

RAW Socket Programmierung und Einsatzfelder

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von
André Volk (201210553)

Betreuer: Prof. Dr. Ch. Steigner & Dipl. Inform. H. Dickel
Institut für Informatik / Arbeitsgruppe Steigner

Koblenz, im Mai 2008

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ja Nein

Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

.....
(Ort, Datum)

.....
(Unterschrift)

Zusammenfassung

Für die Netzwerkprogrammierung hat sich auf breiter Front das Socket API nach Vorbild der Berkley Sockets durchgesetzt. Die „normalen“ Sockets in Form von Stream- oder Datagram-Sockets erleichtern zwar die Programmierarbeit, verschleiern jedoch auch zahlreiche Details der Netzwerkkommunikation vor dem Programmierer. So ist man beispielsweise auf die Nutzung der Protokolle TCP oder UDP eingeschränkt und agiert zwangsläufig bereits auf dem Application-Layer des TCP/IP Referenzmodells.

Für den Zugriff auf tiefer gelegene Netzwerkschichten, d.h. für den Zugriff auf die Headerdaten eines Netzwerkpaketes, hält das Socket API die sogenannten RAW Sockets bereit. Mit ihnen ist es möglich, alle IP Pakete inklusive Headerdaten zu lesen oder von Grund auf neu zu generieren. Hiermit ist es nun auch möglich, Protokolle zu verwenden, die dem Anwendungsprogrammierer bislang nicht zugänglich waren (z.B. ICMP oder OSPF) oder sogar eigene IP basierte Protokolle zu entwickeln.

RAW Sockets stoßen an ihre Grenzen, wenn es darum geht auf den Data-Link-Layer der Netzwerkkommunikation zuzugreifen. Unter Linux gibt es hierfür einen weiteren Socket-Typ: Den PACKET Socket.

Die Studienarbeit möchte einen Einstieg in die Programmierung mit den eher unbekannteren RAW und PACKET Sockets schaffen. Dabei werden einige Beispielprogramme vorgestellt und mögliche Anwendungsgebiete aufgezeigt.

Inhaltsverzeichnis

| | |
|---|-----------|
| 1 Grundlagen | 1 |
| 1.1 Arbeitsweise von Sockets | 1 |
| 1.2 Unterhalb des Application-Layers | 3 |
| 1.3 RAW Sockets | 3 |
| 1.4 RAW Sockets unter bestimmten Betriebssystemen | 4 |
| 1.5 Zugriff auf den Data-Link-Layer | 4 |
| 1.6 Einsatzgebiete von RAW Sockets | 5 |
| 1.6.1 Analyse des Netzwerkverkehrs | 5 |
| 1.6.2 Nutzung von Protokollen unterhalb der Transport- schicht | 5 |
| 1.6.3 Störung und Manipulation des Netzwerkverkehrs . . | 6 |
| 1.7 Besonderheiten beim Einsatz von RAW Sockets | 6 |
| 1.7.1 root-Rechte | 6 |
| 1.7.2 Ports | 7 |
| 1.7.3 bind() und connect() | 7 |
| 1.7.4 Client-Server Semantik | 7 |
| 1.7.5 Automatische und halbautomatische Paketgenerierung | 8 |
| 1.7.6 Adressierung von RAW Sockets | 8 |
| 1.7.7 Promiscuous Mode | 9 |
| 2 Programmierung mit RAW Sockets | 10 |
| 2.1 Aufbau eines RAW Sockets | 10 |
| 2.2 Lesen von einem RAW Socket | 10 |
| 2.3 Beispiel: show_packet_binary.c | 11 |
| 2.4 Type-Casting | 14 |
| 2.5 Position der Header im Speicher | 16 |
| 2.6 Byte-Order | 17 |
| 2.7 Ankunftszeit eines Netzwerkpaketes | 18 |
| 2.8 Schreiben auf einen RAW Socket | 19 |
| 2.9 Adressierung von RAW Sockets | 22 |
| 2.10 Die Internet Prüfsumme | 23 |
| 2.11 Schreibzugriff auf den IP Header mit IP_HDRINCL | 25 |
| 2.12 Beispiel: syn_packet_gen.c | 27 |
| 2.13 Der TCP/UDP Pseudo-Header | 29 |
| 3 Programmierung mit PACKET Sockets | 31 |
| 3.1 Aufbau eines PACKET Sockets | 31 |
| 3.2 Lesen von einem PACKET Socket | 32 |
| 3.3 Beispiel: show_packet_verbose.c | 33 |
| 3.4 Schreiben auf einen PACKET Socket | 35 |
| 3.5 Beispiel: arp_packet_gen.c | 36 |
| 3.6 Adressierung von PACKET Sockets | 37 |

| | | |
|----------|---|-----------|
| 3.7 | Erweitertes Beispiel: icmp_packet_gen.c | 39 |
| 4 | Über die Arbeit mit RAW Sockets | 40 |
| A | Kurzübersich über Protokolldaten | 42 |
| A.1 | Das Ethernet Protokoll | 43 |
| A.1.1 | Bit-Schema | 44 |
| A.1.2 | Datenstruktur | 44 |
| A.1.3 | Weitere Informationen / Quellen | 44 |
| A.2 | Das ARP Protokoll | 45 |
| A.2.1 | Bit-Schema | 45 |
| A.2.2 | Datenstruktur | 46 |
| A.2.3 | Weitere Informationen / Quellen | 46 |
| A.3 | Internet Protokoll (IP) Header | 47 |
| A.3.1 | Bit-Schema | 47 |
| A.3.2 | Datenstruktur | 49 |
| A.3.3 | Weitere Informationen / Quellen | 49 |
| A.4 | ICMP Protokoll Header | 50 |
| A.4.1 | Bit-Schema | 50 |
| A.4.2 | Datenstruktur | 51 |
| A.4.3 | Weitere Informationen / Quellen | 51 |
| A.5 | UDP Protokoll Header | 52 |
| A.5.1 | Bit-Schema | 52 |
| A.5.2 | Datenstruktur | 53 |
| A.5.3 | Weitere Informationen / Quellen | 53 |
| A.6 | TCP Protokoll Header | 54 |
| A.6.1 | Bit-Schema | 54 |
| A.6.2 | Datenstruktur | 56 |
| A.6.3 | Weitere Informationen / Quellen | 56 |
| B | Programmbeispiel show_packet_binary.c | 57 |
| C | Programmbeispiel syn_packet_gen.c | 61 |
| D | Programmbeispiel show_packet_verbose.c | 66 |
| E | Programmbeispiel arp_packet_gen.c | 76 |
| F | Programmbeispiel icmp_packet_gen.c | 80 |

Listings

| | | |
|----|--|----|
| 1 | Ausschnitt aus show_packet_binary.c | 12 |
| 2 | Datenstruktur: struct iphdr aus <netinet/ip.h> | 14 |
| 3 | Beispielprogramm: typecasting_example.c | 15 |
| 4 | Beispielprogramm: timestamp_example.c | 18 |
| 5 | Beispielprogramm: packet_gen_raw_easy.c | 20 |
| 6 | Datenstrukturen: struct sockaddr_in und struct in_addr aus <netinet/in.h> | 23 |
| 7 | Funktion comp_chksum zur Berechnung der Internet Prüf- summe | 24 |
| 8 | Ausschnitt aus syn_packet_gen.c: Variablendeklaration . . . | 27 |
| 9 | Ausschnitt aus syn_packet_gen.c: Type Casting | 27 |
| 10 | Ausschnitt aus syn_packet_gen.c: IP Header füllen | 28 |
| 11 | Ausschnitt aus syn_packet_gen.c: TCP Header füllen | 28 |
| 12 | Datenstruktur für den TCP/UDP Pseudoheader: struct pseu- dohdr | 30 |
| 13 | Ausschnitt aus syn_packet_gen.c: struct data_4_checksum . | 30 |
| 14 | Ausschnitt aus show_packet_verbose.c: Type Casting des Data- Link-Layer Headers | 34 |
| 15 | Ausschnitt aus show_packet_verbose.c: printEthernetInfor- mation() | 34 |
| 16 | Datenstruktur: struct sockaddr_ll aus <linux/if_packet.h> . | 37 |
| 17 | Beispielprogramm: get_if_number_example.c | 38 |
| 18 | struct ether_header aus <net/ethernet.h> | 44 |
| 19 | struct arphdr aus <net/if_arp.h> (modifiziert für IPv4 und Ethernet) | 46 |
| 20 | struct iphdr aus <netinet/ip.h> | 49 |
| 21 | struct icmphdr aus <netinet/ip_icmp.h> | 51 |
| 22 | struct udphdr aus <netinet/udp.h> | 53 |
| 23 | struct tcphdr aus <netinet/tcp.h> | 56 |
| 24 | Beispielprogramm: show_packet_binary.c | 57 |
| 25 | Beispielprogramm: syn_packet_gen.c | 61 |
| 26 | Beispielprogramm: show_packet_verbose.c | 66 |
| 27 | Beispielprogramm: arp_packet_gen.c | 76 |
| 28 | Beispielprogramm: icmp_packet_gen.c | 80 |

