

# Übungen zu Systemprogrammierung 2 (SP2)

## Ü5 – C und Sicherheit

Christoph Erhardt, Jens Schedel, Jürgen Kleinöder

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

WS 2013/14 – 16. bis 20. Dezember 2013

[http://www4.cs.fau.de/Lehre/WS13/V\\_SP2](http://www4.cs.fau.de/Lehre/WS13/V_SP2)

## Agenda

- 5.1 Stack-Aufbau eines Prozesses
- 5.2 Live-Hacking
- 5.3 Gegenmaßnahmen
- 5.4 Weihnachts-Hacking

25-Hacking\_handout



25-Hacking\_handout



## Agenda

- 5.1 Stack-Aufbau eines Prozesses
- 5.2 Live-Hacking
- 5.3 Gegenmaßnahmen
- 5.4 Weihnachts-Hacking

25-Hacking\_handout



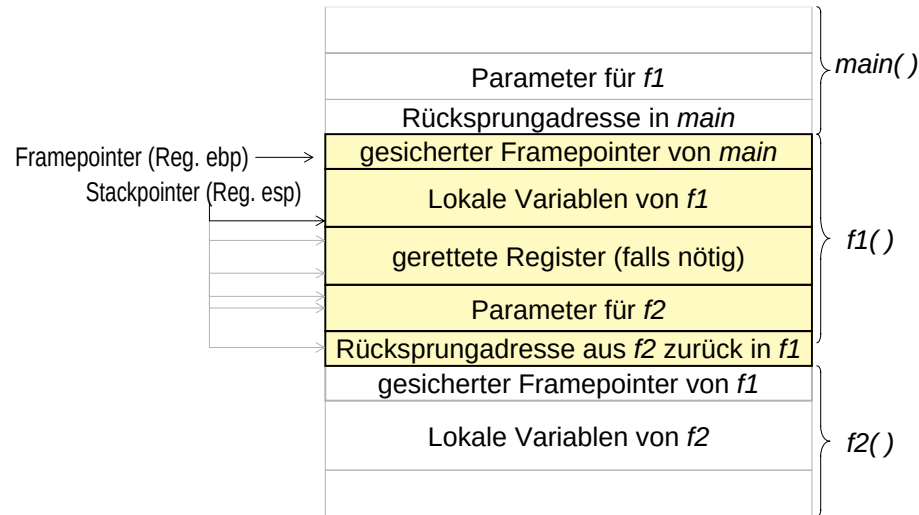
## Stack-Aufbau eines Prozesses

- Bei jedem Funktionsaufruf wird ein **Stack-Frame** angelegt, der u. a.
  - lokale Variablen der Funktion
  - Aufrufparameter an weitere Funktionen
  - gesicherte Register... enthält
- Beim Rücksprung wird dieser Stack-Frame wieder abgeräumt
- Stack-Organisation ist abhängig von:
  - Prozessorarchitektur
  - Compiler (auch von Version und Flags)
  - Betriebssystem
- Im Folgenden: Beispiel für Linux auf einem x86-Prozessor (32-Bit, typisch für CISC-Architektur)
  - Spezifikation: <http://sco.com/developers/devspecs/abi386-4.pdf>
  - RISC-Prozessoren mit Register-Files gehen anders vor

25-Hacking\_handout

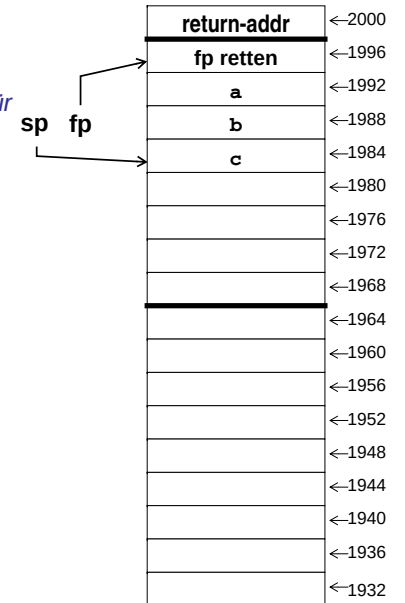


- Aufbau eines Stack-Frames (Funktionen `main()`, `f1()`, `f2()`):



