

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 2014 – 08. bis 12. Dezember 2014

https://www4.cs.fau.de/Lehre/WS14/V_SP2

Agenda

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



5.1 Stack-Aufbau eines Prozesses

5.2 Live-Hacking

5.3 Gegenmaßnahmen

5.4 Hacking



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



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

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

sp fp

return-addr	←2000
fp retten	←1996
a	←1992
b	←1988
c	←1984
	←1980
	←1976
	←1972
	←1968
	←1964
	←1960
	←1956
	←1952
	←1948
	←1944
	←1940
	←1936
	←1932



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

*Parameter
auf Stack legen*
*Bei Aufruf
Rücksprungadresse
auf Stack legen*

sp fp

return-addr	←2000
fp retten	←1996
a	←1992
b	←1988
c	←1984
Parameter b	←1980
Parameter a	←1976
main return-addr	←1972
	←1968
	←1964
	←1960
	←1956
	←1952
	←1948
	←1944
	←1940
	←1936
	←1932



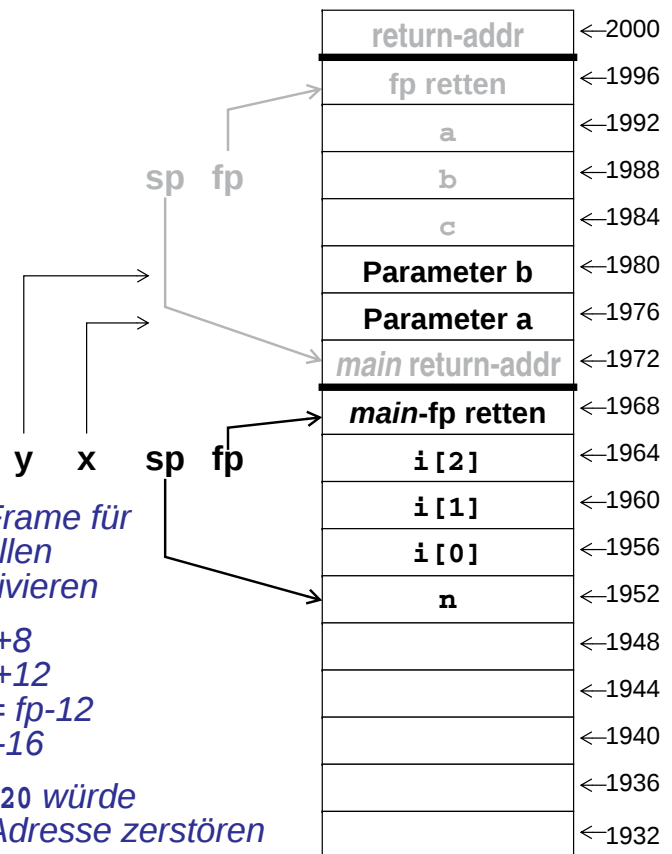
```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

```
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*

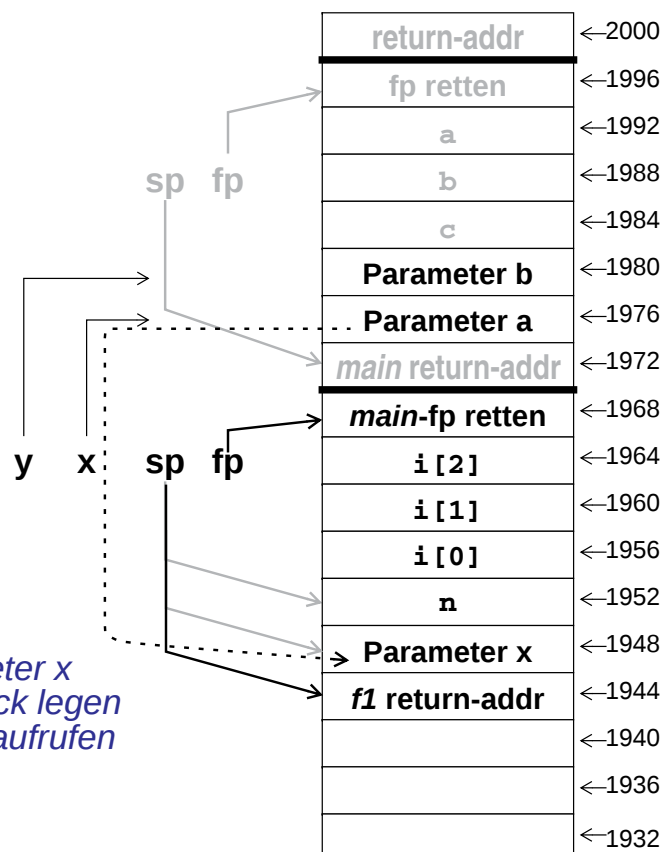
*i[4] = 20 würde
return-Adresse zerstören*



```
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*



