

SecureCrew eZine 1/2003

Inhalt

Coding:

- C99, der immer noch neue Standard von C.

ByteTrek::Bitfolge1

- Was passiert vom Einschalten des Computers bis zur Benutzung eines Programms?

SocketCoding::Grundlagen

- Der 1. Teil von seths Socketprogrammierung-Tutorials

Die SecureCrew, das Team von securecrew.net

- Die Members
- Die Board-Mods

Impressum:

Chefredakteur: Pascal Weibel alias Scipio
[scipio\(at\)securecrew.net](mailto:scipio(at)securecrew.net)

Mitarbeiter: Die SecureCrew:

SecureCrew-Team:

- Tec [\(at\)securecrew.net](mailto:(at)securecrew.net)
- boppy "
- Seth "
- STeFaN "

Moderatoren:

- PeaceTreaty "
- daikatana "

Kontakt: [ezine\(at\)securecrew.net](mailto:ezine(at)securecrew.net)
www.securecrew.net/?ezine

Bezugsquellen:

- Direktanforderung per E-Mail bei der Redaktion [ezine\(at\)securecrew.net](mailto:ezine(at)securecrew.net)
- Bei securecrew.net zum Download (www.securecrew.net/?ezine)

Copyright:

Copyright © 2002
by Pascal Weibel,
<http://www.securecrew.net> und dem
securecrew.net - Team,
(fortan SecureCrew genannt)

Die Inhalte dieses eZines sind urheberrechtlich geschützt.

Der Inhalt (Alle Grafiken, Codes und Texte) des SecureCrew eZines unterliegen dem Copyright der SecureCrew.

Ohne unsere ausdrückliche Genehmigung dürfen Inhalte dieses eZines weder verändert noch gewerblich genutzt werden. Das eZine ist gratis bei uns zu beziehen. Zuwiderhandlungen ziehen straf- oder zivilrechtliche Folgen nach sich. Alle Rechte bleiben vorbehalten.

SecureCrew behält sich das Recht vor, jederzeit und ohne vorherige Ankündigung Verbesserungen und Änderungen einzelner Inhalte vorzunehmen.

C99, der immer noch neue Standart der Programmiersprache C

How it developed

Entwickelt wurde die Programmiersprache C in den Bell Laboratories von AT&T im Jahre 1972 von Dennis M Ritchie, in den nächsten Jahren wurde sie von Brian W. Kernighan noch weiter verbessert. Einen Standart gab es noch nicht.

Als Bibel in Fragen um C galt die erste Ausgabe des Buches "The C Programming Language", geschrieben 1978 von diesen beiden Herren. Doch bereits Anfangs der 80er Jahre wurde die Notwendigkeit eines C Standarts erkannt.

Die Schaffung des ersten ANSI-C-Standarts

1983 Begann das ANSI-Komitee X3J11 (American National Standart Institute) mit der Arbeit an einer Standardisierung für C. Dies ergab den 1989 veröffentlichte erste Standart, C89 (seither als ANSI C bekannt), welcher 1995 nach der 1992 erfolgten Entwicklung von C++ durch Bjarne Stroustrup. Dieser verbesserte Standart beinhaltet einige nützliche Sprachelemente aus C++, welche damals in C nicht vorhanden waren.

C99, der internationale C-Standart

Das Ziel bei der Schaffung von ISO C99 war es, einen Internationalen C-Standart zu finden, der auch von den Verbesserungen der sich immer weiter entwickelnden Objekt-Orjentierten Tochter C++ profitierte.

Wie der Name schon sagt, wurde ISO C99 (ISO = International Standarts Organisation) im Jahre 1999 als neuer Standart für C geschaffen.

A new standart is born - use it!

Das grösste Problem bei der Verbreitung eines neuen Standarts ist die Verbreitung des alten. Selbst eine sich rasant bewegende Branche wie die Informatikbranche kann direkt konservativ wirken.