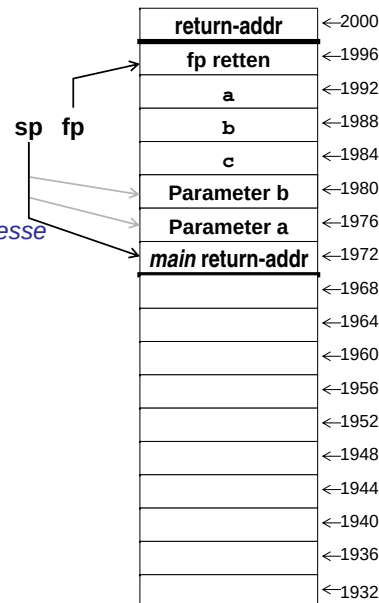
```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

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



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

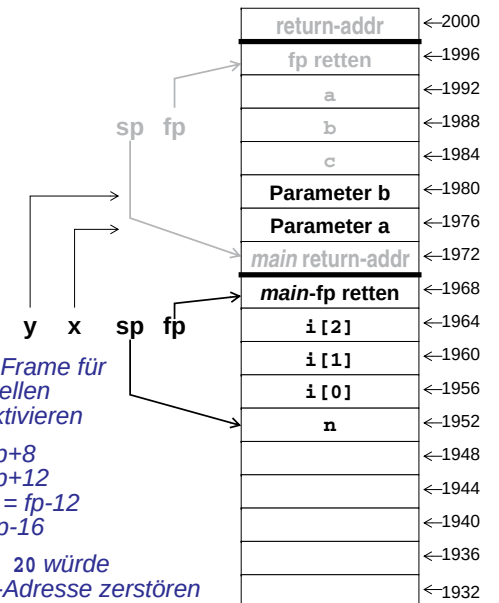
Parameter auf Stack legen  
 Bei Aufruf  
 Rücksprungadresse auf Stack legen