1 Grundlagen

1.1 Arbeitsweise von Sockets

Aufgrund seiner hohen Verbreitung über alle gängigen Betriebssysteme wird für die Netzwerkkommunikation einer Anwendung meist das *Socket Application Programming Interface* (Socket API) verwendet. Es basiert auf dem Berkley Socket API, das für die Programmiersprache C entwickelt wurde (vgl. [3]).

Mit dem Socket API verläuft die Netzwerkkommunikation nach dem gleichen Prinzip wie die Ein- und Ausgabe auf Dateien: Ein Benutzerprogramm fordert einen *Socket* über eine festgelegte Routine beim Betriebssystem an (`socket()`) und erhält einen *Socket-Deskriptor* zurück. Genau wie bei einem *File-Deskriptor* handelt es sich dabei um einen einfachen Integer. Ein Betriebssystem kann durchaus mehrere Sockets zur gleichen Zeit verwalten.

Über einen Socket-Deskriptor kann, ähnlich einer Datei, schreibend (z.B. `per send()` oder `write()`) oder lesend (z.B. `per receive()` oder `read()`) auf einen Socket zugegriffen werden.

Die Daten, die von einem Benutzerprogramm in einen Socket geschrieben wurden, werden vom Betriebssystem mit Headerdaten versehen und über das Netzwerk versandt. Daten, die vom Host über das Netzwerk empfangen wurden und für ein bestimmtes Benutzerprogramm vorgesehen sind (erkennbar z.B. an dem verwendeten Protokoll und der Port-Nummer), werden vom Betriebssystem wiederum ohne Headerdaten im zugehörigen Socket abgelegt. Von dort können sie vom Benutzerprogramm ausgelesen werden. Folglich operiert ein Benutzerprogramm stets nur auf den sogenannten *Nutzdaten* der Netzwerkkommunikation.

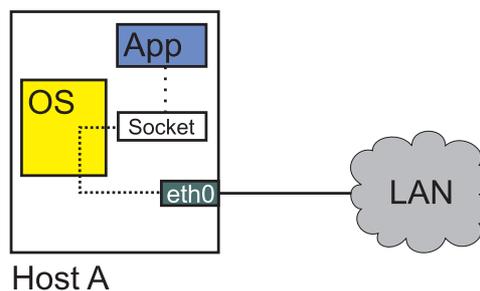


Abbildung 1: Prinzip der Socket-Kommunikation

In Abbildung 1 wird das Prinzip der Kommunikation über Sockets noch einmal schematisch dargestellt.

| |
|--|
| 5. Application-Layer |
| DHCP , DNS , FTP , HTTP , IMAP4 , IRC , NNTP , XMPP , MIME , POP3 , SIP , SMTP , SNMP , SSH , TELNET , BGP , RPC , RTP , RTCP , TLS/SSL , SDP , SOAP , L2TP , PPTP , ... |
| 4. Transport-Layer |
| TCP , UDP , DCCP , SCTP , GTP , ... |
| 3. Network-Layer |
| IP (IPv4 , IPv6) , ICMP , IGMP , RSVP , IPsec , OSPF , ... |
| 2. Data-Link-Layer |
| ATM , DTM , Ethernet , FDDI , Frame Relay , GPRS , PPP , ARP , RARP , ... |
| 1. Physical-Layer |
| Ethernet physical layer , ISDN , Modems , PLC , SONET/SDH , G.709 , Wi-Fi |

Tabelle 1: Das TCP/IP Referenzmodell

Die Routinen und Datentypen des Socket APIs zielen auf eine schnelle und unkomplizierte Bereitstellung und Verwendung eines *Kommunikations-Endpunktes* (Sockets) ab. Dabei sollen möglichst viele protokollspezifische Details völlig versteckt oder zumindest vereinfacht werden:

- Durch die Zuweisung einer *Socket-Adresse*, bestehend aus IP Adresse und Port-Nummer, an einen lokalen Socket wird das Betriebssystem angewiesen, stets diese Adresse als Absender in ausgehenden Netzwerkpaketen zu verwenden. Die Zuweisung erfolgt über die Funktion `bind()`.
- Über die Festlegung des *Socket-Typs* kann gewählt werden, ob z.B. über einen Bytestrom- oder über einzelne Datagramme kommuniziert werden soll. Dazu sind die Socket-Typen `SOCK_STREAM` oder `SOCK_DGRAM` definiert. Implizit legt der Socket-Typ auch das verwendete Transport-Layer Protokoll fest: TCP im Falle des Stream-Sockets und UDP im Falle des Datagramm-Sockets. Die darunter stattfindende Umwandlung in einzelne IP Pakete wird vor dem Programmierer verborgen.
- Ein Verbindungsaufbau über TCP oder UDP Sockets geschieht stets über die Funktionen `listen()` oder `connect()`. Interne Details, wie z.B. der 3-Wege-Handshake von TCP, laufen dabei völlig unbenutzt ab.

Welche Bereiche der Netzwerkkommunikation durch die Verwendung „gewöhnlicher“ TCP oder UDP Sockets verborgen bleiben, lässt sich durch die Betrachtung des TCP/IP Referenzmodells (Tabelle 1) erschließen. Eine Anwendung, die über einen TCP- oder UDP-Socket kommuniziert, bewegt sich bereits auf Layer 5 (Application-Layer).

Alle tiefer gelegenen Schichten (inklusive der Transportschicht selbst) sind für die Applikation unzugänglich und bleiben dem Betriebssystem oder der Hardware vorbehalten.

1.2 Unterhalb des Application-Layers

Das TCP/IP Schichtenmodell spiegelt sich in jedem einzelnen Paket dadurch wider, dass jedes einzelne Protokoll, das verwendet wird, einen eigenen Headerbereich am Anfang jedes einzelnen Pakets beansprucht. Jedes Netzwerkpaket beginnt also mit mehreren Schichten von Headerdaten. Abbildung 2 zeigt ein typisches TCP Paket, das über ein Ethernet gesendet wird.

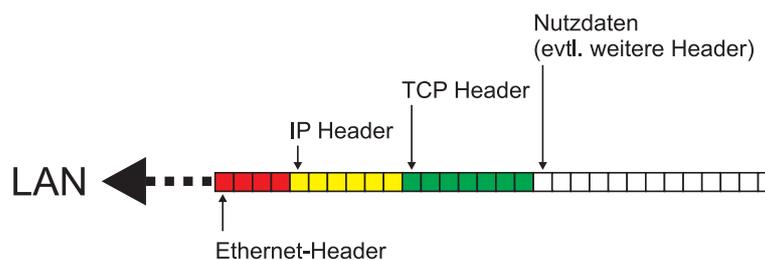


Abbildung 2: TCP Paket mit Headerdaten

Das Paket in Abbildung 2 beginnt mit dem Ethernet Header, der dem Data-Link-Layer zugeordnet ist. Es folgt der IP Header auf dem Network-Layer, der TCP Header auf dem Transport-Layer und schließlich die Nutzdaten auf dem Application-Layer.