Die Vorteile, die eine Benutzung von C99 mit sich bringen sind eindeutig:

- Internationaler, Systemübergreifender Standart. C99 kompatible Programme können auf jedem Compiler, der diesen Standart unterstützt, übersetzt werden. Dies ist bei vielen C-Dialekten zb. von Borland-, Microsoft- und anderen Compilern nicht der Fall, die Dialekte können nicht transferiert werden, fremde Compiler kennen gewisse Funktionen nicht, es führt zu Fehlerausgaben, und wer jemals versucht hat, ein grösseres Programm aus einem Dialekt erfolgreich durch seinen Standartcompiler zu schicken weiss, wie umständlich und Zeitraubend das ist.
- Implementierung wichtiger Funktionen und Konstrukten aus C++

Die wichtigsten Änderungen

Die Unterschiede von ISO-C99 gegenüber ANSI-C89 sind in erster Linie übernommene Möglichkeiten und Vorschriften aus C++, welche das Programmieren vereinfachen oder dies zumindest sollten.

Einige (mehr oder weniger wichtige) Beispiele:

- In Funktionen ohne Argumente kann das Schlüsselwort void weggelassen werden *g*
- Rückgabetypen müssen explizit angegeben werden, kein implizites int mehr
- Logischer Datentyp `_bool`

Das einzig offizielle Schriftdokument dazu:

* ANSI/ISO 9899:1999 - Programming Language C (C99), für \$18 Online als PDF verfügbar (http://www.techstreet.com/cgi-bin/detail?product_id=232462).

ByteTrek::Eine Reise in die unendlichen Bitstroeme des Computers

ByteTrek::Bitfolge 1 Was passiert vom Einschalten des Computers bis zur Benutzung eines Programms?

(by seth & Scipio)

Nicht im Text erklarte Abkuerzungen und Fachbegriffe sind bei ihrer ersten Verwendung *kursiv* geschrieben, Erklaerungen siehe Ende des Artikels

1. Instanz - Das BIOS:

Nachdem wir die Stromzufuhr zum Computer sichergestellt haben ("Einschalte-Knopf") wird ein Speicherchip auf dem Mainboard, *BIOS-ROM* genannt, aktiv. Auf dem Chip ist ein Kleines Programm gespeichert, das nun geladen wird, Das BIOS. Es laedt die wichtigsten *Hardwaremodule*, stellt die erste Schnittstelle zw. User und Computer dar (Verbindung zum Mainboard und Bildschirm) und bietet damit dem User die Moeglichkeit, grobe Einstellungen vorzunehmen, wie z.B. Bootfolge etc.

BIOS steht fuer Basic Input Output System. In der Fachliteratur, besonders in aelteren Quellen auch BIOS ROM genannt, was heute nicht mehr korrekt waere, da es sich meist um EPROMs oder FEPRoMs handelt. Unmittelbar nach dem Start eines PCs wird das BIOS geladen um im wesentlichen zwei Aufgaben zu erfuellen:

1) Zunaechst wird die hardware getestet in wenn moeglich initialisiert. Nach einem solchen Test folgt ein weiterer Test: POST (Power On Self Test), der u.a. den Prozessor auf korrekte Funktionsweise hin prueft und checkt ob das BIOS seine Informationen anstaending geladen hat. Nach diesen und einigen anderen Tests aller moeglicher hardware sollte das BIOS ein ok geben.

2. Instanz - Der MBR:

Wenn POST abgeschlossen ist wird in dem ersten Segment (bei Pcs 0000:0000 bis 0000:FFFF) die interrupt-Vektortabelle initialisiert. Fuer jeden Interrupt findet sich dort dann eine Speicheradresse, die dann angesprungen wird sobald der Interrupt ausgeloeset wird.

Wenn Post sich beendet wird Interrupt 0x16 ausgeloeset, womit dann auch die Suche nach einem Betriebssystem beginnt.