```

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

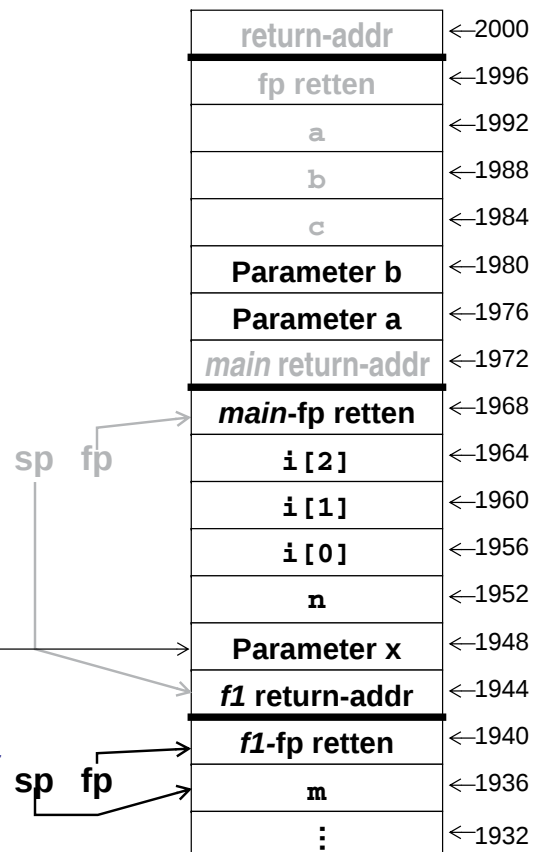
int f1(int x, int y) {
    int i[3];
    int n;

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

int f2(int z) {
    int m;
    m = 100;
    return(z+1);
}

```

Stack-Frame für
f2 erstellen
und aktivieren



```

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;

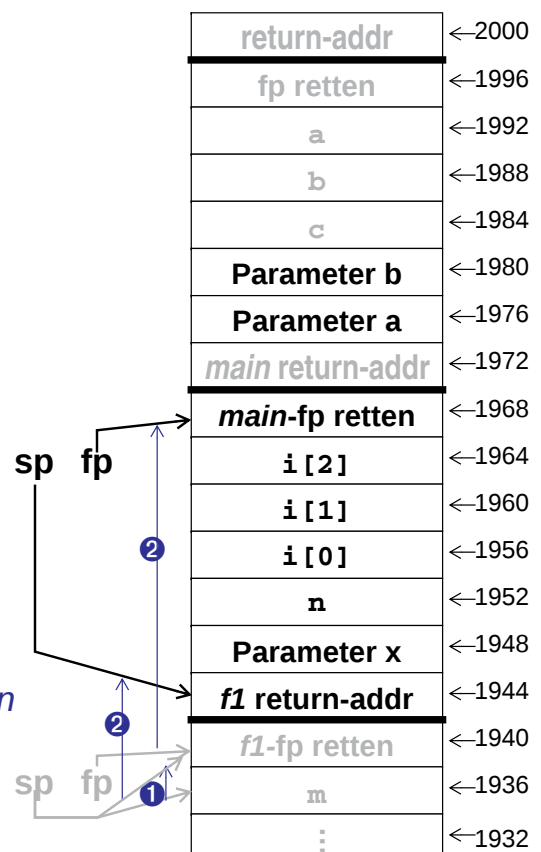
    x++;
    n = f2(x);
    return(n);
}