Für ein Programm, das mit normalen TCP oder UDP Sockets arbeitet, sind nur noch die Nutzdaten eines Pakets sichtbar. Die *Headerdaten* der Protokolle unterhalb des Application-Layers sind unzugänglich.

1.3 RAW Sockets

Für den Zugriff auf tiefer gelegene Abstraktionsschichten, den *Transport-* und den *Network-Layer*, hält das Socket API einen speziellen Socket-Typ bereit: Den RAW Socket (`SOCK_RAW`). Mit ihm lassen sich Pakete inklusive aller Headerdaten bis zum Network-Layer Header auslesen und auch generieren. Typischerweise handelt es sich dabei um Pakete des Internetprotokolls (IP Pakete).

Der *Data-Link-Layer* ist jedoch auch mit (normalen) RAW Sockets noch nicht zugänglich.

1.4 RAW Sockets unter bestimmten Betriebssystemen

RAW Socket Programmierung ist in vielen Punkten höchst betriebssystemspezifisch. Die im Verlauf dieser Arbeit vorgestellten Beispiele wurden alle unter Linux und in der Programmiersprache C entwickelt.

Über das Windows-eigene Winsock API lassen sich RAW Sockets seit WindowsXP ServicePack 2 nur noch sehr eingeschränkt verwenden. TCP Pakete können hier gar nicht mehr und UDP Pakete nur noch mit einer gültigen Absenderadresse über RAW Sockets versendet werden [4].

Die Literatur beschreibt die Programmierung von RAW Sockets fast ausschließlich für UNIX-artige Betriebssysteme. Dabei ist anzumerken, dass die RAW Socket Programmierung auch unter verschiedenen Unix Derivaten nicht immer einheitlich ist.

1.5 Zugriff auf den Data-Link-Layer

RAW Sockets stoßen beim Zugriff auf den Data-Link-Layer an ihre Grenzen. Da der Zugriff auf den Data-Link-Layer softwareseitig jedoch durchaus möglich und für manche Anwendungen sogar notwendig ist, werden weitere Techniken neben den RAW Sockets benötigt. Diese Techniken sind allerdings noch stärker abhängig vom darunterliegenden Betriebssystem.

Unter Linux gibt es für den Zugriff auf den Data-Link-Layer die sogenannten *PACKET Sockets*. PACKET Sockets gehören nicht zu den Standard Berkley Sockets sind aber Teil des Linux-eigenen Socket APIs. Dadurch bleibt (zumindest unter Linux) das Prinzip der Socket Programmierung beim Zugriff auf den Data-Link-Layer erhalten.

PACKET Sockets werden unter Linux als spezielle Form von RAW Sockets behandelt. Insofern gelten viele der Aussagen, die im Folgenden über RAW Sockets gemacht werden, auch für PACKET Sockets. Bei der Implementierung gibt es jedoch Unterschiede zwischen den beiden Socket-Typen, die in späteren Kapiteln noch genauer beleuchtet werden.

Weitere Techniken für den Zugriff auf den Data-Link-Layer auf anderen Betriebssystemen sind z.B. der *BSD Paket Filter* (BPF) unter BSD oder das *Data Link Provider Interface* (DLPI) unter System V.

Um die Vielzahl an Möglichkeiten zu kapseln und unabhängig vom Betriebssystem programmieren zu können, wird stets die Verwendung von universellen *Packet Capturing Bibliotheken* (wie z.B. pcap¹ oder libnet²) empfohlen.

1.6 Einsatzgebiete von RAW Sockets

Für die meisten Anwendungen ist die Verwendung „gewöhnlicher“ Sockets auf TCP- oder UDP-Basis völlig ausreichend und auch empfehlenswert, da der Protokollstack des Betriebssystems im Zweifel immer ausgereifter und stabiler ist, als eine Implementierung mit RAW Sockets. Dennoch gibt es Anwendungsfelder, bei denen man mit „gewöhnlichen“ Sockets nicht zum Ziel kommen kann. Im Folgenden werden einige Einsatzgebiete von RAW Sockets kurz genannt.

1.6.1 Analyse des Netzwerkverkehrs

Mit RAW Sockets lassen sich prinzipiell alle Pakete auslesen, die an einem lokalen Netzwerkinterface empfangen wurden. Normale RAW Sockets unterliegen allerdings unter Linux zahlreichen Beschränkungen. So muss man sich beim Aufbau eines RAW Sockets auf ein bestimmtes, IP-basiertes Protokoll (z.B. ICMP, TCP, UDP usw.) festlegen. Anschließend werden nur eingehende Pakete an den Socket weitergereicht, die zu diesem Protokoll gehören. Portnummern spielen jedoch keine Rolle mehr.

In anderen Unix-Derivaten oder gar unter Windows sind die Beschränkungen zum Teil noch strenger.

Für die Implementierung sogenannter *Network Sniffer* (Analysetools zum Mitschneiden von Netzwerkverkehr) eignen sich die Linux PACKET Sockets besser als RAW Sockets, da diese den gesamten Datenverkehr ohne Beschränkungen auslesen können.

1.6.2 Nutzung von Protokollen unterhalb der Transportschicht

Als praxisrelevantes Beispiel für den Einsatz von RAW Sockets ist die Verwendung des ICMP Protokolls hervorzuheben. Die Testnachrichten (ECHO) von ICMP kommen zwar in der Praxis sehr häufig zum Einsatz, lassen sich jedoch weder mit TCP noch mit UDP basierten Sockets empfangen oder versenden. Mit RAW Sockets ist hingegen beides möglich.

Gleiches gilt natürlich für alle weiteren Protokolle, die nur auf dem Internetprotokoll, nicht jedoch auf TCP oder UDP basieren, beispielsweise

¹<http://www.tcpcap.org/>

²<http://www.packetfactory.net/projects/libnet/>

OSPF oder IGMP, andere seltenere Protokolle oder auch völlig neue, eigene Protokolle. Auf der Website der *Internet Assigned Numbers Authority (IANA)*³ findet sich eine Liste aller offizieller Protokolle, die direkt auf dem Internetprotokoll aufsetzen.

Für die Arbeit mit Data-Link-Layer Protokollen müssen PACKET Sockets verwendet werden. Hier ist beispielsweise das ARP Protokoll als besonders praxisrelevant hervorzuheben.

1.6.3 Störung und Manipulation des Netzwerkverkehrs

Im Allgemeinen liegt die Robustheit und Funktionsfähigkeit eines Rechnernetzes im Interesse jedes einzelnen Teilnehmers. Um dies zu gewährleisten sollte sich jeder Teilnehmer natürlich um ein möglichst korrektes Kommunikations-Verhalten bemühen, was sich z.B. in der fehlerfreien Generierung von Paketen oder der sinnvollen Befolgung von Protokollabläufen zeigt.

Es ist davon auszugehen, dass aus diesem Grund bei allen gängigen Betriebssystemen auf eine besonders stabile Implementierung des Netzwerkprotokoll-Stacks geachtet wurde.

Ein Benutzerprogramm, das mit normalen Sockets arbeitet, hat foglich kaum Möglichkeiten, den allgemeinen Netzwerkverkehr zu stören oder zu manipulieren, da hier die essentiellen Teile der Netzwerk-Kommunikation noch dem Betriebssystem vorbehalten bleiben. Erst die Verwendung von RAW Sockets macht ein gezielt nonkonformes Verhalten im Netzwerk möglich, was die RAW Socket Programmierung vor allem unter *Hackern* beliebt gemacht hat. Zahlreiche Angriffstechniken, z.B. viele *Denial-of-Service* oder *Man-in-the-Middle* Attacken, wären ohne eine vollständig manuelle Paketgenerierung nicht möglich.

1.7 Besonderheiten beim Einsatz von RAW Sockets

Im Vergleich zu dem Einsatz von gewöhnlichen Sockets gibt es bei der Verwendung von RAW Sockets ein paar Besonderheiten, die beachtet werden müssen.