Auf welchem Datentraeger das (zuerst) geschieht wird im BIOS festgelegt.

Gleichgueltig, um welchen Datentraeger es sich handelt wird auf dem Datentraeger zunaechst der erste Sektor gelesen (512 Byte). Der erste Sektor wird auch MBR (Master Boot Record) genannt, wobei der MBR folgende Struktur hat:

- Ein kleines Startprogramm ist in den ersten 446 Bytes untergebracht.

- 4 * 16 Byte enthalten Informationen ueber etvl. Partitionen auf der Festplatte. Bei Disketten faellen diese Informationen weg.

- Ab Offset 0x1FE befindet sich die sog. Magic number, die angibt ob es sich bei diesen Sektor um den MBR handelt. Die Magicnumber ist 2 Bytes lang und hat den Wert 0xAA55.

Im MBR nun steht die Information, wo sich der bootbare Kernel des Betriebssystems (oder der Systemdisk (auch bootdisk genannt) befindet. Der MBR verweist also auf den OS-Kernel oder, im Fall eines Multiboot-Systems, auf den Bootloader (z.b. Grub oder LILO), bei dem man dann das zu bootende OS auswaehlen kann.

Auch Sicherheits-Systemkernel, z.b. der *Failsafe Linux* Kernel laesst sich vom Bootloader aus booten, sozusagen das Aequivalent zum abgesicherten Modus bei Windows, wenn beim Betriebssystem Probleme auftreten (falsches Modul aktiviert o.Ae.).

3. Instanz - Betriebssystem

Nachdem also entweder der MBR oder der Bootloader den OS-Kernel gestartet hat, wird das OS geladen. Waehrend diesem Vorgang laedt das OS die ganzen Hardwaremodule und ermoeoglicht damit im Gegensatz zum BIOS die Benutzung jeglicher Hardware, also auch zu Peripheriegeraeten wie Maus, Tastatur (Hier wird z.b. die Keymap geladen, also sichergestellt, dass ein Schweizer nicht mit amerikanischer Tastaturbelegung arbeiten muss, es sei denn, er hat dies explizit eingestellt), Drucker, Scanner, Netzwerkkarte etc.

Ausserdem stellt das OS ein UI zur Verfuegung, also die Schnittstelle zwischen User und System, zu dem eine Datenverwaltung, ein einfacher Editor (in MS-DOS wird dieser mit edit filename aufgerufen, in UN*X ist dies in der Regel vi) und andere Werkzeuge gehoeren. Das OS bietet dem User zudem eine Basis zur Ausfuehrung von Programmen wie beispielsweise Editoren oder Compilern. Die Editoren und Werkzeuge sind jedoch streng gesehen schon ins OS eingebettete Programme.

Nachdem das OS nun geladen ist haben wir im allgemeinen einen Prompt vor uns (Eingabeaufforderung), entweder DOS oder eine UNIX-Shell, etwa TCSH (TC SHell) oder BASH (Bourne Again Shell). Dies nennt man Kommandozeile. Diese bietet eine Plattform, um Programme auszufuehren, was bedeutet, dass diese Programme z.b. die Tastatur auslesen oder Outputs auf den Bildschirm ausgeben koennen ohne eigene Module dafuer enthalten zu muessen.

4. Instanz - WM / GUI

Programme wiederum koennen jedoch auch ein GUI (Graphical User Interface, graphisches Frontend) fuer das Betriebssystem enthalten. Dies sind sog. *WM*.

Diese werden auf der Kommandozeile ausgefuehrt. Bei alten MS Windows Versionen (3.x) haben wir das mit dem Befehl win aufgerufen, bei spaeteren (ab 95) wurde es automatisch gestartet, da es im Script *autoexec.bat*.

Bei Unix starten wir erst den X server, der den Graphikkartentreiber zur Verfuegung stellt und waehlen einen WM (z.b. KDE, Gnome, Fvwm2...). Nun startet unser Window Manager auf und wir haben das gewohnte Betriebssystem vor uns, wo wir nun Fenster-Basierte Programme starten koennen wie beispielsweise einen Text-Editor wie MS Word unter Windows, KWord, Xemacs, Kate etc unter UN*X.