```
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-Adresse zerstören

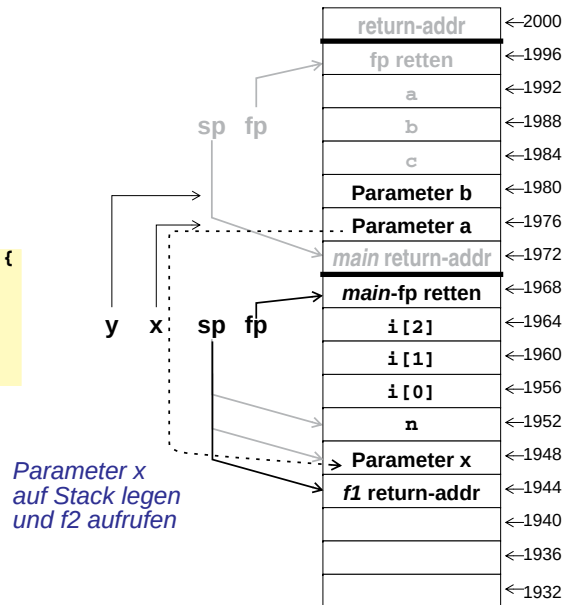


## Beispiel

### Stack-Aufbau eines Prozesses

```
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);
}
```



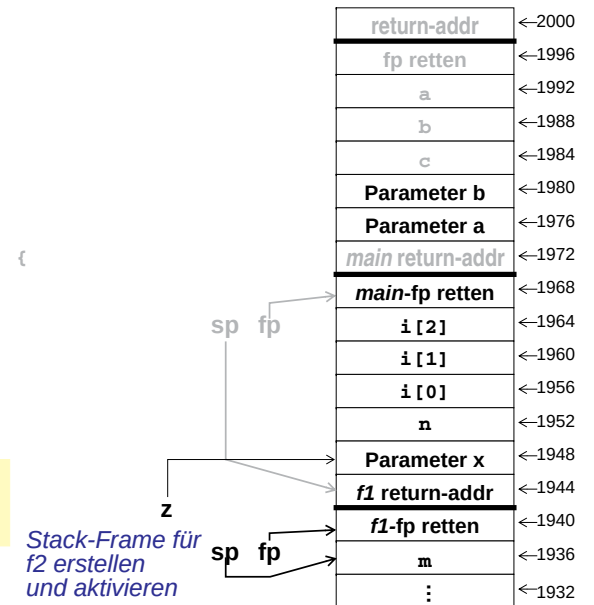
## Beispiel

### Stack-Aufbau eines Prozesses

```
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);
}
```



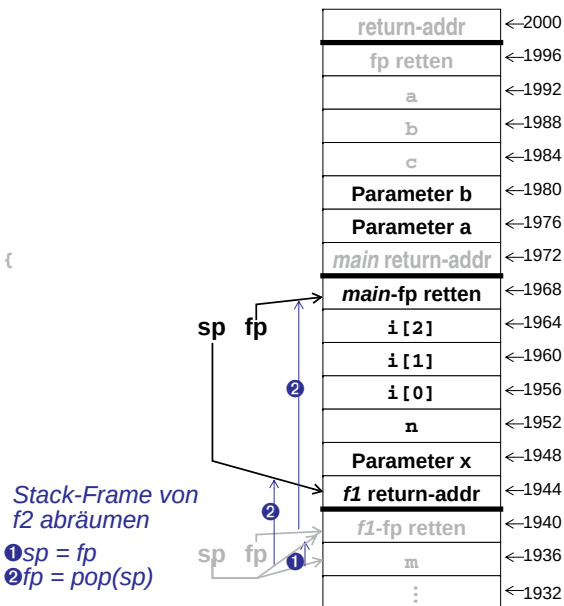
## Beispiel

### Stack-Aufbau eines Prozesses

```
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);
}
```



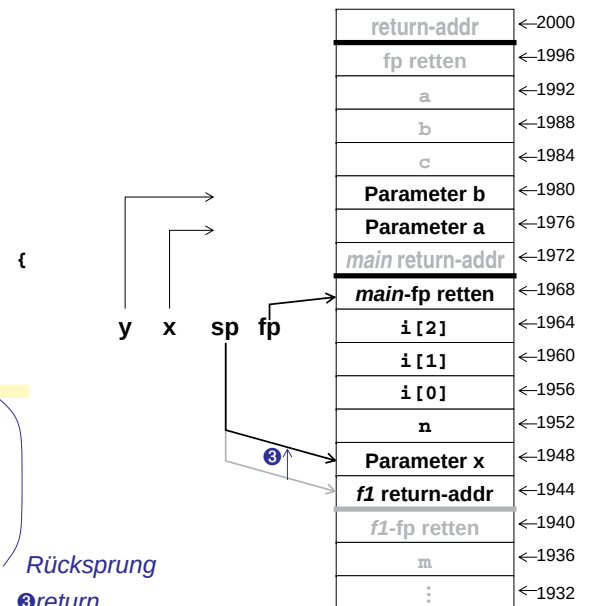
## Beispiel

### Stack-Aufbau eines Prozesses

```
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);
}
```



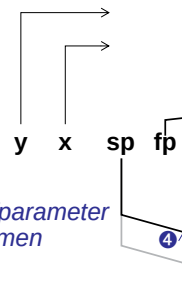
## Beispiel

### Stack-Aufbau eines Prozesses

```
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



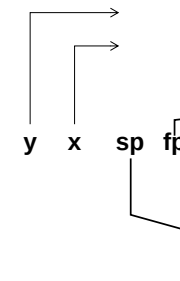
|                  |       |
|------------------|-------|
| return-addr      | ←2000 |
| fp retten        | ←1996 |
| a                | ←1992 |
| b                | ←1988 |
| c                | ←1984 |
| Parameter b      | ←1980 |
| Parameter a      | ←1976 |
| main return-addr | ←1972 |
| main-fp retten   | ←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 |

## Beispiel

### Stack-Aufbau eines Prozesses

