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

SS 13 – 1. bis 5. Juli 2013

http://www4.cs.fau.de/Lehre/SS13/V_SP2

Agenda

25-Hacking_handout

Agenda

25-Hacking_handout

Stackaufbau eines Prozesses

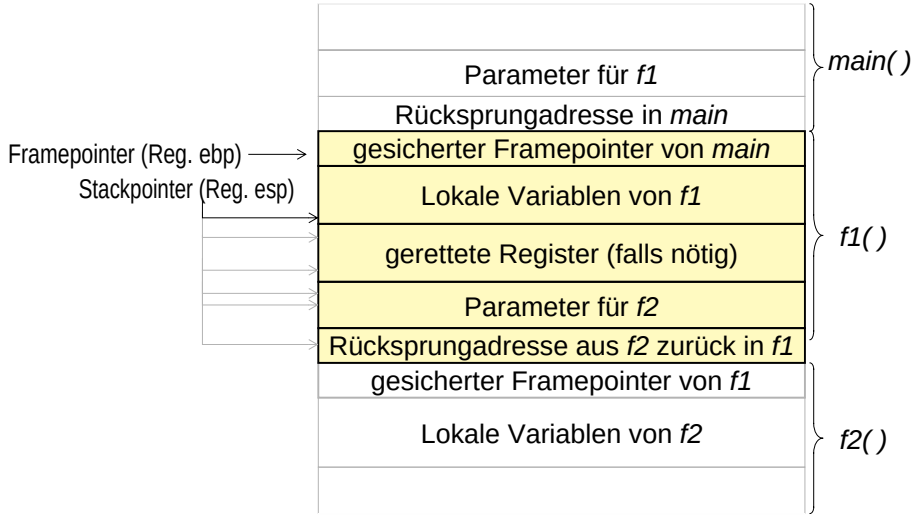
- Bei jedem Funktionsaufruf wird ein Stack-Frame angelegt, der u. a.
  - lokale Variablen der Funktion
  - Aufrufparameter an weitere Funktionen
  - gesicherte Register

  enthält
- Beim Rücksprung wird dieser Stack-Frame wieder abgeräumt
- Stackorganisation ist abhängig von:
  - Prozessorarchitektur
  - Compiler (auch von Version und Flags)
  - Betriebssystem
- Beispiele aus einem UNIX auf einem x86-Prozessor (32-Bit, typisch für CISC-Architektur)
  - RISC-Prozessoren mit Registerfiles gehen anders vor

25-Hacking_handout

Beispiel

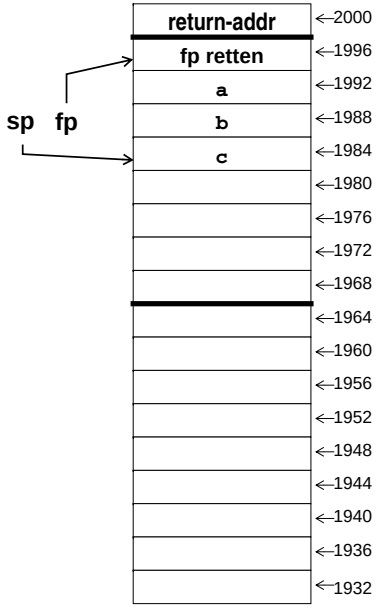
- Aufbau eines Stack-Frames (Funktionen `main()`, `f1()`, `f2()`):



Beispiel

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

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

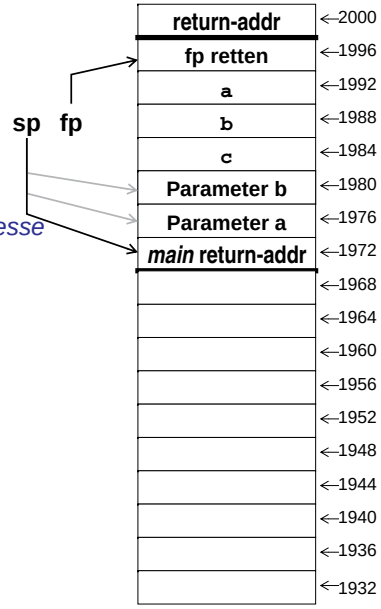
Stack-Frame für main erstellen
&a = fp - 4
&b = fp - 8
&c = fp - 12