Benutzte Begriffe, # = wird im Text erklart:

Booten: (Auf)starten#
Bootable: bootfaehig. Ein bootfaehiger Datentraeger hat ein eigenes System.
BIOS: Basic-Input-Output-System#
ROM: Read-Only-Memory#
MBR: Master Boot Record#
Modul: vgl. Hardwaretreiber unter MS Win
OS: Betriebssystem, Beispiele: Microsoft (MS) Windows, DOS; MacOS; UN*X (vereint mehrere Betriebssysteme)#
UI: User Interface
GUI: Graphical UI#
Failsafe Linux: Ein Standart Linux Kernel, der auch dann funktioniert, wenn der User einen eigenen Kernel erstellt hat oder nichtfunktionierende Module geladen hat
WM: Window Manager
autoexec.bat: Ein Script, das beim DOS-Systemstart automatisch gestartet wird und alle Befehle und Programme, die darin aufgefuehrt sind, ebenfalls automatisch ausfuehrt.

Naechste Ausgabe von ByteTrek

ByteTrek::Bitfolge2
Wie kommt meine Eingabe im Programm von der Tastatur auf den Bildschirm?
Grundlagen des IO (In- Output, Ein- Ausgabe) Konzepts.

SocketProgrammierung:: Grundlagen

Socket Programmierung in C

Grundlagen

by seth, set(at)securecrew.net

Eine Einfuehrung in die TCP/IP sowie UDP Grundlagen erspare ich mir. Wer sich da noch informieren muss/moechte, der wird spaetestens bei www.google.de fuendig werden. Zu diesem Thema existieren eine Menge gute Texte.

Was vermittelt werden soll:

- Grundlegende Hintergrundinformationen.
- Ein Verstaendins dafuer wie man mit sockets programmiert.
- Wissen ueber die wichtigsten Grundlagen, ohne die ohnehin nichts geht.

..wobei hier lediglich ipv4 abgehandelt wird.

Soweit der Plan.....

Am besten lernt man, wenn man sich daran macht irgendetwas "sinvolles" produzieren zu wollen.

In diesem Fall basteln wir uns einen kleinen Server und einen passenden client.

Zu diesem Zweck baseln wir einen kleinen Server und einen passenden Cient dazu. Der Server wird etwas text an den client senden und sich dann beenden. Am besten macht ihr das im Netzwerk oder mit Freunden uebers Inet.

Im Anhang gibts die codes, die wir jetzt mal Schritt fuer Schritt zusammenstellen.

1.0 Sockets

1.1 Socket Adresstrukturen (ipv4)

2.0 Funktionen

2.1 socket()

2.2 bind()

2.3 connect()

2.4 listen()

2.5 accept()

2.6 recv()

2.7 send()

3.0 Praktische Beispiele

3.1 Server Grundstruktur

3.2 Client Grundstruktur

1.0 Sockets

1.1 Socket Adresstrukturen

Die meisten Funktionen die wir benutzen benoetigen Hinweise auf die aktuelle Socketstruktur.

Die wird durch Zeiger realisiert.

Fuer jedes unterstuetzte Protokoll existiert eine eigene Adresstruktur welche in `netinet/in.h` definiert sind.

Fuer unsere Zwecke ist die Socket Adresstruktur von IPv4 interessant.

