

Übungen zu Systemprogrammierung 2 (SP2)

ÜH – C und Sicherheit

C. Erhardt, J. Schedel, A. Ziegler, J. Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2015 – 26. bis 30. Oktober 2015

https://www4.cs.fau.de/Lehre/WS15/V_SP2

Agenda

- 5.1 Linux-Install-Party der FSI
- 5.2 Stack-Aufbau eines Prozesses
- 5.3 Live-Hacking
- 5.4 Gegenmaßnahmen



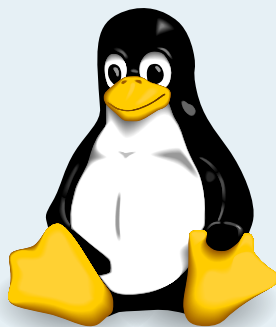
- 5.1 Linux-Install-Party der FSI
- 5.2 Stack-Aufbau eines Prozesses
- 5.3 Live-Hacking
- 5.4 Gegenmaßnahmen



Veranstaltungshinweis

Linux-Install-Party der FSI

- am Montag, den 02.11.2015, ab 14:00
- im 02.152-113 (Blaues Hochhaus, 2. Stock)
- weitere Informationen unter
<https://fsi.cs.fau.de/linuxinstall>



- 5.1 Linux-Install-Party der FSI
- 5.2 Stack-Aufbau eines Prozesses
- 5.3 Live-Hacking
- 5.4 Gegenmaßnahmen



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



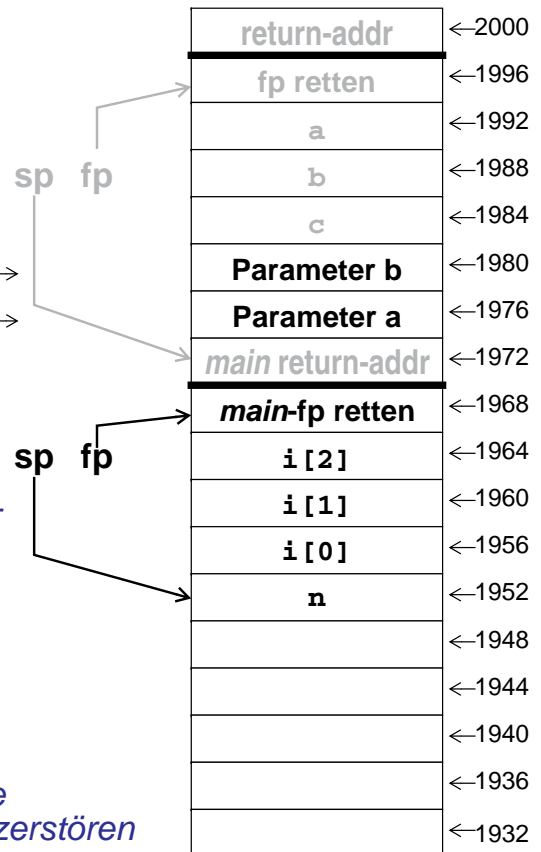
```
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}
```