int f2(int z) {
    int m;
    m = 100;
    return(z+1);
}

```

Stack-Frame von
f2 abräumen

- ① sp = fp
- ② fp = pop(sp)



```

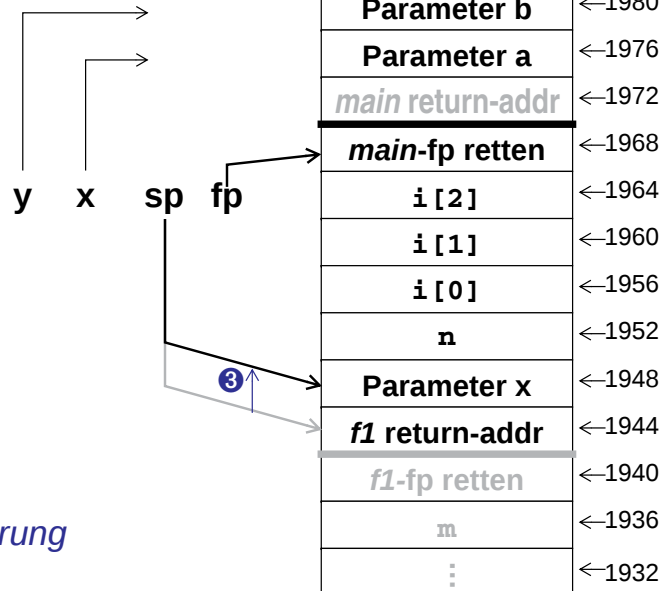
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}

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



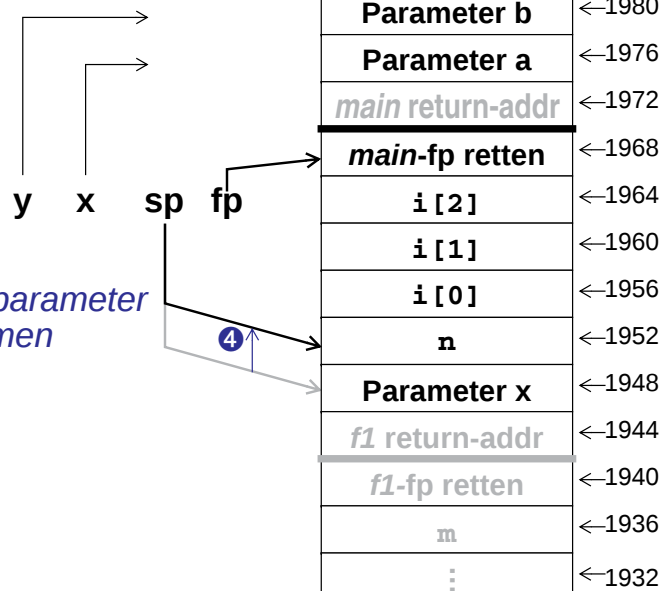
```

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

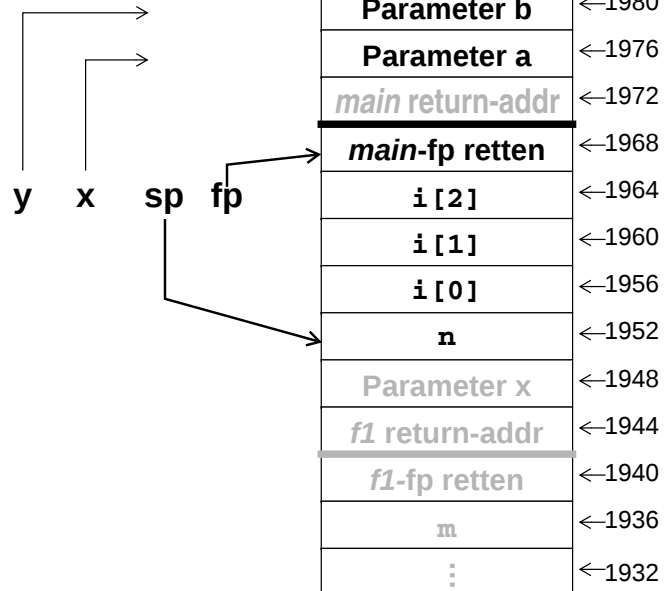