Alle Adresstrukturen beginnen mit dem Praefix "sockaddr_".

```
struct in_addr
{
    in_addr_t s_addr; /* 32 Bit Netzwerkadresse (IP
Adresse) */
};
```

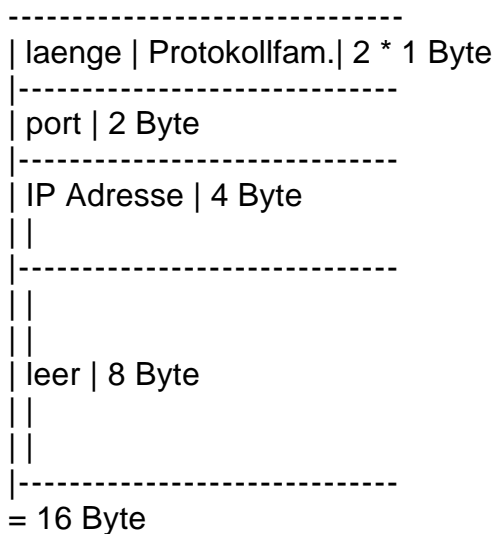
```
struct sockaddr_in
{
    uint8_t sin_len; /* Laenge der Gesamtstruktur */
    sa_family_t sin_family; /* Protokollfamilie */
    in_port_t sin_port; /* TCP- oder UDP Portnummer */
};
```

```
struct in_addr sin_addr; /* 32 Bit
Netzwerkadresse (IP Adresse) */
```

```
char sin_zero[8]; /* ein "lueckenfueller", macht
weiter gar nichts */
};
```

Den Wert sin_zero[] benoetigt man um die Struktur immer auf konstanter Groesse von 16 Byte zu halten.

Grafisch kann man sich diese Struktur so vorstellen:



(Das nur mal um hier etwas "Designarbeit" zu leisten..denn die groesse spielt fuer uns kaum eine Rolle)

Wir wir mit diesen Strukturen arbeiten wird spaeter gezeigt.

2.0 Funktionen

Um was halbwegs brauchbares entwickeln zu koennen sollte man Kenntniss ueber die noetigsten Funktionen haben:

2.1 socket() (Client/Server)

Die Funktion socket() fordert einen socket vom Betriebssystem an.

Der Prototyp der Funktion sieht so aus:

```
int socket(int domain, int type, int protocol)
-----
domain - die zu nutzende Protokollfamilie
type - Typ ( fuer uns ist hier lediglich der Wert
SOCK_STREAM interessant. Es existieren aber
noch andere. Eine Beschreibung gibts mit "man
socket")
protocol - dieser Parameter gibt an welches
Protokoll aufgrund der ersten beider Parameter
genutzt
werden soll. Wir haben die Moeglichkeit 0
anzugeben; der Systemkern trifft dann die
(verlaesslich) passende Wahl fuer unser
vorhaben. Was aber nicht unbedingt fuer
exotische
Vorhaben gelten muss.
```

domain kann folgende Werte annehmen

- AF_UNIX - Unix Domain Sockets
- AF_INET - TCP/IP
- AF_AX25 - Ax.25 (Amateurradio) :)
- AF_IPX - Novell IPX
- AF_APPLETALK - Appletalk
- AF_NETROM - NetROM (Amateurradio)

Fuer uns wichtig ist AF_INET.

2.2 bind() (server)

Wenn man einen socket angefordert hat, muss er serverseitig "von Hand" nutzbar gemacht werden.

Dazu ordnet bind() (bei Internetprotokollen) dem Socket eine lokale Protokolladresse zu, was dann unter "binden" zu verstehen ist. Von aussen haben wir dann einen offenen port, der u.U. auf anfragen wartet (Bei dem client koennen wir uns das sparen; wenn wir connect() aufrufen waehlt der Kernel fuer uns einen sog kurzlebigen port aus).

```
int bind(int sockfd, struct sockaddr *adr, int
adrlength)
```

sockfd - der Socket Filedeskriptor der mit socket() angefordert wird
struct sockaddr *adr - Ein Zeiger auf eine protokollspezifische Adresse
adrlen - enthaelt die Groesse unserer Adresse

2.3 connect() (client)

Der Client nutzt connect() um sich mit einem Server zu verbinden.

```
int connect(int sockfd, const struct sockaddr
*serveraddr, socklen_t addrlen)
```

sockfd - der socket filedeskriptor
*serveraddr - Zeiger auf die Socket Adressstruktur des servers (IP Adresse und Port)
addrlen - die Groesse der Adressstruktur

Wenn es sich um einen TCP Socket handelt initiiert connect() selbststaendig den 3-way handshake.