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

*Stack-Frame für
f1 erstellen
und aktivieren*

$\&x = fp+8$
 $\&y = fp+12$
 $\&i[0] = fp-12$
 $\&n = fp-16$

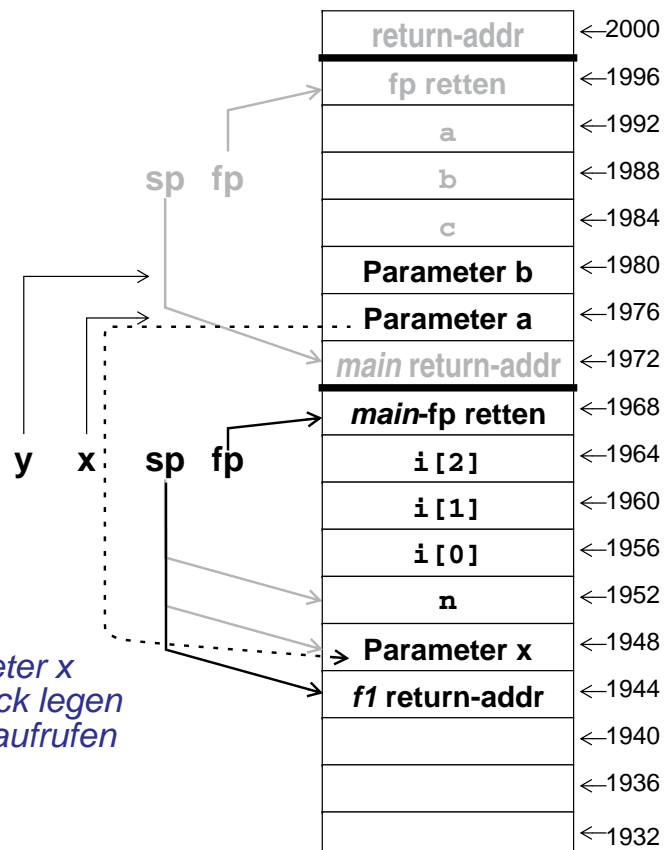
*i[4] = 20 würde
return-Adresse zerstören*



```
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);
}

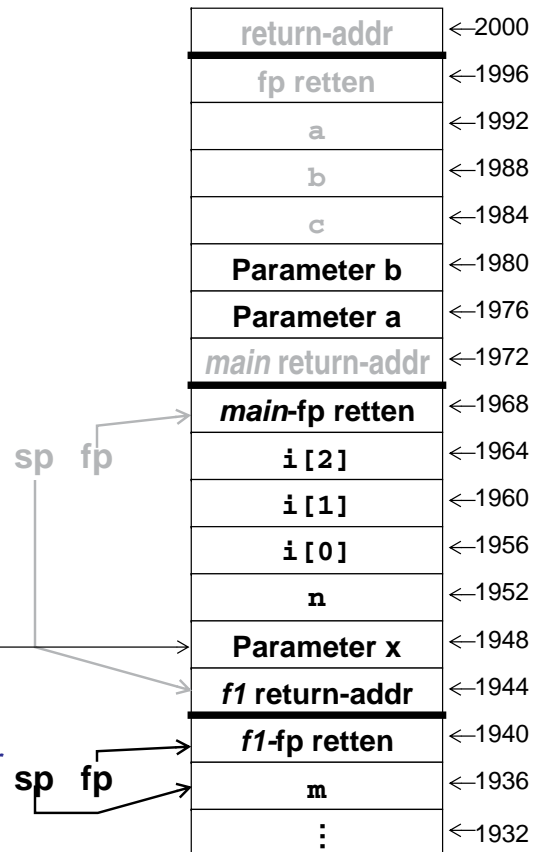
int f1(int x, int y) {
    int i[3];
    int n;

    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;

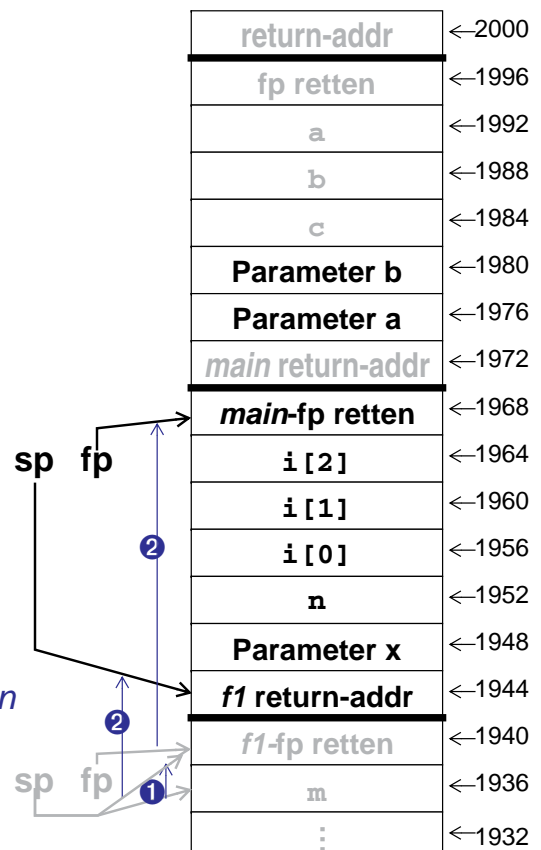
    x++;
    n = f2(x);
    return(n);
}

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