```

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);
}

```

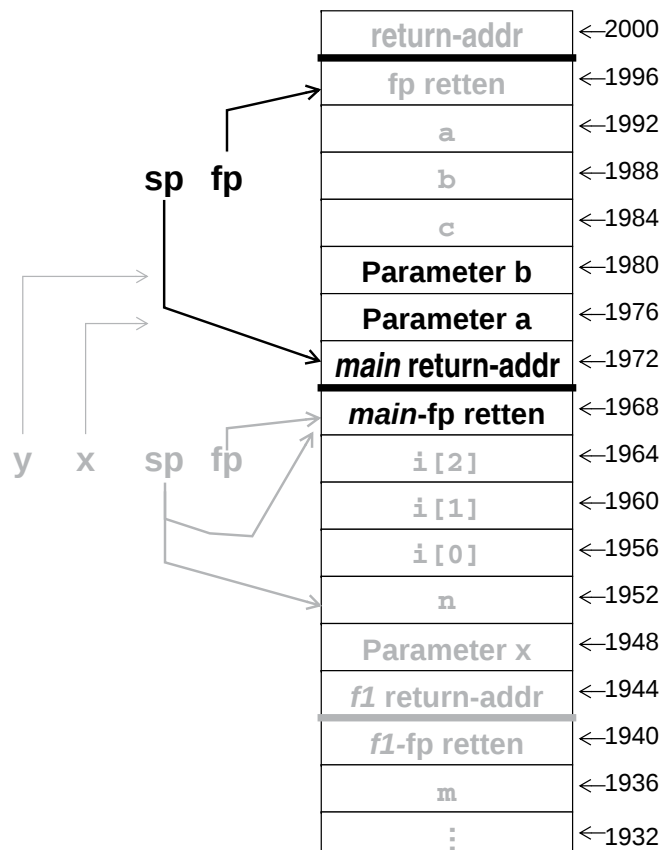


```

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);
}

```



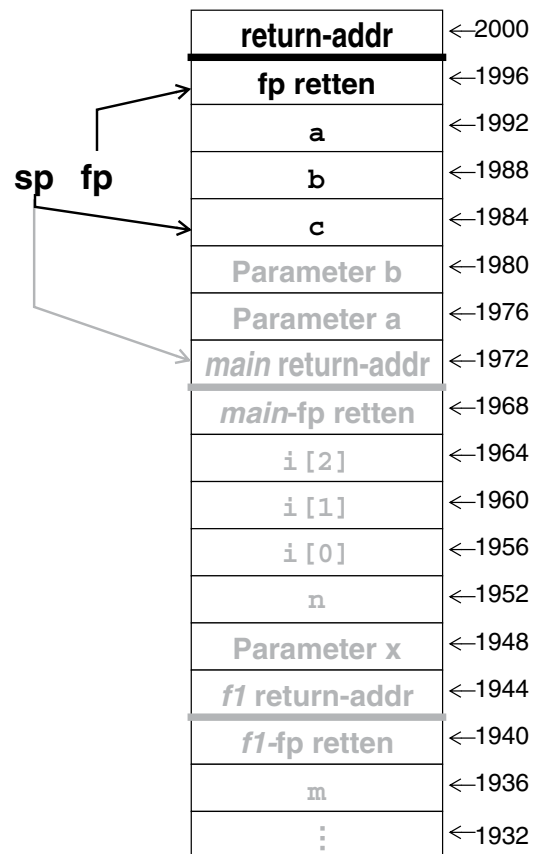
```