1.7.1 root-Rechte

Korrekte Headerinformationen sind essentiell für die Integrität und das Funktionieren der Netzwerkkommunikation. Da die Verantwortung, korrekte Paketheader zu erzeugen durch die Verwendung von RAW Sockets

³<http://www.iana.org/assignments/protocol-numbers>

vom Benutzerprozess übernommen werden kann, muss zumindest sichergestellt werden, dass diese Aufgabe nur der höchsten Autorität zukommen kann. In der Unix-Welt ist das der Benutzer „root“.

Anwendungen, in denen ein RAW Socket erstellt wird, dürfen also nur mit root-Rechten ausgeführt werden.

1.7.2 Ports

Ports sind ein Konzept der Transport-Layer Protokolle (z.B. TCP oder UDP). RAW Sockets arbeiten jedoch bereits auf dem Network-Layer wodurch Ports hier natürlich noch keine Bedeutung haben.

RAW Sockets können folglich nicht auf einem bestimmten Port lauschen. Für gewöhnlich nimmt ein RAW Socket einfach allen Datenverkehr eines bestimmten IP-basierten Protokolls an. Eine eventuell vorhandene Portnummer in einem Paket müsste bei Bedarf von Hand aus dem TCP oder UDP Header extrahiert werden. Die dazu nötigen Techniken werden im späteren Verlauf dieser Arbeit noch ausführlich erklärt.

1.7.3 bind() und connect()

Die typischen Methoden des Socket APIs `bind()` und `connect()` sind zusammen mit RAW Sockets zwar noch nutzbar, jedoch nicht mehr unbedingt nötig.

Mit `bind()` lässt sich beispielsweise eine Absenderadresse definieren, die der Kernel automatisch im IP Header der zu sendenden Pakete eintragen kann. Beim Empfang von Paketen werden nur Pakete angenommen, die an die mit `bind()` festgelegte Adresse gerichtet sind.

Mit `connect()` wird eine Zieladresse für einen RAW Socket festgelegt. Eine per `connect()` definierte Adresse wird als Empfänger im IP Header jedes ausgehenden Pakets eingetragen und es werden beim Empfang nur die Pakete angenommen, die von dieser Adresse stammen.

Im Zusammenhang mit PACKET Sockets hat die Funktion `connect()` keine Bedeutung mehr. `bind()` hingegen ist sinnvoll, um den Socket an ein bestimmtes Netzwerkinterface zu binden.

1.7.4 Client-Server Semantik

Die Abschnitte `sec:ports` und `sec:bindconnect` lassen schon vermuten, dass die von normalen Sockets gewohnte Client-Server Semantik mit RAW Sockets keine Anwendung findet. Die typischen Server-Funktionen `listen()` und `accept()` haben im Zusammenhang mit RAW Sockets keine Bedeutung mehr und die typische Client-Funktion `connect()` ist nur noch optional einsetzbar und entfällt für PACKET Sockets sogar vollständig.

Mit RAW Sockets werden nur noch einzelne, meist unsortierte und unzusammenhängende Pakete versendet und empfangen. Die Auswertung und Interpretation der Daten sowie ein eventuell gewünschtes Verhalten als Client oder Server obliegt allein der Anwendung.

1.7.5 Automatische und halbautomatische Paketgenerierung

Trotz der Verwendung von RAW Sockets übernimmt standardmäßig das Betriebssystem die Generierung des IP Headers. Erst die Socket-Option `IP_HDRINCL`, die über die Methode `setsockopt()` gesetzt werden muss, erlaubt die manuelle Konstruktion des IP Headers durch eine Anwendung. Doch selbst bei aktivierter `IP_HDRINCL` Option setzt das Betriebssystem die IP Prüfsumme noch immer selbst ein. Die IP Identifikationsnummer oder die Absenderadresse werden automatisch vom Kernel in den IP Header eingesetzt, wenn diese zuvor mit dem Wert Null gefüllt wurden. Die Paketgenerierung mit RAW Sockets und `IP_HDRINCL` Option erfolgt somit halbautomatisch. Die Programmierarbeit kann dadurch ein wenig erleichtert werden.

Mit `PACKET_SOCKETS` müssen Pakete immer vollständig von Hand generiert werden.

1.7.6 Adressierung von RAW Sockets

Die *Socketadresse* eines „gewöhnlichen“ Sockets setzt sich zusammen aus IP Adresse, Port und dem verwendeten (Transport-)Protokoll.

Diese Adressierung ist auf RAW Sockets nicht mehr anwendbar, da es sich bei einem RAW Socket nicht mehr zwangsläufig um einen *Kommunikationsendpunkt* handeln muss.

Art und Umfang einer Socketadresse kann je nach Einsatzzweck variieren. Für einen RAW Socket, der ankommenden Datenverkehr ausschließlich liest (Sniffer), ist gar keine Adressierung notwendig. Ein RAW Socket, der Pakete versendet, deren IP Header jedoch komplett automatisch generiert werden ist die Definition einer Empfänger-Socketadresse notwendig. Die Angabe einer Portnummer kann dabei jedoch durchaus entfallen, wenn ein Protokoll verwendet wird, das keine Ports unterstützt (z.B. ICMP).

Für alle anderen RAW oder PACKET Sockets ist die Angabe einer Socketadresse im Prinzip nur noch nötig, um das Netzwerkinterface festzulegen, über das gesendet werden soll.

Socketadressen werden von den Funktionen `bind()`, `connect()`, `sendto()` oder `recvfrom()` als Spezialisierung von `struct sockaddr` verlangt. RAW Sockets verwenden dabei `struct sockaddr_in`, PACKET Sockets verwenden `struct sockaddr_ll`.

1.7.7 Promiscuous Mode

Der Promiscuous Mode bezeichnet einen bestimmten Empfangsmodus der Netzwerk-Hardware, bei dem der gesamte Netzwerk-Datenverkehr, der an einem Interface eingeht, ausgelesen wird. Im Normalfall (ohne Promiscuous Mode) werden nur diejenigen Pakete betrachtet, die auch wirklich an die eigene Netzwerkschnittstelle adressiert wurden.

RAW Sockets kennen den Promiscuous Mode nicht. Mit ihnen lässt sich nur der Datenverkehr auslesen, der an die eigene Netzwerkschnittstelle adressiert wurde.

PACKET Sockets nehmen prinzipiell den gesamten Netzwerkverkehr an allen Schnittstellen des Hosts an. Durch eine Bindung des Sockets mittels `bind()` ist allenfalls die Begrenzung auf ein bestimmtes Netzwerk Interface möglich.

Sollte das explizite Aktivieren des Promiscuous Mode in einem bestimmten Anwendungsfall nötig sein, so ist dies über einen `ioctl()` System-Call möglich (siehe dazu „man 7 netdevice“).

2 Programmierung mit RAW Sockets

2.1 Aufbau eines RAW Sockets

Bei der Programmierung mit RAW Sockets wird nicht zwischen einem Server- und einem Client-Socket unterschieden. Sobald ein RAW Socket aufgebaut wurde kann auf diesen lesend und schreibend zugegriffen werden. Zum Aufbau eines Sockets wird wie gewohnt die `socket()` Funktion aus `<netinet/in.h>` benötigt:

```
int socket(int family, int type, int protocol);
```

Die `socket()` Funktion gibt im Erfolgsfall einen nicht negativen Socket Deskriptor zurück. Im Fehlerfall liefert `socket()` den Wert -1. Ein typischer Aufruf von `socket()` zum Erstellen eines RAW Sockets sieht wie folgt aus:

```
s = socket (AF_INET, SOCK_RAW, IPPROTO_TCP);
```

- Das Argument *family* erwartet eine Konstante, die Aufschluss über die verwendete Adressfamilie gibt. `AF_INET` legt fest, dass mit IPv4 Adressen gearbeitet wird.
- Der Parameter *type* beschreibt die Art des verwendeten Sockets. Im Falle von RAW Sockets benötigen wir die Konstante `SOCK_RAW`.
- An dritter Stelle wird `socket()` die Konstante *protocol* übergeben. Diese legt das Protokoll der Pakete, die über den Socket gelesen oder geschrieben werden können fest. Im Zusammenhang mit `AF_INET` sind hier nur IP basierte Protokolle möglich. Die gängigsten Konstanten sind: `IPPROTO_TCP`, `IPPROTO_UDP` und `IPPROTO_ICMP`. `IPPROTO_*` Konstanten finden sich in der C Headerdatei `<netinet/in.h>`, die entsprechenden Protokollnummern sind auf der Website der IANA⁴ aufgelistet.