```

Stack-Frame von
f2 abräumen

- ① sp = fp
- ② fp = pop(sp)



```

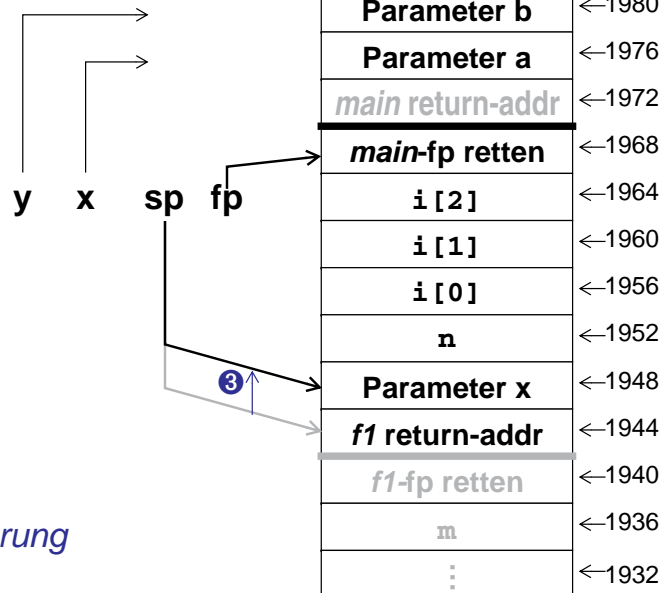
main() {
    int a, b, c;
    a = 10;
    b = 20;
    f1(a, b);
    return(a);
}

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



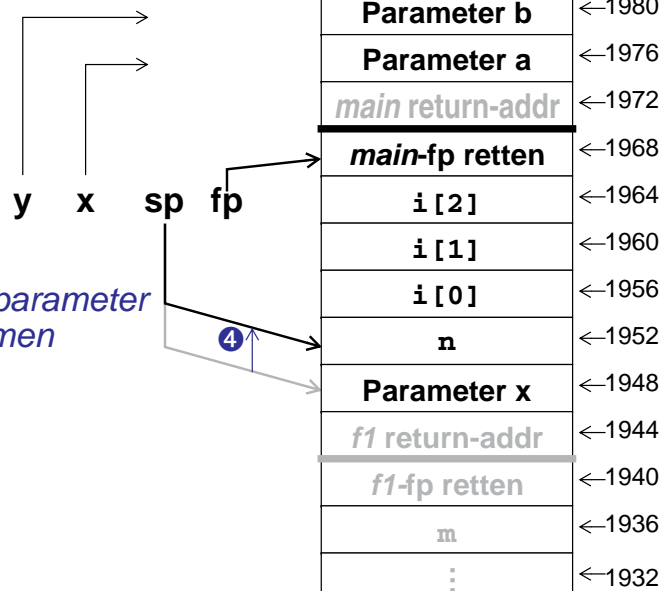
```

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

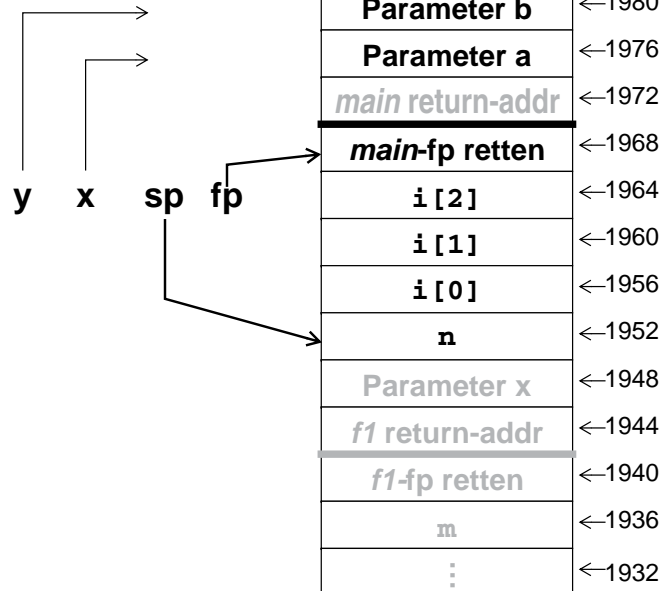


```

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);
}