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);
}

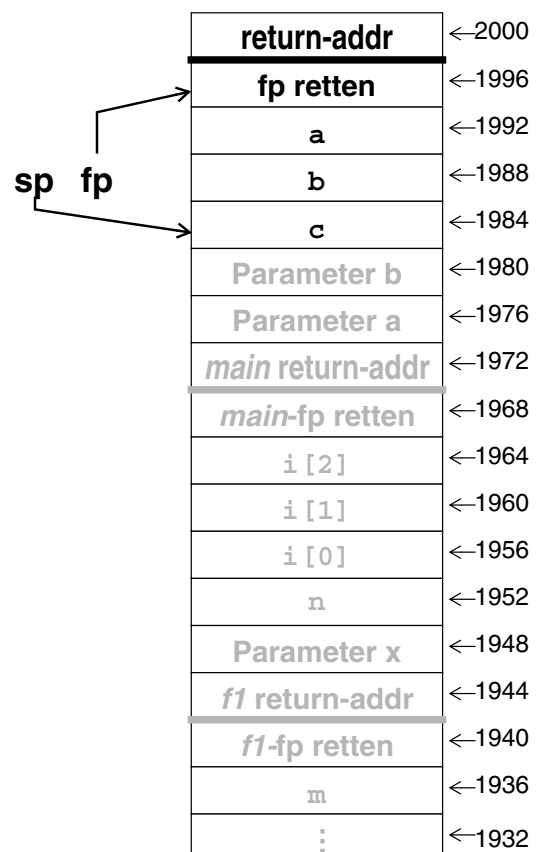
```



```

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

```



5.1 Stack-Aufbau eines Prozesses

5.2 Live-Hacking

5.3 Gegenmaßnahmen

5.4 Hacking



Live-Hacking

- Simple Authentifizierungs-Programm (z. B. einem Netzwerkdienst vorgeschaltet):
 1. Passwortabfrage
 2. Korrektes Passwort → Starten einer Shell
- Code liegt in `/proj/i4sp2/pub/hack-demo`
 - Ausführen mit Skript `run.sh`
- Schaffen wir es die Shell zu starten, ohne das korrekte Passwort zu kennen?



■ Passwort-Authentifizierung:

```
static int authenticate(void) {  
    fputs("Password: ", stdout);  
    fflush(stdout);  
  
    char password[8 + 1]; // Maximum: 8 characters and '\0'  
    int n = scanf("%s", password);  
    if (n == EOF)  
        return -1;  
  
    return checkPassword(password);  
}
```

■ scanf() überprüft nicht auf Pufferüberschreitung!

- Das Array password liegt auf dem Stack
- Nach dem Einlesen von 9 Zeichen überschreiben alle folgenden Zeichen andere Daten auf dem Stack



Angriffsplan

1. Pufferüberlauf innerhalb von `authenticate()` hervorrufen
2. Rücksprungadresse mit der Adresse der Funktion `executeShell()` überschreiben
3. Shell benutzen und freuen :-)



Wo im Textsegment liegen unsere Funktionen?