Beispiel

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

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

Parameter auf Stack legen
Bei Aufruf Rücksprungadresse auf Stack legen



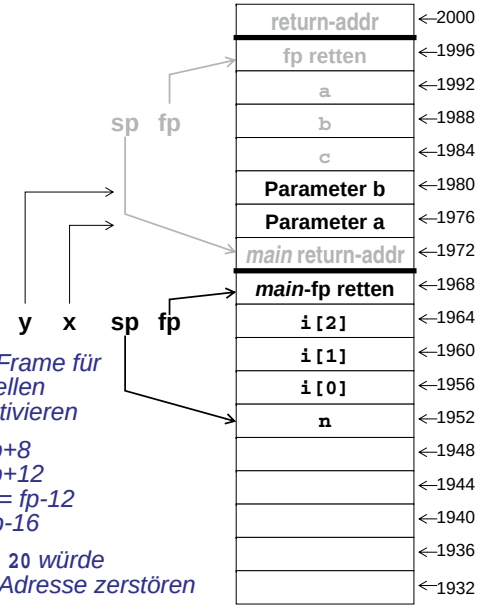
Beispiel

```
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);
}
```

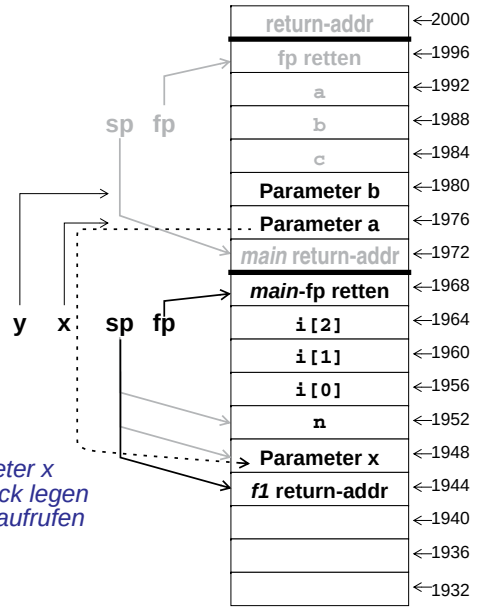
Stack-Frame für f1 erstellen und aktivieren

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

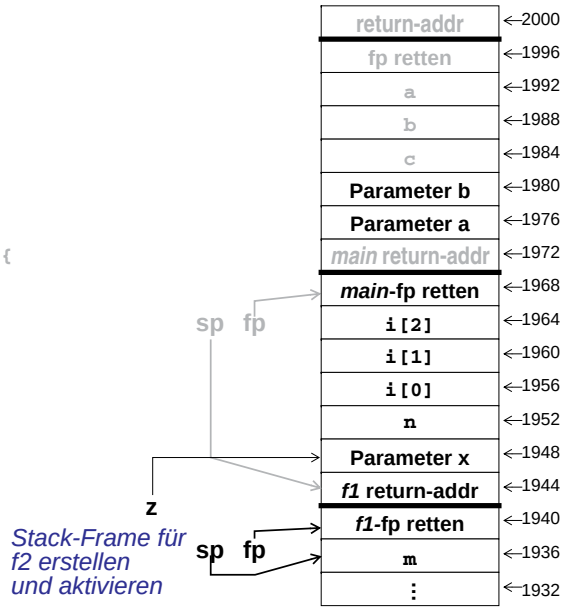
`i[4] = 20` *würde return-Adresse zerstören*



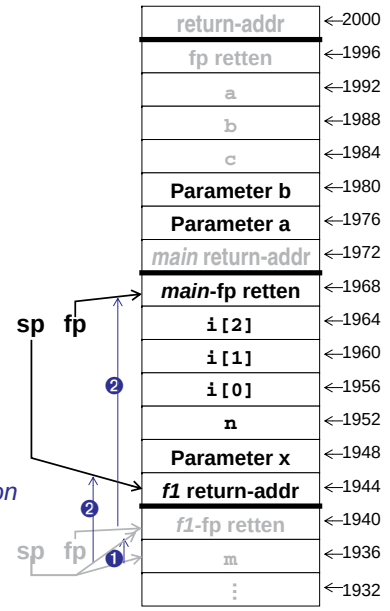
```
main() {
        int a, b, c;

        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);
}
```