```

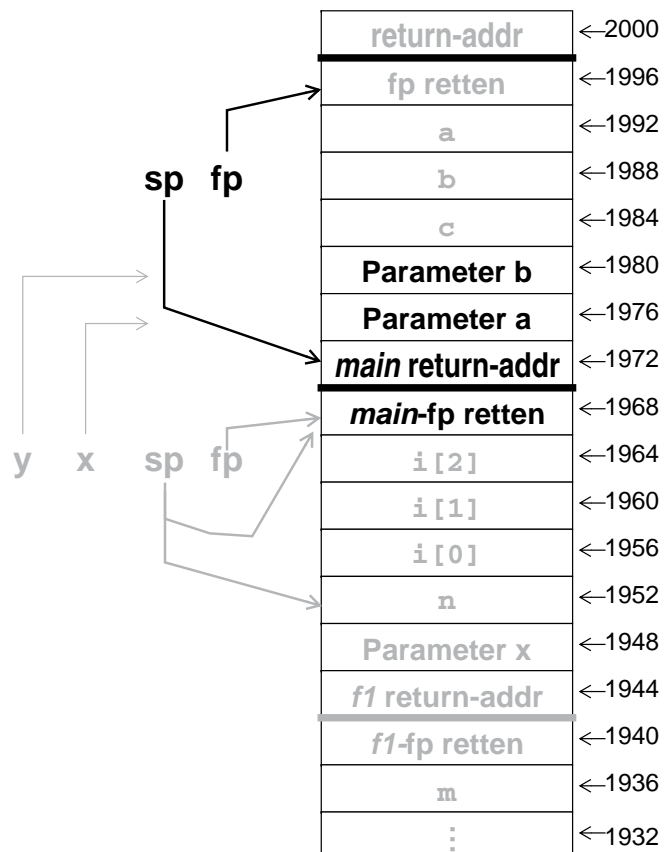


```

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);
}

```



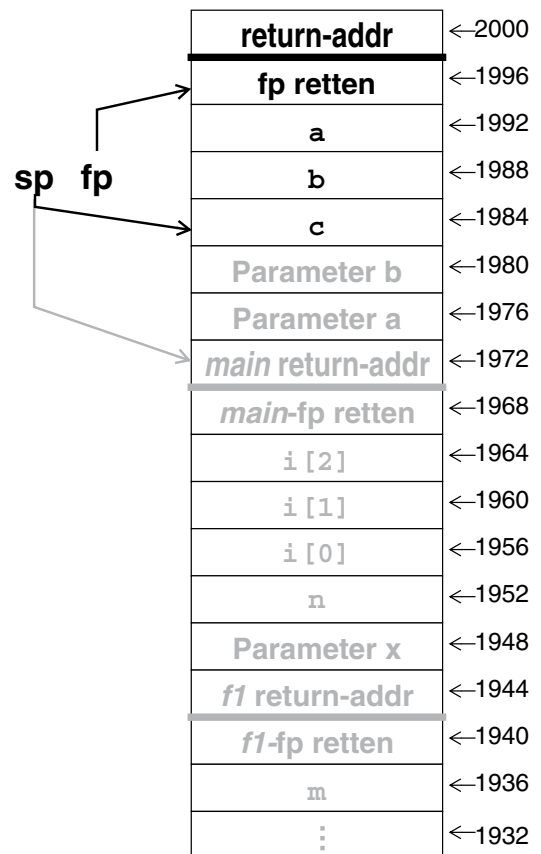

```

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);
}

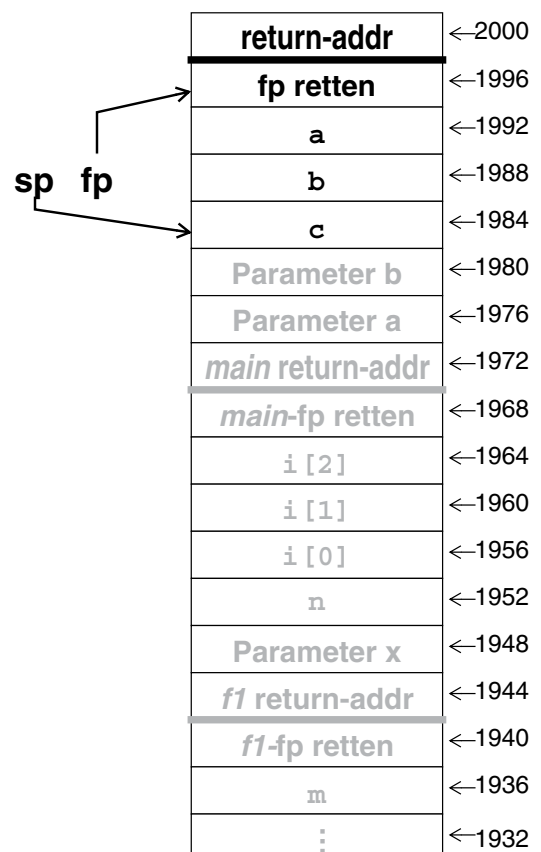
```



```

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