```
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);
}
```



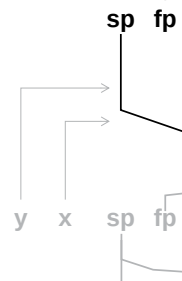
|                  |       |
|------------------|-------|
| return-addr      | ←2000 |
| fp retten        | ←1996 |
| a                | ←1992 |
| b                | ←1988 |
| c                | ←1984 |
| Parameter b      | ←1980 |
| Parameter a      | ←1976 |
| main return-addr | ←1972 |
| main-fp retten   | ←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 |

## Beispiel

### Stack-Aufbau eines Prozesses

```
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);
}
```



|                  |       |
|------------------|-------|
| return-addr      | ←2000 |
| fp retten        | ←1996 |
| a                | ←1992 |
| b                | ←1988 |
| c                | ←1984 |
| Parameter b      | ←1980 |
| Parameter a      | ←1976 |
| main return-addr | ←1972 |
| main-fp retten   | ←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 |

## Beispiel

### Stack-Aufbau eines Prozesses

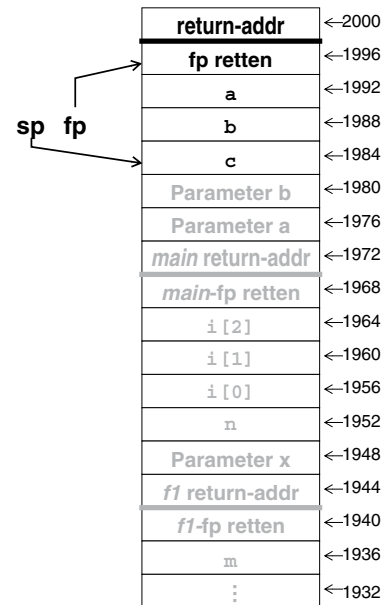
```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

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



|                  |       |
|------------------|-------|
| return-addr      | ←2000 |
| fp retten        | ←1996 |
| a                | ←1992 |
| b                | ←1988 |
| c                | ←1984 |
| Parameter b      | ←1980 |
| Parameter a      | ←1976 |
| main return-addr | ←1972 |
| main-fp retten   | ←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 |

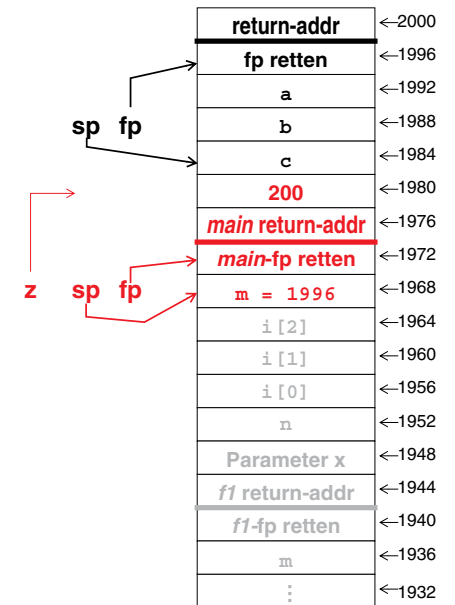
```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```



```
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 + m;
}
```



## Agenda

- 5.1 Stack-Aufbau eines Prozesses
- 5.2 Live-Hacking
- 5.3 Gegenmaßnahmen
- 5.4 Weihnachts-Hacking

## Live-Hacking

- Simples Authentifizierungs-Programm:
  1. Passwortabfrage
  2. Korrektes Passwort → Starten einer Shell
- Schaffen wir es die Shell zu starten, ohne das korrekte Passwort zu kennen?
- Code liegt in /proj/i4sp2/pub/hack-demo

## ■ Passwort-Authentifizierung:

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

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

    char password[8 + 1]; // Maximum: 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

- Pufferüberlauf innerhalb von askForPassword() hervorrufen
- Rücksprungadresse mit der Adresse der Funktion executeShell() überschreiben
- Shell benutzen und freuen :-)

## Analysieren des Code-Layouts

## ■ Wo im Textsegment liegen die Funktionen?