2.2 Lesen von einem RAW Socket

Mit dem Befehl `read()` können, ähnlich dem Lesen aus einer Datei, die am Socket eingehenden Daten ausgelesen werden:

```
int read(int fd, char *Buff, int NumBytes);
```

⁴<http://www.iana.org/assignments/protocol-numbers>

Der Rückgabewert von `read()` gibt die Anzahl der gelesenen Bytes wieder. Der Parameter `fd` (= file descriptor) bezeichnet normalerweise die Datei aus der gelesen werden soll. Im Falle der Socket-Programmierung wird hier der Socket Deskriptor eingesetzt. `*Buff` zeigt auf den Lesepuffer (z.B. ein Array vom Typ `char`), in den die gelesenen Daten gespeichert werden. Über `NumBytes` muss festgelegt werden wieviele Bytes maximal gelesen werden sollen. Ein typischer Aufruf von `read()` zum Lesen von einem RAW Socket sieht wie folgt aus:

```
bytes = read(s, packet, sizeof(packet));
```

Alternativ können zum Lesen von einem RAW Socket auch die Funktionen `recv()` oder `recvfrom()` benutzt werden.

Anders als bei gewöhnlichen TCP oder UDP Sockets enthält der Lesepuffer nach der Ankunft von Netzwerkpaketen nicht mehr nur die *Payload* (Nutzlast) der Datenpakete, sondern auch die Headerdaten. Wird der Puffer seriell ausgelesen, so erhält man zuerst die Daten des IP Headers, dann die Headerdaten des Transportprotokolls (meist TCP oder UDP) und danach erst die Daten, die auch ein gewöhnlicher Socket „sehen“ würde.

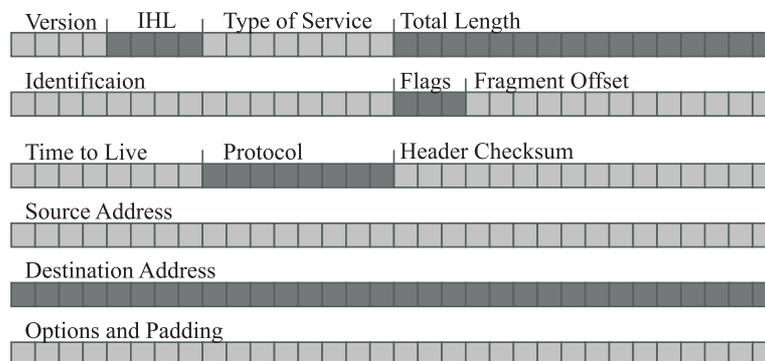


Abbildung 3: Der IPv4 Header

Headerdaten werden oft als 32 Bit breite Datenblocks dargestellt, in denen man die systematische Belegung durch diverse Datenfelder erkennen kann (siehe Abbildung 3). Diese schematischen Darstellungen ermöglichen die Interpretation der „rohen“, binären Daten in Netzwerkpaketen.

2.3 Beispiel: `show_packet_binary.c`

Das Beispielprogramm `show_packet_binary.c` aus Anhang B veranschaulicht, welchen Umfang die Daten, die aus einem RAW Socket „her-

auskommen“, besitzen und wie diese Daten tatsächlich beschaffen sind. Dazu liest das Programm das nächste ICMP Paket, das den Host erreicht, und gibt es in binärer Form komplett auf dem Bildschirm aus.

Das Codefragment in Listing 1 skizziert die Arbeitsweise des Programms: Die Daten des Lesepuffers werden Byte für Byte durchlaufen, wobei jedes Byte als 8-stelliger Binärwert ausgegeben wird.

```
1  int s, i;
2  char packet[ETHERMTU]; // ETHERMTU = 1500
3
4  /**
5   * Hier wird der SOCK_RAW Socket für ICMP Pakete angelegt.
6   * Um weitere Protokolle auszulesen müssen die Parameter
7   * entsprechend angepasst werden.
8   */
9
10 s = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
11
12 /**
13  * Mit read werden nun Daten vom Socket gelesen.
14  * Ein einmaliger Aufruf von read liefert genau ein
15  * Paket zurück.
16  *
17  * bytes          : Zahl der gelesenen Bytes
18  * s              : Socket, von dem gelesen werden soll
19  * packet         : Speicherplatz in den die gelesenen
20  *                 Daten geschrieben werden sollen
21  * sizeof(packet) : Länge des Speicherbereichs, in den
22  *                 geschrieben werden darf
23  * 0              : Keine besonderen Flags
24  */
25
26 bytes = read(s, packet, sizeof(packet));
27
28
29 // Durchlaufe die gelesenen Daten Byte für Byte
30
31 for (i = 0; i < bytes; i++){
32
33     // Gibt das Byte in 8-stelliger Binaerdarstellung aus
34     print_binary_byte (packet[i]);
35
36 }
```

Listing 1: Ausschnitt aus show_packet_binary.c

Nach dem Kompilieren und Starten des Programms `show_packet_binary` lässt sich die folgende Bildschirmausgabe provozieren, indem man parallel auf der Konsole z.B. den Befehl `ping localhost` absetzt:

```
01000101 00000000 00000000 01010100
00000000 00000000 01000000 00000000
01000000 00000001 00111100 10100111
01111111 00000000 00000000 00000001
01111111 00000000 00000000 00000001
00001000 00000000 00111110 10010011
11011100 00010001 00000000 00000001
11100110 01111011 00010111 01000111
11110000 10010011 00000100 00000000
00001000 00001001 00001010 00001011
00001100 00001101 00001110 00001111
00010000 00010001 00010010 00010011
00010100 00010101 00010110 00010111
00011000 00011001 00011010 00011011
00011100 00011101 00011110 00011111
00100000 00100001 00100010 00100011
00100100 00100101 00100110 00100111
00101000 00101001 00101010 00101011
00101100 00101101 00101110 00101111
00110000 00110001 00110010 00110011
00110100 00110101 00110110 00110111
```

Abbildung 4: Ausgabe von `show_packet_binary.c`: Ein ICMP Echo Request Paket

Mit dem Befehl `ping localhost` versendet der Host ein ICMP Echo Request Paket und erwartet ein ICMP Echo Reply Paket als Antwort. Obwohl einem RAW Socket normalerweise alle ausgehenden Pakete vorenthalten bleiben, kann in diesem Fall bereits das Echo Request Paket gelesen werden. Der Host hatte das Paket zuvor an sich selbst (`localhost`) geschickt und es somit nicht nur versendet sondern auf dem Loopbackinterface auch wieder empfangen. Abbildung 4 zeigt somit ein ICMP Echo Request Paket.

Zur Interpretation der binären Daten kann die schematische Darstellung eines IP Headers (Abbildung 3) ähnlich einer Schablone benutzt werden: In den ersten vier Bits lässt sich beispielsweise die binäre Vier erkennen, die auf die IP Version (IPv4) hinweist. Die vierte Zeile enthält die IP-Adresse des Absenders, Zeile 5 die IP-Adresse des Empfängers. Beide Adressen sind im obigen Beispiel identisch. Die Gruppierung in Blocks von jeweils 8 Bit Breite erleichtert die Umwandlung von der binären Darstellung in die Dotted-Decimal Darstellung und so lässt sich sowohl in Zeile

4 als auch in Zeile 5 die IP-Adresse 127.0.0.1, die Loopback-Adresse, erkennen. Der Befehl `ping` hat den Namen „localhost“ erwartungsgemäß in eine IP Adresse aufgelöst und ein IP Paket generiert, in dem die eigene Loopback-Adresse sowohl im Empfänger- als auch im Absenderfeld eingetragen wurde.