2.4 listen() (server)

Listen() macht nur fuer die Serversoftware Sinn. Wird mit socket() ein Socket angelegt begegnet uns der Kernel zunaechst mit dem Vorurteil das es sich um einen aktiven Socket handeln wird. Das bedeutet dasz er irgendwann an connect() mit uebergeben wird. Da wir, wenn wir einen Server schreiben wollen solche(!) aktiven Sockets nicht brauchen, biegen wir ihn so um, dasz er den Status listen erhaelt.

```
int listen(int sockfd, int backlog)
```

sockfd - Socket Filedeskriptor
backlog - gibt die maximale Anzahl der Verbindungen an. Diese mehranfragen stellt der Kernel in eine Queue, die der Reihe nach abgearbeitet werden.

Tatsechlich existieren jedoch zwei Warteschlangen:

- die Erste enthaelt unvollstaendige Verbindungsaanforderungen,
- die Zweite vollstaendig hergestellt Verbindungen.

2.5 accept() (server)

Auch accept() wird nur vom Server verwendet. Wird accept() angerufen so wird die naechste Verbindung in der Warteschlange fuer vollstaendige Verbindungen zurueckgegeben. Wenn die Warteschlange leer ist schlaeft der Prozess.

```
int accept(int sockfd, struct sockaddr *clientadr,
socklen_t *adrlen)
```

sockfd - Socket Filedeskriptor
*clientadr - Ist ein Zeiger auf die Socket Adressstruktur des Clients
*adrlen - gibt die Groesse von *clientadr an.

wenn accept() korrekt ausgefuehrt wurde erhalten wir einen neuen Deskriptor der auf die dann bestehende TCP Verbindung mit dem Client zeigt.

2.6 recv()

mit recv() liest man "messages" von einem verbundenen socket aus. Praktisch wuerde man meist mit der Funktion recvfrom() arbeiten, fuer unsere Grundlagenforschung jedoch erstmal nicht noetig. Tatasechlich unterscheiden sich diese Funktion unwesentlich voneinander
recvfrom() wird fuer nicht verbindungsorientierte Protokolle wie UDP eingesetzt und benoetigen als zusaetzliche Argumente u.a. noch einen Zeiger auf eine Adressstruktur des Senders.
recv() wird benutzt wenn man weisz, dasz man von einem verbundenen(!) Socket liest.

```
int recv(int sockfd, char *buffer[], int buflen, int
flag)
```

sockfd - der Socket Filedeskiptor
buffer - ein char Array wo die empfangene
Message abgelegt wird.
buflen - Groesse von buff
flag - fuer uns immer 0

Durch das flag Argument kann man neben der 0
noch einige andere Argumente auch mit einer
OR
Verknuepfung angeben, die das Verhalten der
FUnktion aendern. So kann man z.B. das flag
MSG_DONTROUTE
angeben um dem Kernel mitzuteilen das sich der
Verkehr in einem lokalen Netz abspielt und
deswegen
keine Suche in Routingtabellen noetig ist (was
bedeutet dasz man ressourcen spart).
Fuer uns (noch) nicht von Belang, also flag = 0
weitere infos gibts mit man recv :)

2.7 send()

send() eignet sich hervorragend zum versenden
von daten :)
send() und recv() werden i.d.R. zusammen
benutz. D.h. was mittels send() gesendet wird
wird mit
recv() empfangen.

```
int send(int sockfd, const void *buff, int bytes, int  
flag)
```

sockfd - Socket Filedeskiptor
buff - ein array mit der zu versendenden Message
buflen - Groesse von buff
flag - wie bei recv()

3.0 Praktische Beispiele

3.1 Server Grundstruktur

Weiter oben haben wir die allernoetigsten

Funktionen kennengelernt um einen kleinen
Daten-
transfer zu realisieren. Um dorthin zu gelangen
schreiben wir uns Schritt fuer Schritt ein
paar kleine Beispiele zusammen.

