

Übungen zu Systemprogrammierung 2

ÜH – C und Sicherheit

Wintersemester 2025/26

Jürgen Kleinöder, Thomas Preisner, Tobias Häberlein, Ole Wiedemann

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät



H.1 Stack-Aufbau eines Prozesses

H.2 Live-Hacking

H.3 Gegenmaßnahmen

H.4 Hacking



H.1 Stack-Aufbau eines Prozesses

H.2 Live-Hacking

H.3 Gegenmaßnahmen

H.4 Hacking

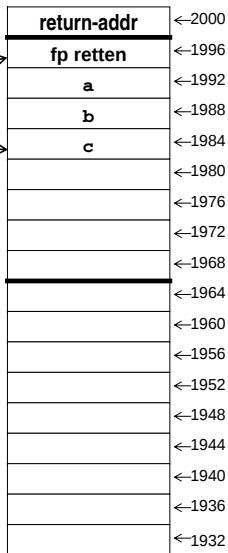


- Bei jedem Funktionsaufruf wird ein **Stack-Frame** angelegt, der u. a.
 - lokale Variablen der Funktion
 - Aufrufparameter an weitere Funktionen
 - gesicherte Register... enthält
- Beim Rücksprung wird dieser Stack-Frame wieder abgeräumt
- Stack-Organisation ist abhängig von:
 - Prozessorarchitektur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

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

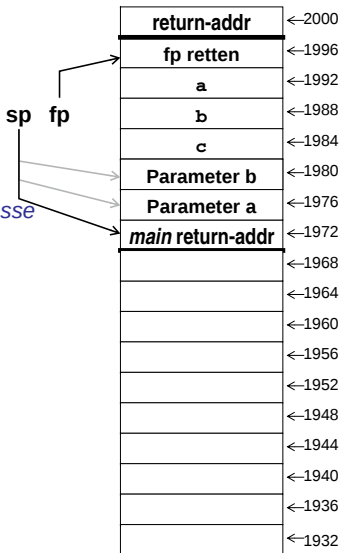
*Stack-Frame für
main erstellen*
 $\&a = fp - 4$
 $\&b = fp - 8$
 $\&c = fp - 12$

sp fp



```
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++;

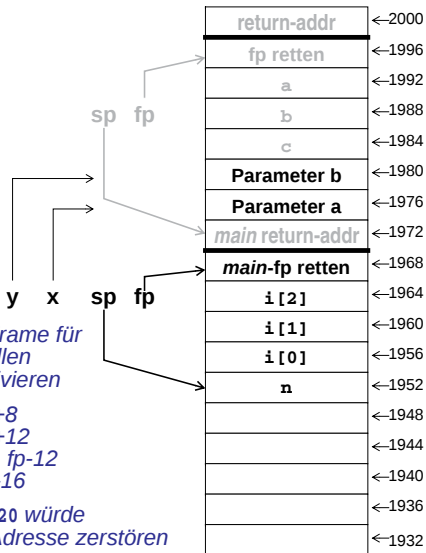
    n = f2(x);

    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*



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

    a = 10;
    b = 20;

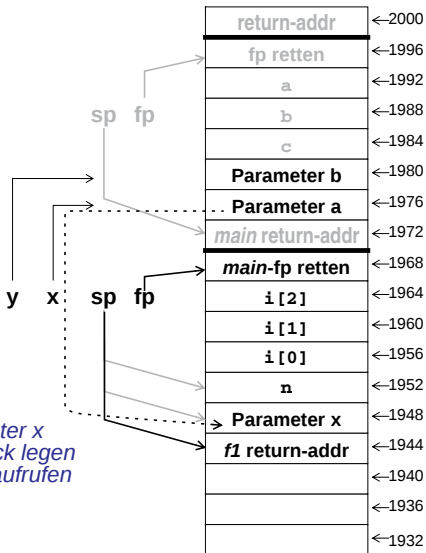
    f1(a, b);

    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;

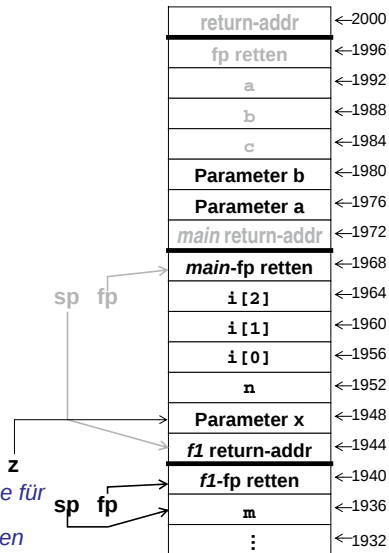
    x++;
    n = f2(x);
    return(n);
}
```