Ab Zeile 6 könnten laut Schema noch diverse IP-spezifische Optionen folgen. Das `ping` Programm hat jedoch keine weiteren Optionen gesetzt und so beginnt in der sechsten Zeile bereits der Header des ICMP Protokolls. Eine schematische Darstellung des ICMP Headers, die für die weitere Interpretation herangezogen werden kann, findet sich in Anhang A.4.1.

2.4 Type-Casting

Das Prinzip, eine „Schablone“ über binäre Daten zu legen, lässt sich in der Programmiersprache C mittels *Type-Casting* umsetzen. Hierbei wird einem Speicherbereich ein bestimmter Datentyp zugeordnet. Wenn es sich bei diesem Datentyp um einen zusammengesetzten Datentyp (**struct**) handelt, kann man anschließend gezielt bestimmte Bereiche des Speicherbereichs (z.B. das 3. Byte) über die Felder des Datentyps ansprechen.

```
1 struct iphdr
2 {
3     #if __BYTE_ORDER == __LITTLE_ENDIAN
4         unsigned int ihl:4;
5         unsigned int version:4;
6     #elif __BYTE_ORDER == __BIG_ENDIAN
7         unsigned int version:4;
8         unsigned int ihl:4;
9     #else
10    # error "Please fix <bits/endian.h>"
11    #endif
12    u_int8_t tos;
13    u_int16_t tot_len;
14    u_int16_t id;
15    u_int16_t frag_off;
16    u_int8_t ttl;
17    u_int8_t protocol;
18    u_int16_t check;
19    u_int32_t saddr;
20    u_int32_t daddr;
21    /*The options start here. */
22 }
```

Listing 2: Datenstruktur: struct iphdr aus <netinet/ip.h>

Für die Arbeit mit RAW Sockets finden sich in den C Bibliotheken des Betriebssystems bereits vordefinierte Datentypen für die Headerdaten aller gängigen Netzwerkprotokolle. Ein Datentyp für IP Headerdaten wird beispielsweise in der Headerdatei <netinet/ip.h> definiert als **struct iphdr** (siehe Listing 2).

Das Beispielprogramm `typecasting_example.c` in Listing 3 demonstriert die typische Anwendung des Type-Castings in der Socket-Programmierung.

```
1 #include <netinet/in.h> // Socket Funktionen
2 #include <netinet/ip.h> // IPv4 Header-Datenstruktur
3 #include <netinet/ip_icmp.h> // ICMPv4 Header-Datenstruktur
4
5 int main(int argc, char *argv[])
6 {
7     int s, bytes;
8     char packet[1500];
9
10    // Zeiger auf Header-Datenstrukturen anlegen.
11
12    struct iphdr *ip_hdr;
13    struct icmp_hdr *icmp_hdr;
14
15
16    // RAW Socket anlegen und davon lesen.
17
18    s = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
19
20    bytes = read(s, packet, sizeof(packet));
21
22    // Header-Datenstrukturen auf den Lesebuffer casten
23
24    ip_hdr = (struct iphdr*)packet;
25    icmp_hdr = (struct icmp_hdr*)(packet+sizeof(struct iphdr));
26
27    // Zugriff auf einzelne Felder der Paketheader
28
29    printf ("IP Version: %i\n", ip_hdr->version);
30    printf ("ICMP Type: %i\n", icmp_hdr->type);
31 }
```

Listing 3: Beispielprogramm: `typecasting_example.c`

In den Zeilen 12 und 13 werden Zeiger auf die benötigten Header-Datenstrukturen deklariert. In den Zeilen 24 und 25 findet das eigentli-

che Type-Casting statt: Dem Zeiger `ip_hdr` wird mit der Speicheradresse von `packet` die Startadresse der gelesenen Daten zugewiesen. Gleichzeitig wird festgelegt, dass die Daten, auf die der Pointer `ip_hdr` zeigt, ab sofort als `struct iphdr` interpretiert werden sollen.

Um die Speicheradressen der nachfolgenden Protokollheader zu bestimmen, wird die Größe der vorangehenden Header auf die Startadresse der Daten im Speicher addiert (z.B. `packet+sizeof(struct iphdr)`). Auf diese neuen Speicheradressen werden anschließend wieder die entsprechenden Header-Datenstrukturen gecastet (siehe Zeile 25).

Mit Hilfe der Felder der Header-Datenstrukturen kann nun direkt auf die einzelnen Daten der Protokollheader im gelesenen Paket zugegriffen werden, wie die Zeilen 29 und 30 beispielhaft zeigen.

Ein umfangreiches Beispiel für das Auslesen von Netzwerkpaketen mit zusätzlichem Type-Casting stellt das Programm `show_packet_verbose.c` dar, das im späteren Verlauf dieser Arbeit vorgestellt wird (vgl. Kapitel 3.3).

2.5 Position der Header im Speicher

Bei Verwendung eines (normalen) RAW Sockets findet sich der Header des Network-Layer Protokolls (im Allgemeinen also der Header des Internetprotokolls) direkt am Anfang der ausgelesenen Daten. Soll jedoch auf die Header eines höheren Protokolls oder gar auf die Nutzdaten des Netzwerkpaketes zugegriffen werden, so muss die Art und die Länge von jedem verwendeten Protokollheader bekannt sein, um die Position der jeweils verwendeten Daten bestimmen zu können.

Analog zu Zeile 25 in Listing 3 ließe sich die Startposition des Transport-Layer Headers im Speicher hinter einem IP Header z.B. wie folgt bestimmen:

```
transLayerPos = (int)packet+sizeof(struct iphdr);
```

Dabei geht man automatisch von einem IP Header mit fester, minimaler Größe, nämlich der Größe der Datenstruktur `struct iphdr`, aus. In Abbildung 3 ist jedoch erkennbar, dass der IP Header durch das Anhängen beliebig vieler Optionen durchaus eine variable Größe besitzen kann. Gleiches gilt auch für die Längen anderer Protokollheader. Daher ist die korrekte Bestimmung von Headerlängen unbedingt notwendig. Wenn ein Protokoll eine variable Headerlänge vorsieht, so wird die Headerlänge in der Regel in einem Pflichtfeld des jeweiligen Protokollheaders eingetragen. Im Falle des Internetprotokolls findet sich die Headerlänge im Feld IHL.

Bei der Interpretation des Wertes im Längensfeld eines Protokollheaders ist jedoch zu beachten, dass sich die Maße durchaus unterscheiden. Im IHL Feld des IP Protokollheaders ist beispielsweise die Anzahl der 32-Bit Wörter im IP Header gemeint. Der Mindestwert 5, bei einem IP Header ohne

anhängende Optionen, bezeichnet somit eine Headerlänge von $5 * 32 \text{ Bit} = 20 \text{ Bytes}$.

Die folgenden Codezeilen skizzieren die Extraktion der IP Headerlänge aus dem IP Header:

```
// Type-Casting
ip_hdr = (struct iphdr *)packet;
// Extraktion des Laengenfeldes
netLayerLgt = ((int)ip_hdr->ihl) * 4;
// Position des Transport-Layer Headers
transLayerPos = (int)packet+netLayerLgt;
```

Auch bei TCP sind variable Headerlängen möglich. Hier wird die aktuelle Headerlänge wieder als Anzahl von 32-Bit Worten angegeben. Diese findet sich jedoch im Feld Data Offset (doff).

Eine Liste der Feldnamen der wichtigsten TCP/IP Protokolle inkl. Anmerkungen und Beschreibung der protokollspezifischen Besonderheiten findet sich ab Anhang A.

2.6 Byte-Order

Die Unterscheidung von Network und Host Byte-Order ist bei der Netzkommunikation im Allgemeinen und bei der Betrachtung von Headerdaten im Besonderen unumgänglich.