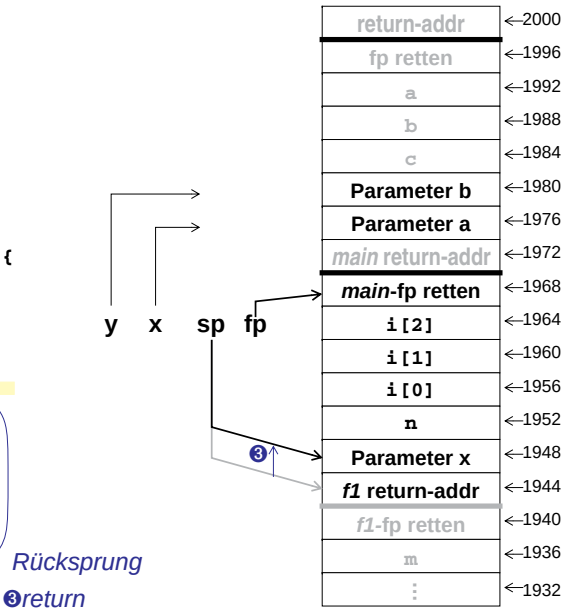

```
main() {
        int a, b, c;

        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);
}
```



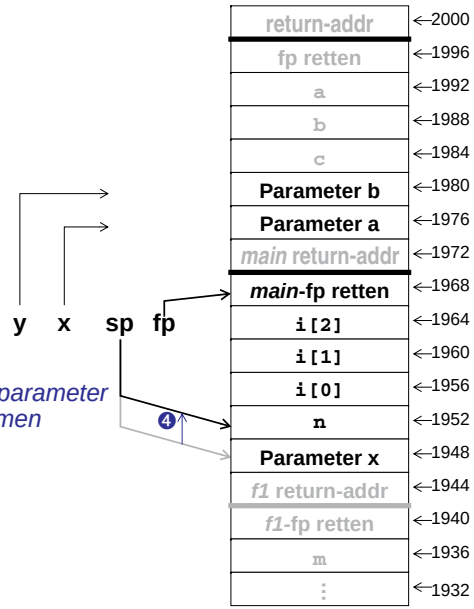

```
main() {
        int a, b, c;

        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);
}
```



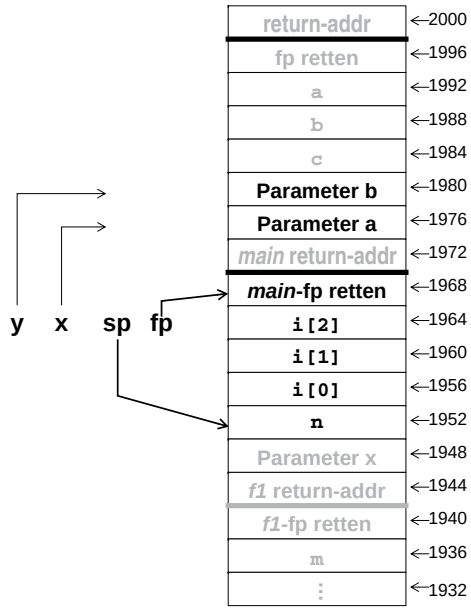

```
main() {
        int a, b, c;

        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);
}
```



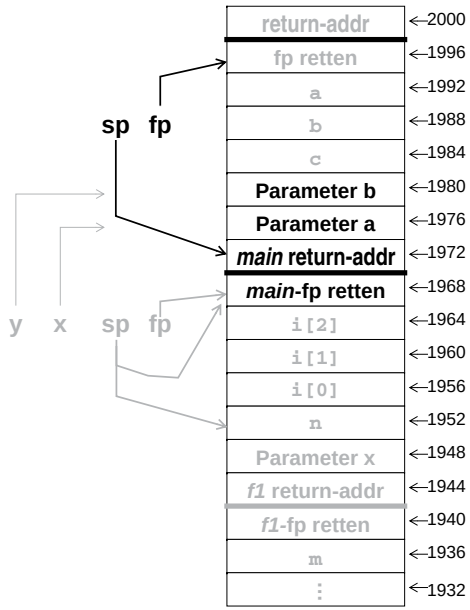
```
main() {
        int a, b, c;

        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);
}
```