```
$ nm auth
080489e0 r PASSWD_FILE
08048a04 r SHELL
08049b8c d _DYNAMIC
08049c88 d _GLOBAL_OFFSET_TABLE_
080489c4 R _IO_stdin_used
          w _ITM_deregisterTMCloneTable
          w _ITM_registerTMCloneTable
          w _Jv_RegisterClasses
08048b7c r __FRAME_END__
08049b88 d __JCR_END__
08049b88 d __JCR_LIST__
08049cd8 D __TMC_END__
08049cd8 A __bss_start
08049cd0 D __data_start
08048730 t __do_global_dtors_aux
08049b84 t __do_global_dtors_aux_fini_array_entry
08049cd4 D __dso_handle
08049b80 t __frame_dummy_init_array_entry
          w __gmon_start__
0804899a T __i686.get_pc_thunk.bx
08049b84 t __init_array_end
08049b80 t __init_array_start
          U __isoc99_scanf@@GLIBC_2.7
08048930 T __libc_csu_fini
08048940 T __libc_csu_init
          U __libc_start_main@@GLIBC_2.0
08049cd8 A _edata
08049ce8 A _end
080489a0 T _fini
080489c0 R _fp_hw
08048568 T _init
08048690 T _start
0804884c t authenticate
0804877c t checkPassword
08049ce4 b completed.5730
          U crypt@@GLIBC_2.0
08049cd0 W data_start
080486c0 t deregister_tm_clones
          U execl@@GLIBC_2.0
080488b4 t executeShell
          U exit@@GLIBC_2.0
          U fclose@@GLIBC_2.1
          U ferror@@GLIBC_2.0
          U fflush@@GLIBC_2.0
          U fgetpwent@@GLIBC_2.0
          U fopen@@GLIBC_2.1
08048750 t frame_dummy
          U fwrite@@GLIBC_2.0
080488ee T main
          U perror@@GLIBC_2.0
          U puts@@GLIBC_2.0
080486f0 t register_tm_clones
08049ce0 B stdout@@GLIBC_2.0
          U strcmp@@GLIBC_2.0
```

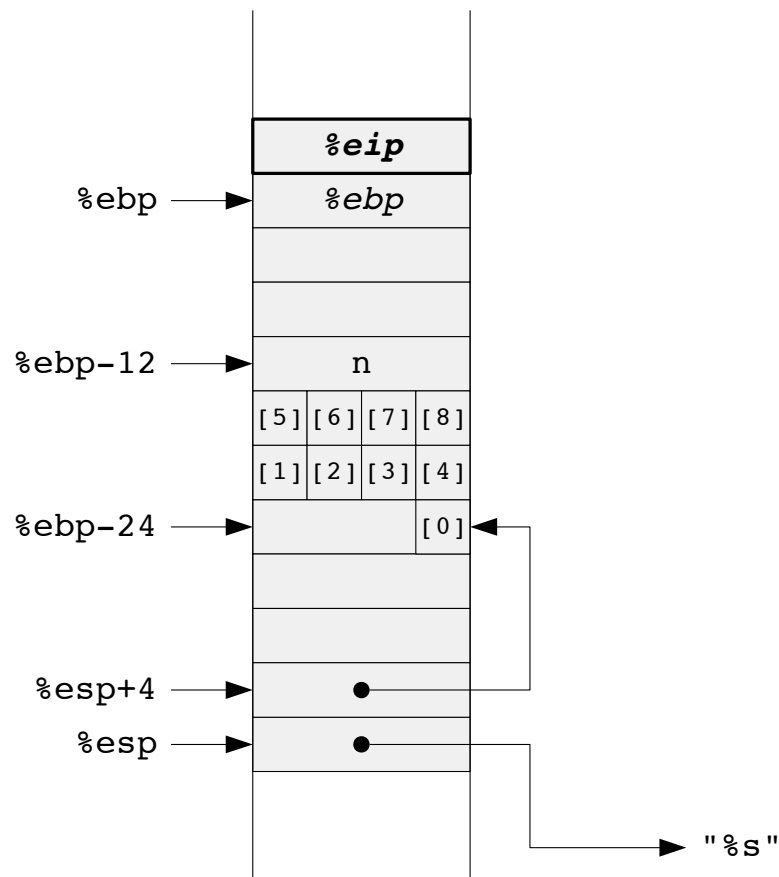
Analysieren des Stack-Layouts