Immer wenn ein Wert innerhalb eines Netzwerkpaketes einen Speicherplatz von mehr als einem Byte belegt, muss dieser Wert vor der Weiterverarbeitung auf dem Host zuerst von der Network Byte-Order (Big-Endian) in die Host Byte-Order umgewandelt werden. Im umgekehrten Fall muss eine Variable, die mehr als 1 Byte Speicherplatz beansprucht, vor dem Versand über einen Socket zuerst in die Network Byte-Order transformiert werden.

Für die genannten Transformationen hält das BSD Socket API folgende Funktionen bereit:

| Signatur | Umwandlung |
|---|-----------------|
| <code>uint32_t htonl(uint32_t hostlong)</code> | host-to-network |
| <code>uint16_t htons(uint16_t hostshort)</code> | host-to-network |
| <code>uint32_t ntohl(uint32_t netlong)</code> | network-to-host |
| <code>uint16_t ntohs(uint16_t netshort)</code> | network-to-host |

Die oben genannten Funktionen sind in der Headerdatei `<arpa/inet.h>` definiert.

Unter Linux kann davon ausgegangen werden, dass alle Daten, die an einen Socket gereicht, bzw. von einem Socket gelesen werden, in Network Byte-Order vorliegen. In anderen Unix-Derivaten (z.B. BSD) kann es jedoch

vorkommen, dass manche Felder eines Paketheaders in Host Byte-Order erwartet werden.

2.7 Ankunftszeit eines Netzwerkpaketes

Die genaue Ankunftszeit des zuletzt empfangenen Pakets lässt sich mittels eines `ioctl()` System-Calls direkt vom Betriebssystem Kernel ermitteln. Die Funktion `ioctl()` findet sich in der Headerdatei `<sys/ioctl.h>`. Als Parameter werden der Socket Deskriptor, der Call-Code `SIOCGSTAMP` und ein Zeiger auf eine `struct timeval` Variable an die Funktion `ioctl()` übergeben. Nach dem Funktionsaufruf findet sich die Ankunftszeit des letzten Paketes als Zeitstempel an der Zeigeradresse des `struct timeval` Parameters und kann z.B. mit Hilfe der Funktion `strftime()`, die in der Headerdatei `<time.h>` definiert ist, in einen lesbaren String umgewandelt werden. Listing 4 zeigt das Vorgehen beispielhaft.

```
1 #include <netinet/in.h>
2 #include <sys/ioctl.h>
3 #include <time.h>
4
5 int main(int argc, char *argv[])
6 {
7     // Variablen fuer Zeitangaben
8     time_t curtime;
9     struct timeval tv;
10    int    milliseconds;
11    char   time_buffer[30];
12
13    int    s, bytes;
14    char   packet[1500];
15
16    // Socket aufbauen und vom Socket lesen...
17    s = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
18    bytes = read(s, packet, sizeof(packet));
19
20    // Ankunftszeit des letzten Paketes ermitteln...
21    ioctl(s, SIOCGSTAMP, &tv);
22    curtime=tv.tv_sec;
23    milliseconds=(int)tv.tv_usec;
24    strftime (time_buffer, 30, "%Y-%m-%d, %T",
25             localtime(&curtime));
26    printf("\nZeit: %s.%06d\n", time_buffer, milliseconds);
27 }
```

Listing 4: Beispielprogramm: `timestamp_example.c`

2.8 Schreiben auf einen RAW Socket

Um auf einen RAW Socket zu schreiben, werden prinzipiell die gleichen Techniken verwendet, die auch schon zum Auswerten von empfangenem Datenverkehr nötig waren.

Zuerst wird ein Schreibpuffer, vorzugsweise ein Array vom Typ Unsigned Char oder Unsigned Short (Byte Array), im Speicher angelegt. Anschließend werden die erforderlichen Header-Datenstrukturen auf den Schreibpuffer gecastet. Der Puffer kann dann über die Felder der Header-Datenstrukturen mit Inhalt gefüllt werden. Am Ende wird der Puffer auf den Socket geschrieben.

Anders als beim Lesen von einem RAW Socket gibt es jedoch zu beachten, dass der komplette Network-Layer Header (IP Header) standardmäßig noch vom Betriebssystemkernel erstellt wird. Der mit normalen RAW Sockets generierbare Bereich beginnt also erst mit dem Transport-Layer Header (siehe 5).

Erst die Verwendung der Socket-Option `IP_HDRINCL` ermöglicht die manuelle Paketgenerierung des gesamten IP Paketes inklusive Network-Layer Header.

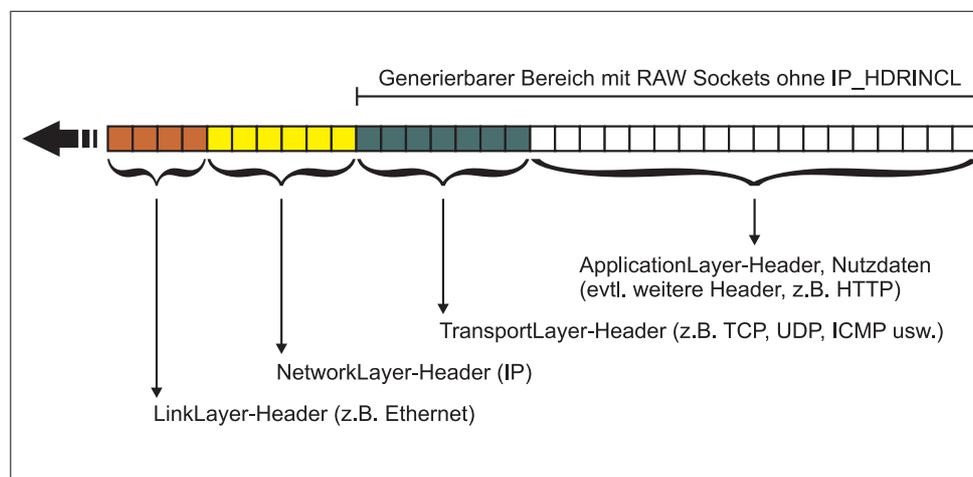


Abbildung 5: Paketgenerierung auf RAW Sockets ohne `IP_HDRINCL`

Das Beispiel `packet_gen_raw_easy.c` in Listing 5 zeigt den „normalen“, einfachsten Weg der Paketgenerierung. Im späteren Verlauf dieser Arbeit werden jedoch auch umfangreichere Möglichkeiten der vollständig manuellen Paketgenerierung auf Basis von RAW und PACKET Sockets vorgestellt.

```

1  #include <string.h>           // String und Memory Fkt
2  #include <errno.h>          // Error Handling
3  #include <sys/socket.h>      // Socket Funktionen
4  #include <arpa/inet.h>       // Wichtige Fkt wie htons()
5  #include <netinet/ip_icmp.h> // struct icmphdr
6
7  int main(int argc, char *argv[])
8  {
9
10     int sock, bytes;
11
12     // Die Groessen inerhalb des Pakets
13     unsigned int SIZE_ICMPHDR = sizeof(struct icmphdr);
14     unsigned int SIZE_RANDOMDATA = 10;
15     unsigned int SIZE_PACKET = SIZE_ICMPHDR + SIZE_RANDOMDATA;
16
17     // Der Puffer fuer das gesamte Paket
18     unsigned char packet[SIZE_PACKET];
19
20     // Zeiger auf Header-Datenstrukt. auf den Puffer casten
21     struct icmphdr *icmpheader = (struct icmphdr*) packet;
22     char *icmprandomdata = (char*)(packet + SIZE_ICMPHDR);
23
24     // Wichtig: Packet Buffer mit Nullen fuellen
25     memset(packet, 0, SIZE_PACKET);
26
27     // ICMP Headerdaten setzen (auf Byte-Order achten!)
28     icmpheader->type           = ICMP_ECHO;
29     icmpheader->code           = 0;
30     // Checksum muss zuerst Null sein. Berechnung folgt.
31     icmpheader->checksum       = 0;
32     icmpheader->un.echo.id     = htons(12345);
33     icmpheader->un.echo.sequence = htons(1);
34
35     icmprandomdata = "Hallo Welt!";
36
37
38     // Berechnung der ICMP Checksumme:
39     icmpheader->checksum = comp_cksum(
40         (unsigned short*)icmpheader,
41         SIZE_ICMPHDR+SIZE_RANDOMDATA);
42
43
44     // RAW Socket aufbauen
45     sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
46
47
48     // Empfaengeradresse in struct sockaddr_in ablegen.
49     struct sockaddr_in addr;