```
$ nm auth
0804876f r PASSWORD
08048750 r SHELL
08049890 d _DYNAMIC
08049984 d _GLOBAL_OFFSET_TABLE_
0804874c R _IO_stdin_used
      w _ITM_deregisterTMCloneTable
      w _ITM_registerTMCloneTable
      w __Jv_RegisterClasses
08048880 r __FRAME_END__
0804988c d __JCR_END__
0804988c d __JCR_LIST__
080499bc D __TMC_END__
080499bc A __bss_start
080499b4 D __data_start
08048580 t __do_global_dtors_aux
08049888 t __do_global_dtors_aux_fini_array_entry
080499b8 D __dso_handle
08049884 t __frame_dummy_init_array_entry
      w __gmon_start__
0804872a T __i686.get_pc_thunk.bx
08049888 t __init_array_end
08049884 t __init_array_start
      U __isoc99_scanf@GLIBC_2.7
080486c0 T __libc_csu_fini
080486d0 T __libc_csu_init
      U __libc_start_main@GLIBC_2.0
080499bc A _edata
080499e8 A _end
08048730 T _fini
08048748 R _fp_hw
0804840c T _init
080484e0 T _start
080485cc t askForPassword
080499e4 b completed.5730
080499b4 W data_start
08048510 t deregister_tm_clones
      U execl@GLIBC_2.0
0804862f t executeShell
      U exit@GLIBC_2.0
080485a0 t frame_dummy
      U fwrite@GLIBC_2.0
08048669 T main
      U perror@GLIBC_2.0
      U puts@GLIBC_2.0
08048540 t register_tm_clones
080499c0 B stderr@GLIBC_2.0
080499e0 B stdout@GLIBC_2.0
      U strcmp@GLIBC_2.0
```

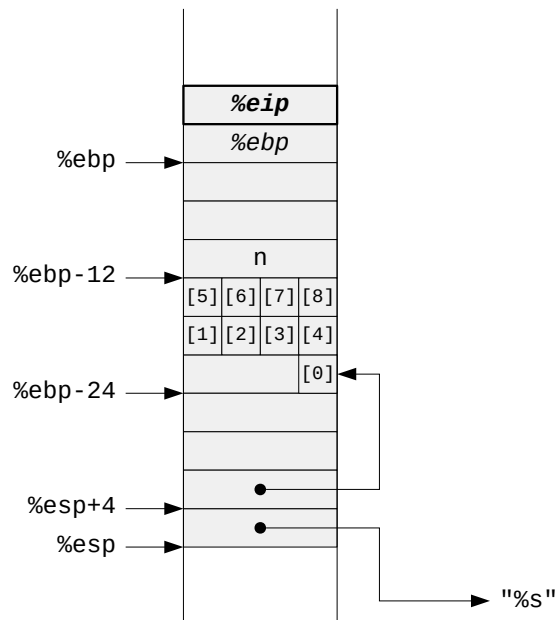
## Analysieren des Stack-Layouts

