Internet Wurm
 - Frühe Versionen des finger Servers (fingerd) benutzten gets() um die Argumente des Clienten zu lesen:
 - finger droh@cs.cmu.edu
 - Wurm attackierte den fingerd Server durch Senden eines falschen (phony) Arguments:
 - finger "exploit-code padding new-return-address"
 - Exploit-Code: führte eine „root shell“ mit einer direkten TCP- Verbindung zum Angreifer auf der Maschine des Opfers aus