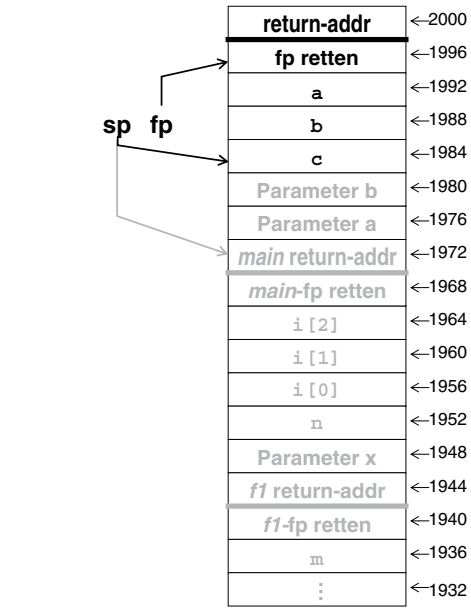

```
main() {
        int a, b, c;

        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);
}
```



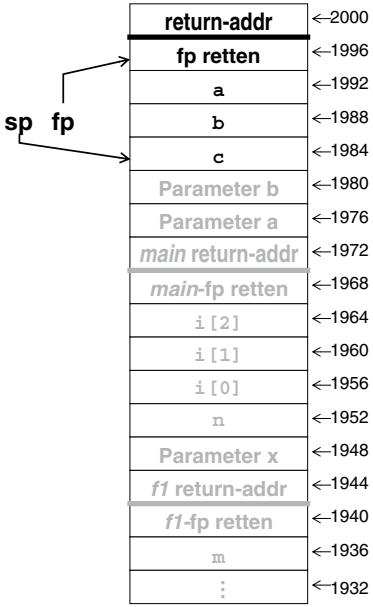

```
main() {
        int a, b, c;

        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);
}


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

        x++;

        n = f2(x);

        return(n);
}
```



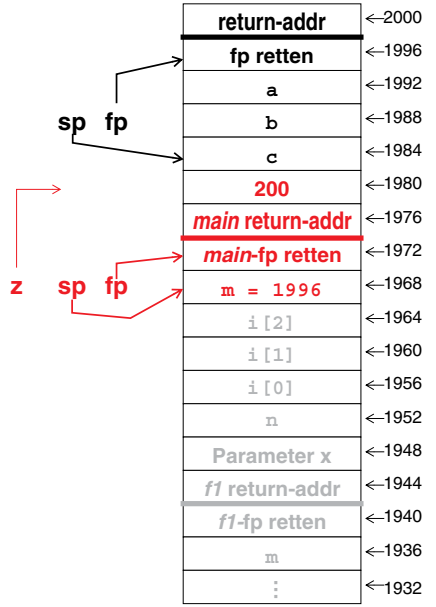
```
main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```