```



- 5.1 Linux-Install-Party der FSI
- 5.2 Stack-Aufbau eines Prozesses
- 5.3 Live-Hacking
- 5.4 Gegenmaßnahmen



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
08049bf8 d _DYNAMIC
08049cec d _GLOBAL_OFFSET_TABLE_
080489c4 R __IO_stdin_used
           w __ITM_deregisterTMCloneTable
           w __ITM_registerTMCloneTable
           w __Jv_RegisterClasses
08048be8 r __FRAME_END__
08049bf4 d __JCR_END__
08049bf4 d __JCR_LIST__
08049d3c D __TMC_END__
08049d3c B __bss_start
08049d34 D __data_start
08048700 t __do_global_dtors_aux
08049bf0 t __do_global_dtors_aux_fini_array_entry
08049d38 D __dso_handle
08049bec t __frame_dummy_init_array_entry
           w __gmon_start__
08049bf0 t __init_array_end
08049bec t __init_array_start
           U __isoc99_scanf@@GLIBC_2.7
08048990 T __libc_csu_fini
08048920 T __libc_csu_init
           U __libc_start_main@@GLIBC_2.0
08048680 T __x86.get_pc_thunk.bx
08049d3c D _edata
08049d48 B _end
08048994 T _fini
080489c0 R _fp_hw
0804852c T _init
08048650 T _start
08048831 t authenticate
0804874b t checkPassword
08049d44 b completed.6279
           U crypt@@GLIBC_2.0
08049d34 W data_start
08048690 t deregister_tm_clones
           U execl@@GLIBC_2.0
08048894 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
08048720 t frame_dummy
           U fwrite@@GLIBC_2.0
080488cb T main
           U perror@@GLIBC_2.0
           U puts@@GLIBC_2.0
080486c0 t register_tm_clones
08049d40 B stdout@@GLIBC_2.0
           U strcmp@@GLIBC_2.0
```

Analysieren des Stack-Layouts

```
$ objdump -d auth
08048831 <authenticate>:
08048831: 55
08048832: 89 e5
08048834: 83 ec 28
08048837: a1 40 9d 04 08
0804883c: 50
0804883d: 6a 0a
0804883f: 6a 01
08048841: 68 3a 8a 04 08
08048846: e8 65 fd ff ff
0804884b: 83 c4 10
0804884e: a1 40 9d 04 08
08048853: 83 ec 0c
08048856: 50
08048857: e8 14 fd ff ff
0804885c: 83 c4 10
0804885f: 83 ec 08
08048862: 8d 45 eb
08048865: 50
08048866: 68 45 8a 04 08
0804886b: e8 d0 fd ff ff
08048870: 83 c4 10
08048873: 89 45 f4
08048876: 83 7d f4 ff
0804887a: 75 07
0804887c: b8 ff ff ff ff
08048881: eb 0f
08048883: 83 ec 0c
08048886: 8d 45 eb
08048889: 50
0804888a: e8 bc fe ff ff
0804888f: 83 c4 10
```