Prinzipiell kann ein Server ja allerlei Dienste
anbieten, sei es ftp, telnet oder was es sonst
noch so gibt. Um jetzt endlich mal produktiv zu
werden (oder so was aehnliches) fangen wir
mal an
die Grundstruktur eines Servers zu schreiben.
Dieses "Grundprinzip" gilt fuer eine Menge
anderer
Programme auch, nur werden diese meist ;) noch
"etwas" komplexer. Hier soll uns eine Demo
reichen.

Zunaechst schauen wir uns mal den code fuer
einen server an.
Was der machen soll duerfte ja klar sein.

```
-----  
-----  
#include <stdio.h>  
#include <netdb.h>  
#include <netinet/in.h>  
#include <unistd.h>  
#include <sys/socket.h>  
#include <sys/types.h>  
  
#define MAXCON 1024  
  
int main(int argc, char* argv[])  
{  
  int i = 1;  
  int sock, c, b;  
  int a_deskript;  
  int client_size;  
  char array[128];  
  
  struct sockaddr_in server, client; /*unsere  
  Strukturen*/
```

```
/* testen ob die richtige Anzahl an Argumenten  
uebergeben wurde */  
if (argc != 2)  
{  
  printf("\nSYNTAX: server <port-nummer> \n");  
  exit(1);  
}
```

```

/* der String , den wir versenden wollen */
printf("\nText eingeben (max 128 Zeichen): ");
scanf("%s",&array);

server.sin_addr.s_addr = INADDR_ANY;
/* in argv[1] steht die Portnummer */
server.sin_port = htons((unsigned short int) atol
(argv[1]));
server.sin_family = AF_INET;

/* socket anfordern*/
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock == -1)
{
perror("\nSOCKET FAILED");
exit(1);
}
else
{
printf("\nSOCKET done");
}

/* an den port "binden" */
if (b = bind(sock, &server, sizeof(server)) == -1)
{
perror("\nBIND FAILED");

exit(1);
}
else
{
printf("\nBIND done");
}
/* auf status listen setzen */
if (listen(sock,MAXCON)== -1)
{
perror("\nLISTEN FAILED");
close(sock);
exit(1);
}
else
{
printf("\nLISTEN done");
}

/* hier startet eine endlosschleife, die solange
laeuft bis sie auf einen accept()
reagieren kann */
for ( ; ; )
{
client_size = sizeof(client);
a_deskript = (accept(sock, &client,
&client_size));

```

```

if (a_deskript == -1)
{
perror("\nACCEPT FAILED");
close(sock);
exit(1);
}
else
{
printf("\nACCEPT done");
}
/* Ausgabe IP Adress vom client */
printf("\nclient von %s\n",
inet_ntoa(client.sin_addr));
/* senden des strings */
send(a_deskript, array, strlen(array),0);
close(sock);
exit(0);
}

return(0);
}

```


In die Abfragen ob eine Funktion erfolgreich zurueckkehrt ist packen wir jeweils einen close(socket) um sicherzugehen dasz der socket wieder freigegeben wird.

Als Argument wird der Port mit angegeben.