```
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

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`

Live-Hacking

- 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

Angriffsvektor

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

Analysieren des Code-Layouts

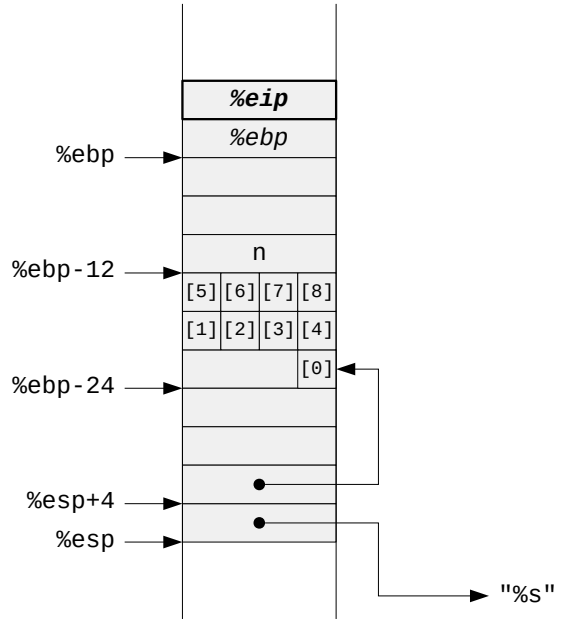
- Wo im Textsegment liegen die Funktionen?

```
$ nm auth
08048750 r PASSWORD                                    080486d0 T __libc_csu_init
08048758 r SHELL                                               U __libc_start_main@@GLIBC_2.0
08049894 d _DYNAMIC                                    080499c0 A _edata
08049988 d _GLOBAL_OFFSET_TABLE_                       080499e8 A _end
0804874c R _IO_stdin_used                              08048730 T _fini
         w _ITM_deregisterTMCloneTable                 08048748 R _fp_hw
         w _ITM_registerTMCloneTable                   0804840c T _init
         w _Jv_RegisterClasses                         080484e0 T _start
08048884 r __FRAME_END__                               080485cc t askForPassword
08049890 d __JCR_END__                                 080499e4 b completed.5730
08049890 d __JCR_LIST__                                080499b8 W data_start
080499c0 D __TMC_END__                                 08048510 t deregister_tm_clones
080499c0 A __bss_start                                         U execl@@GLIBC_2.0
080499b8 D __data_start                                0804862f t executeShell
08048580 t __do_global_dtors_aux                               U exit@@GLIBC_2.0
0804988c t __do_global_dtors_aux_fini_array_entry      080485a0 t frame_dummy
080499bc D __dso_handle                                        U fwrite@@GLIBC_2.0
08049888 t __frame_dummy_init_array_entry              08048669 T main
         w __gmon_start__                                      U perror@@GLIBC_2.0
0804872a T __i686.get_pc_thunk.bx                              U puts@@GLIBC_2.0
0804988c t __init_array_end                            08048540 t register_tm_clones
08049888 t __init_array_start                          080499c0 B stderr@@GLIBC_2.0
         U __isoc99_scanf@@GLIBC_2.7                   080499e0 B stdout@@GLIBC_2.0
080486c0 T __libc_csu_fini                                     U strcmp@@GLIBC_2.0
```


Analysieren des Stack-Layouts


```
$ objdump -d auth
080485cc <askForPassword>:
 80485cc:  55                    push   %ebp
 80485cd:  89 e5                 mov    %esp,%ebp              Aufbauen des Stack-Frames
 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 77 87 04 08  movl   $0x8048777,(%esp)
 80485f2:  e8 79 fe ff ff        call   8048470 <fwrite@plt>
 80485f7:  8d 45 eb              lea    -0x15(%ebp),%eax       Lesen von password
 80485fa:  89 44 24 04           mov    %eax,0x4(%esp)
 80485fe:  c7 04 24 82 87 04 08  movl   $0x8048782,(%esp)
 8048605:  e8 c6 fe ff ff        call   80484d0 <__isoc99_scanf@plt>
 804860a:  89 45 f4              mov    %eax,-0xc(%ebp)        Schreiben von n
 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 50 87 04  movl   $0x8048750,0x4(%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
```

Stack-Layout beim Aufruf von `scanf()`



Ausnutzen des Pufferüberlaufs

- 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

Einschleusen von Schadcode

- 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 *NOP-Rutsche* vor den eigentlichen Schadcode
- Übliches Ziel: auf dem angegriffenen Rechner eine fernsteuerbare Shell bekommen

Agenda

Vermeiden von Pufferüberläufen in C

- 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()`
    - 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. `strncpy()`, `strncat()`, `snprintf()` benutzen
     - **Aber Vorsicht vor Fallstricken** – Man-Pages genau durchlesen!
     - Beispiel: `strncpy()` terminiert den String nicht mit `'\0'`, falls der Zielpuffer zu klein ist :-(

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`)

Programm-Ebene: *Canaries / Stack Cookies*

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

Agenda

„Weihnachts“-Hacking

- Shell-Server `harsh` (*Holey Assailable Remote Shell*):
  - Verfügbar spätestens ab 8. Juli
  - 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. 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*):
  - Verfügbar spätestens ab 8. Juli
  - Details siehe `/proj/i4sp2/pub/i4s/doc/readme.txt`
- Beide Dienste voraussichtlich bis Mitte August erreichbar

Frohe Weihnachten und einen guten Rutsch!

(Bild: `http://hackerstreefarm.com`)