```
$ objdump -d auth
080485cc <askForPassword>:
80485cc: 55          push    %ebp
80485cd: 89 e5       mov     %esp,%ebp
80485cf: 83 ec 28    sub     $0x28,%esp
80485d2: a1 e0 99 04 08 mov     0x80499e0,%eax
80485d7: 89 44 24 0c mov     %eax,0xc(%esp)
80485db: c7 44 24 08 0a 00 00 movl    $0xa,0x8(%esp)
80485e2: 00
80485e3: c7 44 24 04 01 00 00 movl    $0x1,0x4(%esp)
80485ea: 00
80485eb: c7 04 24 75 87 04 08 movl    $0x8048775,(%esp)
80485f2: e8 79 fe ff ff call    8048470 <fwrite@plt>
80485f7: 8d 45 eb    lea     -0x15(%ebp),%eax
80485fa: 89 44 24 04 mov     %eax,0x4(%esp)
80485fe: c7 04 24 80 87 04 08 movl    $0x8048780,(%esp)
8048605: e8 c6 fe ff ff call    80484d0 <__isoc99_scanf@plt>
804860a: 89 45 f4    mov     %eax,-0xc(%ebp)
804860d: 83 7d f4 ff cmpl    $0xffffffff,-0xc(%ebp)
8048611: 75 07       jne     804861a <askForPassword+0x4e>
8048613: b8 ff ff ff ff mov     $0xffffffff,%eax
8048618: eb 13       jmp     804862d <askForPassword+0x61>
804861a: c7 44 24 04 6f 87 04 movl    $0x804876f,0x4(%esp)
8048621: 08
8048622: 8d 45 eb    lea     -0x15(%ebp),%eax
8048625: 89 04 24    mov     %eax,(%esp)
8048628: e8 23 fe ff ff call    8048450 <strcmp@plt>
804862d: c9         leave
804862e: c3         ret
```

Aufbauen des Stack-Frames

Lesen der Adresse von password

Schreiben von n



- Erzeugen eines manipulierenden Eingabe-Datenstroms mit Hilfe eines kleinen Programms, das
  - zuerst einen Bytestrom schickt, der zu einem Stack-Überlauf und dem fehlerhaften Rücksprung (und damit zum Aufruf von `executeShell()`) führt:
    - 9 Bytes fürs char-Array
    - 4 Bytes für Variable `n`
    - 12 Bytes für Füll-Slots und Frame-Pointer
    - 4 Bytes für die neue Rücksprungadresse `0x0804862f`
      - Byte-Order beachten!
    - 1 Byte `'\n'` zum Abschließen der Eingabe
  - anschließend alle Zeichen von `stdin` hinterherschickt (die bekommt dann die in `executeShell()` gestartete Shell)

- Starten des Hilfsprogramms und Umleiten der Ausgabe ins `auth`-Programm

**PWNED!**



- In unserem Beispiel ist der im Rahmen des Angriffs auszuführende Code bereits Bestandteil des Programms
- Gefährlichere Alternative:
  - Zusätzlich zu der Manipulation der Rücksprungadresse schickt man eigenen Maschinencode hinterher – und manipuliert die Rücksprungadresse so, dass sie auf den mitgeschickten Code im Stack zeigt
  - Falls die Stack-Adresse nur grob bekannt ist, baut man eine „Rutsche“ aus `NOP`-Instruktionen vor den eigentlichen Schadcode
- Übliches Ziel: auf dem angegriffenen Rechner eine fernsteuerbare Shell bekommen



- Pufferüberläufe sind nur eine von vielen möglichen Sicherheitslücken in C-Programmen
- Ganzzahlüber-/unterläufe:
 

```
// Lies width und height vom Benutzer
int *matrix = malloc(width * height * sizeof(*matrix));
// Befülle matrix mit Daten vom Benutzer
```

  - Falls `width * height > SIZE_MAX`, wird zu wenig Speicher für die Matrix alloziert!
  - Puffer auf dem Heap wird überlaufen
- Format-String-Angriffe:
 