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

    m = 100;

    return(z+1);
}
```

*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;

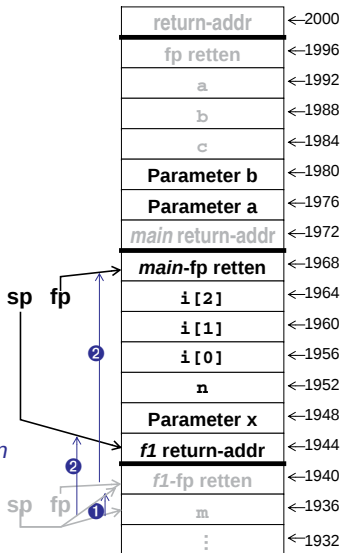
    x++;
    n = f2(x);
    return(n);
}
```

```
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

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 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

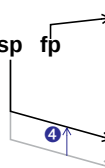
```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

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

④ Aufrufparameter
abräumen

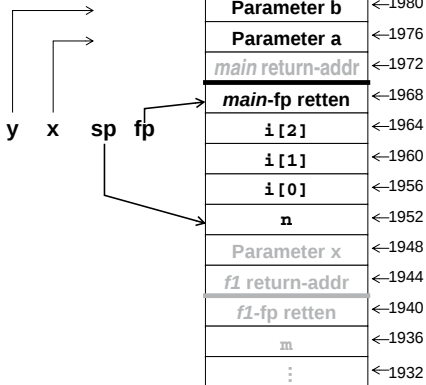
y x sp fp

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



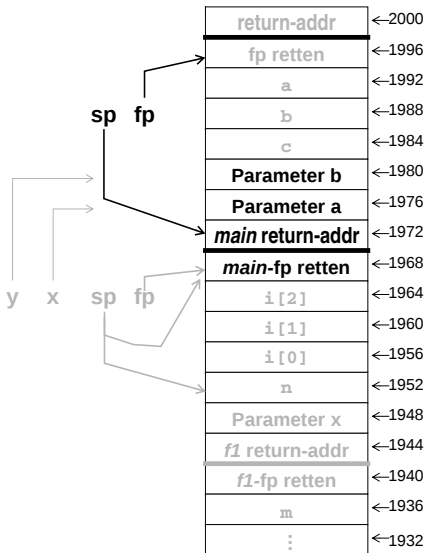
```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

```
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);
}
```

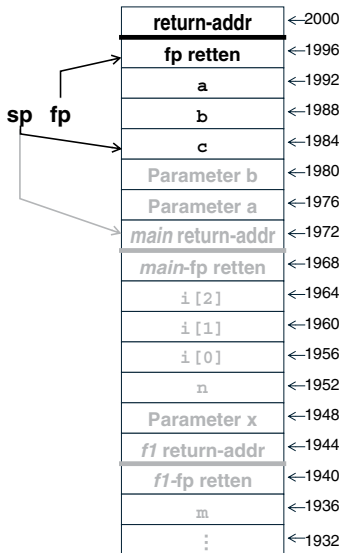
```
int f1(int x, int y) {
    int i[3];
    int n;
    x++;
    n = f2(x);
    return(n);
}
```



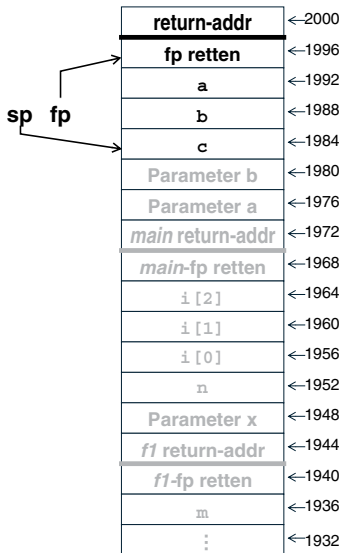
```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

```
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);
}
```





H.1 Stack-Aufbau eines Prozesses

H.2 Live-Hacking

H.3 Gegenmaßnahmen

H.4 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`
- 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(3p) 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