```
$ objdump -d auth
0804884c <authenticate>:
0804884c: 55          push    %ebp
0804884d: 89 e5       mov     %esp,%ebp
0804884f: 83 ec 28    sub     $0x28,%esp
08048852: a1 a0 9c 04 08 mov     0x8049ca0,%eax
08048857: 89 44 24 0c mov     %eax,0xc(%esp)
0804885b: c7 44 24 08 0a 00 00 movl    $0xa,0x8(%esp)
08048862: 00
08048863: c7 44 24 04 01 00 00 movl    $0x1,0x4(%esp)
0804886a: 00
0804886b: c7 04 24 fe 89 04 08 movl    $0x80489fe,(%esp)
08048872: e8 79 fd ff ff call    80485f0 <fwrite@plt>
08048877: a1 a0 9c 04 08 mov     0x8049ca0,%eax
0804887c: 89 04 24    mov     %eax,(%esp)
0804887f: e8 2c fd ff ff call    80485b0 <fflush@plt>
08048884: 8d 45 eb    lea     -0x15(%ebp),%eax
08048887: 89 44 24 04 mov     %eax,0x4(%esp)
0804888b: c7 04 24 09 8a 04 08 movl    $0x8048a09,(%esp)
08048892: e8 e9 fd ff ff call    8048680 <__isoc99_scanf@plt>
08048897: 89 45 f4    mov     %eax,-0xc(%ebp)
0804889a: 83 7d f4 ff cmpl    $0xffffffff,-0xc(%ebp)
0804889e: 75 07       jne     80488a7 <authenticate+0x5b>
080488a0: b8 ff ff ff ff mov     $0xffffffff,%eax
080488a5: eb 0b       jmp     80488b2 <authenticate+0x66>
080488a7: 8d 45 eb    lea     -0x15(%ebp),%eax
080488aa: 89 04 24    mov     %eax,(%esp)
080488ad: e8 ca fe ff ff call    804877c <checkPassword>
080488b2: c9         leave
080488b3: c3         ret
```

Aufbauen des Stack-Frames

Lesen der Adresse von password

Schreiben von n



Ausnutzen des Pufferüberlaufs

- Manipulierenden Eingabe-Datenstrom mit Hilfe eines kleinen Programms erzeugen, das
 - zuerst eine Bytesequenz schickt, die zu Stack-Überlauf und fehlerhaftem 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 `0x080488b4`
→ Byte-Order beachten!
 - 1 Byte `'\n'` zum Abschließen der Eingabe
 - anschließend alle Zeichen von `stdin` hinterherschickt (die bekommt dann die in `executeShell()` gestartete Shell)
- Hilfsprogramm starten und Ausgabe an den `auth`-Prozess senden



- 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



Weitere Einfallstore

- 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));
// Befuelle matrix mit Daten vom Benutzer
```

- Falls `width * height * sizeof(*matrix) > 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);
```

- Benutzer kann `printf()` einen beliebigen Format-String unterjubeln
- Durch geschicktes Einfügen von %-Platzhaltern kann er beliebige Stack-Inhalte auslesen und u. U. beliebige Speicherinhalte überschreiben



5.1 Stack-Aufbau eines Prozesses

5.2 Live-Hacking

5.3 Gegenmaßnahmen

5.4 Hacking



Vermeiden von Pufferüberläufen in C Gegenmaßnahmen

- **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 Hacking



Hacking

- Shell-Server **harsh** (*Holey Assailable Remote Shell*):
 - Läuft auf Rechner **fau100a.cs.fau.de**, Port 10443
 - Verbindungen nur aus dem CIP-Netz (131.188.30.0/24)
 - Verbinden z. B. mit netcat: `nc -q0 fau100a 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!



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



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