```
// Lies string vom Benutzer
printf(string);
```



- 5.1 Stack-Aufbau eines Prozesses
- 5.2 Live-Hacking
- 5.3 Gegenmaßnahmen
- 5.4 Weihnachts-Hacking

- **Allerwichtigste Schutzmaßnahme ist das Bauen robuster Software!**
- Die folgenden Funktionen sind **absolut tabu** – man kann sie nicht korrekt verwenden:
  - `scanf("%s", buffer);`
    - Stattdessen: `char buffer[10]; scanf("%9s", buffer);`
  - `gets()`
    - Seit SUSv4 nicht mehr Teil der Standardbibliothek :-)
    - Stattdessen `fgets()` benutzen
- Nur mit Vorsicht zu genießen sind u. a. `strcpy()`, `strcat()`, `sprintf()` und eigene Schleifenkonstrukte
- Korrekte Implementierungsmöglichkeiten:
  1. Den Zielpuffer von vornherein mit der richtigen Größe anlegen
    - Wenn das geht, ist es immer der beste Weg!
  2. `snprintf()` benutzen
    - Alternativen `strncpy()`, `strncat()` haben keine wohldefinierte Semantik
    - Beispiel: `strncpy()` terminiert String nicht mit `'\0'`, falls Puffer zu klein :-)

## Technische Gegenmaßnahmen

- Fehlerfreie Software ist eine Utopie :-/
- Das Ausnutzen von Pufferüberläufen kann aber durch technische Maßnahmen immerhin erschwert werden

### Hardware-Ebene: *NX-Bit*

- Rechteverwaltung für Speicherseiten (rwx):
  - Prüfung jedes Speicherzugriffs durch die MMU
  - Sprung in eine als nicht ausführbar markierte Seite → **Trap**
  - Gängige Richtlinie: **W^X** – entweder schreiben oder ausführen
- Unterstützung in allen modernen CPU-Architekturen
  - Ausnahme: Intel x86 (vor x86\_64)
- Verhindert z. B. Ausführen von Schadcode auf Stack oder Heap
- Manipulierte Sprünge auf existierende Code-Sequenzen sind aber weiterhin möglich (*Return-Oriented Programming*)

## Technische Gegenmaßnahmen

### Betriebssystem-Ebene: *Address-Space Layout Randomisation*

- Zufällige Positionierung der Sektionen im logischen Adressraum
- Erschwert Angriffe, bei denen Adressen bekannt sein müssen
- Umsetzbarkeit:
  - Heap, Stack: bei allen Programmen möglich
  - Daten, BSS, Code: Programm muss als *Position-Independent Executable* kompiliert worden sein (-fPIE)

### Compiler-Ebene: *Canaries / Stack Cookies*

- Ablegen einer (zufälligen) magischen Zahl in jedem Stack-Frame
- Vor Rücksprung wird überprüft, ob der Wert verändert wurde
- Im GCC Aktivierung mit **-fstack-protector**



- 5.1 Stack-Aufbau eines Prozesses
- 5.2 Live-Hacking
- 5.3 Gegenmaßnahmen
- 5.4 Weihnachts-Hacking



- Shell-Server harsh (*Holey Assailable Remote Shell*):
  - Läuft auf Rechner faui00a.cs.fau.de, Port 10443
    - Verbindungen nur aus dem CIP-Netz (131.188.30.0/24)
    - Verbinden z. B. mit netcat: nc -q0 faui00a 10443
  - Startet nach Eingabe des richtigen Passworts eine einfache Shell: cash (*Castrated Shell*)
  - cash erlaubt Registrierung des eigenen Namens in einer *Hall of Fame*
- Quell- und Binärcode (32-Bit) in /proj/i4sp2/pub/harsh
- Teilnahme freiwillig, keine Bewertung
- Exploit basteln:
  1. Schwachstelle im Quellcode finden
  2. Binärcode analysieren (nm, objdump, gdb)
  3. Position und Layout der interessanten Daten und Codestücke herausfinden
  4. Manipulierten Datenstrom bauen und einschleusen
  5. ???
  6. PROFIT!



## Weihnachts-Hacking

- Bildbearbeitungs-Server i4s (*i4 Insecure Image Inversion Service*):
  - Details siehe /proj/i4sp2/pub/i4s/doc/readme.txt
- Beide Dienste voraussichtlich bis Semesterende erreichbar



## Frohe Weihnachten und einen guten Rutsch!



(Originalbild: <http://hackerstreefarm.com>)