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
```

```
08049868 t execute_stack_op.cold
08049871 t .L92
0804987a t uw_update_context_1.cold
0804988c t execute_cfa_program_specialized.cold
0804988c t .L321
08049894 t execute_cfa_program_generic.cold
08049894 t .L460
0804989c t uw_frame_state_for.cold
080498a5 t uw_init_context_1.cold
080498aa t _Unwind_RaiseException_Phase2.cold
080498b3 t _Unwind_ForcedUnwind_Phase2.cold
080498bc t _Unwind_GetGR.cold
080498c1 t _Unwind_SetGR.cold
080498c8 t _Unwind_RaiseException.cold
080498d3 t _Unwind_Resume.cold
080498de t _Unwind_Resume_or_Rethrow.cold
080498e3 t _Unwind_Backtrace.cold
080498ee t .L49
080498ee t read_encoded_value_with_base.cold
080498f5 t fde_mixed_encoding_extract.cold
080498fe t classify_object_over_fdes.cold
08049907 t __deregister_frame_info_bases.part.0.cold
0804990c t fde_single_encoding_extract.cold
08049911 t fde_single_encoding_compare.cold
08049918 t fde_mixed_encoding_compare.cold
0804991f t add_fdes.isra.0.cold
08049928 t linear_search_fdes.cold
```

```
08049931 t _Unwind_Find_FDE.cold
0804993a t base_of_encoded_value.cold
0804993f t .L17
0804993f t read_encoded_value_with_base.cold
08049946 t __gcc_personality_v0.cold
08049950 t btree_release_tree_recursively
080499a0 t btree_destroy
080499f0 t release_registered_frames
08049a30 t _IO_stdfiles_init
08049a60 T _start
08049a8d t __wrap_main
08049aa0 T _dl_relocate_static_pie
08049ab0 T __x86.get_pc_thunk.bx
08049ac0 t deregister_tm_clones
08049b00 t register_tm_clones
08049b40 t __do_global_dtors_aux
08049b90 t frame_dummy
08049bd5 t checkPassword
08049cbb t authenticate
08049d1e t executeShell
08049d55 T main
08049db0 T crypt
08049dd6 T __x86.get_pc_thunk.ax
08049de0 t get_hashfn
08049e80 t do_crypt
08049fc0 T crypt_rn
```

```
$ objdump -d auth
08049cbb <authenticate>:
 8049cbb: 55
 8049cbc: 89 e5
 8049cbe: 83 ec 18
 8049cc1: a1 50 94 14 08
 8049cc6: 50
 8049cc7: 6a 0a
 8049cc9: 6a 01
 8049ccb: 68 9a 40 0f 08
 8049cd0: e8 ab b2 02 00
 8049cd5: 83 c4 10
 8049cd8: a1 50 94 14 08
 8049cdd: 83 ec 0c
 8049ce0: 50
 8049ce1: e8 ca af 02 00
 8049ce6: 83 c4 10
 8049ce9: 83 ec 08
 8049cec: 8d 45 eb
 8049cef: 50
 8049cf0: 68 a5 40 0f 08
 8049cf5: e8 06 2c 02 00
 8049cfa: 83 c4 10
 8049cfd: 89 45 f4
 8049d00: 83 7d f4 ff
 8049d04: 75 07
 8049d06: b8 ff ff ff ff
 8049d0b: eb 0f
 8049d0d: 83 ec 0c
 8049d10: 8d 45 eb
 8049d13: 50
 8049d14: e8 bc fe ff ff
 8049d19: 83 c4 10
 8049d1c: c9
 8049d1d: c3
```

```
push    %ebp
mov     %esp,%ebp
sub     $0x18,%esp
mov     0x8149450,%eax
push    %eax
push    $0xa
push    $0x1
push    $0x80f409a
call    8074f80 <_IO_fwrite>
add     $0x10,%esp
mov     0x8149450,%eax
sub     $0xc,%esp
push    %eax
call    8074cb0 <_IO_fflush>
add     $0x10,%esp
sub     $0x8,%esp
```

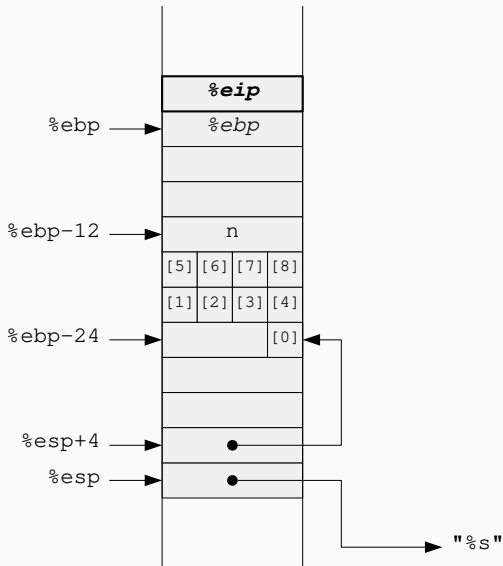
Aufbauen des Stack-Frames