3.1 Client Grundstruktur

Passend zu diesem server noch der client code, der erheblich kuerzer ist.

```

#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>

```

```
#define BUFFER_SIZE 8024

int main(int argc, char* argv[])
{

char buffer[BUFFER_SIZE];
char tempo[8024];
int dub;
int bytes;
int sock, a, b;
struct sockaddr_in server;
dub=0;

if (argc != 3)
{
perror("\nSYNTAX: client <host-ip> <port> \n");
exit(1);
}

/* IP des Servers */
server.sin_addr.s_addr = inet_addr(argv[1]);
/* Port zu dem ein connect gemacht werden soll
*/
server.sin_port = htons((unsigned short
int)atoi(argv[2]));
server.sin_family = AF_INET;

sock = socket(AF_INET, SOCK_STREAM,0);

if(sock == -1)
{
perror("\n//SOCKET FAILED");
exit(0);
}
/* das connecten */
if (connect(sock, &server, sizeof(server)) == -1)
{
perror("\nCONNECT FAILED");
exit(0);
}
/* in bytes packen wir den string der durch
recv()() gelesen wird */
bytes = recv(sock, buffer, sizeof(buffer),0);
b = 0;
a = 0;

if (bytes == -1)
{
perror("\nRECV FAILED");
exit(1);
}

printf("\n%s\n",buffer);
```

```
return(0);
}
```


Diese Beispiele sind sehr schlicht gehalten und machen, wie gesagt, nicht viel. Bevor ihr jetzt fragt wie es weitergeht analysiert die codes und beantwortet eure Fragen mit hilfe der manpages. Es gehoert sich rueckgabewerte zu geben, auch wenn wir diese nicht auswerten (was aber letztendlich jedem selber ueberlassen bleibt). Desweiteren ist natuerlich hier null Sicherheit zu finden.

Als kleinen Anreiz soll versucht werden eine komplette Textdatei zu oeffnen und zu versenden. Der Client soll diese empfangen und unter dem urspruenglichen Namen auch wieder speichern.

viel spass

:wq!

Die SecureCrew:

Tec:

Real Name: Siegfried Bolz
Date of Birth: 24/04/1976
Spezialitäten: Security, Linux, Programmieren
Funktion: Admin, Leader
Sound: Trance, Techno, 80ties
Herkunft: Kempten
Beruf: Informatik Student
Motto: Code jetzt, stirb später!

seth:

Real name: Herbert Maia
Date of Birth: --/--/-- (nicht angegeben)
Spezialitäten: Fragen stellen, C
Funktion: Member, Mod, Kritiker
Herkunft: Hannover
Beruf: Programmierer
Motto: Weltfrieden

Scipio:

Real Name: Pascal Weibel
Date of Birth: 08/02/1981
Spezialitäten: Linux, C/C++ Coding,
Electronics
Funktion: Member, Mod,
eZine-Chefredakteur
Beruf: Informationstechnologie und
Elektrotechnik - Studi
Sound: Progressive- / Hardtrance
Herkunft: Eastern Switzerland
Motto: Live to Rave!

boppy:

Real Name: Henning K. Bopp
Date of Birth: 02/12/1984
Spezialitäten: Windows
Funktion: Technischer Admin, Moderator
und Trottel für alles
Sound: Rock, NewMetal, Punk
Herkunft: W-E-City, NRW, Deutschland
Motto: I don't want to change the world,
and I don't want the world to
change me!

STeFaN:

Real Name: Jörg Lehmann
Date of Birth: 02/05/1978
Spezialitäten: Sicherheit allgemein, Bugs,
E-Technik
Funktion: Member, Mod, M\$-DAU ;-)
Beruf: EEBE, BKF
Sound: Trance, House, Charts
Herkunft: Berlin
Motto: Geniesse jeden Tag als wäre es
dein letzter!

Die SC-Board Moderatoren

PeaceTreaty:

Real Name: Wolfgang Hotwagner
Date of Birth: keine Angaben
Spezialitäten: Programmierung
Funktion: Board-Moderator
Sound: Reggae, Rock
Herkunft: Oesterreich
Beruf: Elektriker
Motto: keine Angaben

daikatana:

Real name: keine Angaben
Date of Birth: 16/05/1962
Spezialitaeten: Programmierung, UN*X
Funktion: Board-Moderator
Herkunft: Berlin
Beruf: keine Angaben
Motto: keine Angaben