```
push    %ebp
mov     %esp,%ebp
sub     $0x18,%esp
mov     0x8049d40,%eax
push    %eax
push    $0xa
push    $0x1
push    $0x8048a3a
call    80485b0 <fwrite@plt>
add     $0x10,%esp
mov     0x8049d40,%eax
sub     $0xc,%esp
push    %eax
call    8048570 <fflush@plt>
add     $0x10,%esp
sub     $0x8,%esp
```

Aufbauen des Stack-Frames

```
lea     -0x15(%ebp),%eax
push    %eax
```

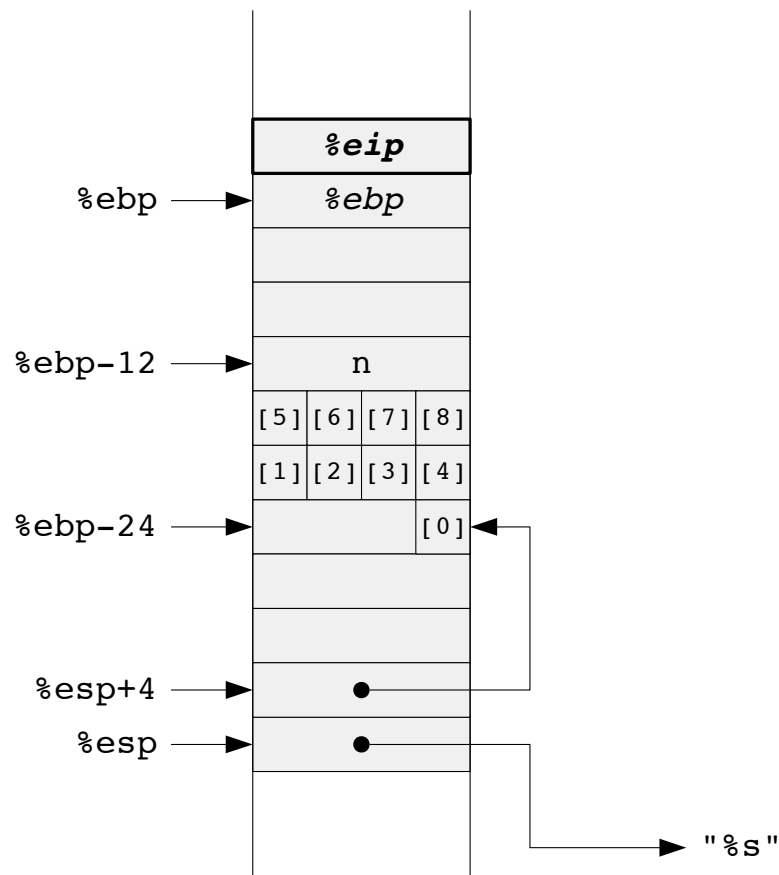
Lesen der Adresse von password

```
push    $0x8048a45
call    8048640 <__isoc99_scanf@plt>
add     $0x10,%esp
```

```
mov     %eax,-0xc(%ebp)
```

Schreiben von n

```
cmpl    $0xffffffff,-0xc(%ebp)
jne     8048883 <authenticate+0x52>
mov     $0xffffffff,%eax
jmp     8048892 <authenticate+0x61>
sub     $0xc,%esp
lea     -0x15(%ebp),%eax
push    %eax
call    804874b <checkPassword>
add     $0x10,%esp
```



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 `0x08048894`
→ 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

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



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 Linux-Install-Party der FSI
- 5.2 Stack-Aufbau eines Prozesses
- 5.3 Live-Hacking
- 5.4 Gegenmaßnahmen



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*