```
lea     -0x15(%ebp),%eax
push    %eax
push    $0x80f40a5
call    806c900 <__isoc99_scanf>
add     $0x10,%esp
```

Lesen der Adresse von password

```
mov     %eax,-0xc(%ebp)
cmpl    $0xffffffff,-0xc(%ebp)
jne     8049d0d <authenticate+0x52>
mov     $0xffffffff,%eax
jmp     8049d1c <authenticate+0x61>
sub     $0xc,%esp
lea     -0x15(%ebp),%eax
push    %eax
call    8049bd5 <checkPassword>
add     $0x10,%esp
leave
ret
```

Schreiben von n



- 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 `0x08049d1e`
→ 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

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

```
// Lese width und height vom Benutzer  
int *matrix = malloc(width * height * sizeof(*matrix));  
// Befülle 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
- Gegenmaßnahme: Arithmetik mit Überlaufprüfung
 - Nichttrivial, deshalb als Compiler-Builtin
 - Siehe gcc.gnu.org und clang.llvm.org
 - Compilerflag `-fsanitize=undefined` sorgt dafür das **vorzeichenbehaftete** Überläufe erkannt werden



■ Format-String-Angriffe:

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

- Benutzer kann printf(3p) einen beliebigen Format-String unterjubeln
- Durch geschicktes Einfügen von %-Platzhaltern kann er beliebige Stack-Inhalte auslesen und u. U. beliebige Speicherinhalte überschreiben (z.B. %n)

■ Gegenmaßnahme: Ausgabe auch von einzelnen Strings mit Formatstring (oder puts(3p))

```
printf("%s", untrustedString);  
puts(untrustedString); // automatischer Zeilenumbruch
```



H.1 Stack-Aufbau eines Prozesses

H.2 Live-Hacking

H.3 Gegenmaßnahmen

H.4 Hacking

- **Allerwichtigste Schutzmaßnahme ist das Bauen robuster Software!**
- Eingaben Lesen **immer** mit begrenzter Länge
 - `scanf("%s", buffer);`
 - Stattdessen: `char buffer[10]; scanf("%9s", buffer);`
 - Platz für `'\n'`, `'\0'` etc. bedenken
- Nur mit Vorsicht zu genießen sind u. a. `strcpy(3p)`, `strcat(3)`, `sprintf(3p)` 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(3p)` benutzen
 - Alternativen `strncpy(3p)`, `strncat(3p)` haben keine sinnvolle Semantik
 - Beispiel: `strncpy(3p)` terminiert String nicht mit `'\0'`, falls Puffer zu klein :-)

- 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*)

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
-
- NX-Bit und -fPIE mittlerweile standardmäßig aktiviert in gcc

Hardware-Ebene: Shadow stacks

- Beim Aufruf einer Funktion wird die Return-Adresse auf zwei verschiedenen Stacks abgelegt
- Vor dem Zurückspringen wird überprüft, ob die beiden Adressen noch gleich sind
- Hat sich eine Adresse verändert, wird eine Ausnahme ausgelöst
- Aktivierung mit `-mshstk`

Hardware-Ebene: Indirect branch tracking

- Es darf nicht mehr an beliebige, sondern nur noch an bestimmte vordefinierte Stellen im Adressraum des Programms gesprungen werden
- Adressen werden mit bestimmter Instruktion markiert (`endbr32` oder `endbr64`)
- Aktivierung mit `-fcf-protection=full`



H.1 Stack-Aufbau eines Prozesses

H.2 Live-Hacking

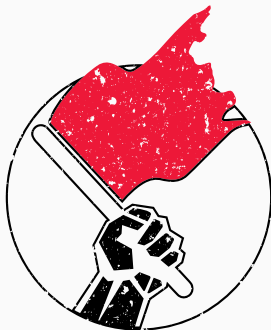
H.3 Gegenmaßnahmen

H.4 Hacking

- Shell-Server harsh (*Holey Assailable Remote Shell*):
 - Läuft auf Rechner `i4hacking.cs.fau.de`, Port `10443`
 - Verbindungen nur aus dem CIP-Netz
 - Verbinden z.B. mit netcat: `nc -q0 i4hacking 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 ab sofort bis Semesterende erreichbar





Mehr Hacking? FAUST!

- Details unter <https://faust.cs.fau.de/participate/>
- Frei zugängliche wöchentliche Treffen (siehe Webseite / Chat)