```

```

50     addr.sin_addr.s_addr = inet_addr("192.168.100.1");
51     addr.sin_family = AF_INET;
52     addr.sin_port = htons(0);
53
54
55     // Paket versenden
56     bytes = sendto(sock, packet, SIZE_PACKET, 0,
57                   (struct sockaddr*) &addr, sizeof(addr));
58 }

```

Listing 5: Beispielprogramm: packet_gen_raw_easy.c

In Listing 5 wird ein ICMP Echo Request Paket generiert und an die Adresse 192.168.100.1 gesendet. Der RAW Socket ohne `IP_HDRINCL` Option erwartet die zu sendenden Daten ohne IP Header. Daher wird in in Zeile 21 bereits die Datenstruktur des ICMP Headers auf die Startadresse des Schreibpuffers gecastet.

Hinter dem ICMP Header kann noch ein beliebig langes Feld mit beliebigem Inhalt folgen (`icmprandomdata`).

In Zeile 25 wird der gesamte Schreibpuffer mit Nullen aufgefüllt, da eine sorgfältige Initialisierung von Speicherbereichen bei der Generierung von Netzwerkpaketen besonders essentiell ist. Häufig kommt es vor, dass bestimmte Felder in Protokollheadern für den konkreten Anwendungsfall nicht relevant sind und somit leer bleiben sollen. Ein nicht initialisierter Schreibpuffer kann jedoch leicht zu einer unvorhersehbaren Belegung dieser nicht genutzten Felder führen und stellt eine ernstzunehmende Fehlerquelle dar.

Ab Zeile 27 wird der Schreibpuffer mit Hilfe der Felder der Datenstruktur `struct icmp_hdr` mit Inhalt gefüllt. Dabei ist, wie in Kapitel 2.6 beschrieben, zu beachten, dass Werte, die mehr als ein Byte Speicherplatz belegen, in die Network Byte-Order transformiert werden müssen.

Bei der Generierung des Headers oberhalb des Network-Layers erhält man prinzipiell keine Unterstützung durch die Socket API oder den Betriebssystemkernel. Daher muss z.B. auch die ICMP Prüfsumme selbst berechnet und eingetragen werden. Die Berechnung der Prüfsumme geschieht in Zeile 39 mit dem Aufruf der Funktion `comp_chksum()`, welche in Kapitel 2.10 näher vorgestellt wird. Vor der Prüfsummenberechnung müssen alle Felder des ICMP Headers inklusive dem Feld mit den zufälligen Daten (`icmprandomdata`) vollständig ausgefüllt worden sein, wobei einige Felder natürlich auch den Wert Null enthalten können. Das Feld `checksum` muss vor der Prüfsummenberechnung zwingend auf Null gesetzt werden.

Der Aufruf der `socket()` Funktion in Zeile 44 ist identisch mit den `socket()` Aufrufen aus den früheren Listings. Es wird ein RAW Socket aufgebaut, der auf Basis von IP Protokolladressen (`AF_INET`) arbeitet und auf

das ICMP Protokoll festgelegt wird.

Das fertige Paket wird in Zeile 55 mit Hilfe der Funktion `sendto()` versendet.

Das Programm `ping.c`⁵ von Felix Opatz, das sich auch im Begleitmaterial dieser Arbeit findet, bietet ein weiteres, ausführlicheres Beispiel, das den kombinierten Lese- und Schreibzugriff auf RAW Sockets ohne `IP_HDRINCL` Option demonstriert.

2.9 Adressierung von RAW Sockets

Für die Kommunikation über ein Netzwerk sind Adressen elementar. Jedes Paket, das über ein Netzwerk versendet wird, benötigt eine Absender- und eine Empfängeradresse. Eine Absenderadresse muss bei der Arbeit mit RAW Sockets nicht zwangsläufig vom Benutzerprogramm definiert werden. Das Betriebssystem setzt automatisch die Adresse des Interfaces, über welches das Paket gesendet werden soll, als Absenderadresse in jedes ausgehende Paket ein.

Dieses Vorgehen wird auch im Beispiel `packet_gen_raw_easy.c` in Listing 5 gezeigt. Mit der Funktion `bind()` kann jedoch auch explizit eine Absenderadresse definiert werden.

Im Gegensatz zur Absenderadresse ist die Definition einer Empfängeradresse in jedem Fall notwendig. Um einen Empfänger für jedes einzelne, ausgehende Paket zu definieren, bietet sich die Nutzung der Funktion `sendto()` an. `sendto()` wird in `<sys/socket.h>` definiert und besitzt folgende Signatur:

```
ssize_t sendto(int s, const void *msg, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
```

Neben dem Socket Deskriptor `s`, den zu sendenden Daten `msg`, der Länge der Daten `len` und den `flags` (können meist auf Null gesetzt werden) erwartet `sendto()` auch eine Empfängeradresse vom Typ `struct sockaddr` im Parameter `to`.

`struct sockaddr` ist eine Art „Dummy“-Datenstruktur, die im konkreten Anwendungsfall durch eine spezialisierte Socketadress-Datenstruktur ersetzt werden muss. Beispielsweise wird für IP Pakete die Datenstruktur `struct sockaddr_in` verwendet, welche auch im Zusammenhang mit gewöhnlichen TCP- oder UDP-Sockets eingesetzt wird. Die spezialisierte Socket-Adresse wird bei der Übergabe an `sendto()` in die allgemeine `struct sockaddr` gecastet.

⁵<http://www.zotteljedi.de/kleinkram/src/ping.c>

```

1  /* Socket address for IP based sockets */
2  struct sockaddr_in {
3
4      /* address family: AF_INET */
5      sa_family_t    sin_family;
6
7      /* port in network byte order */
8      u_int16_t      sin_port;
9
10     /* internet address          */
11     struct in_addr  sin_addr;
12 };
13
14 /* Internet address. */
15 struct in_addr {
16
17     /* address in network byte order */
18     u_int32_t      s_addr;
19 };

```

Listing 6: Datenstrukturen: struct sockaddr_in und struct in_addr aus <netinet/in.h>

Im Beispiel `packet_gen_raw_easy.c` in Listing 5 wird der typische Umgang mit Socketaddress - Datenstrukturen in den Zeilen 48-51 gezeigt. Die Daten in einer Socketaddress - Datenstruktur müssen bereits in Network Byte-Order vorliegen (siehe Kapitel 2.6). Das Feld `sin_port` spielt bei Protokollen, die keine Ports verwenden (z.B. ICMP), keine Rolle und kann mit dem Wert Null gefüllt werden.

Die möglicherweise missverständliche Abkürzung „in“ in „sockaddr_in“ (sowie in diversen anderen Bezeichnungen, die im Kontext von Sockets verwendet werden) steht übrigens nicht für „eingehend“ oder „hereinkommend“ sondern für „Internet“ und bezieht sich allgemein auf das Internet Protokoll (IP). Das Präfix „sin_“ in den Feldern der Datenstruktur steht analog etwa für „Internet Socket“ oder „IP basierter Socket“.

2.10 Die Internet Prüfsumme

Prüfsummen werden eingesetzt um Datenintegrität zu gewährleisten. Prüfsummen in Netzwerkprotokollen dienen in der Regel der Erkennung von fehlerhaft übertragenen Header- oder Nutzdaten. Die zu schützenden Daten werden nach einem bestimmten, mathematischen Verfahren in einen festen Wert, die Prüfsumme, transformiert, wobei veränderte Daten auch möglichst zu einer veränderten Prüfsumme führen sollen.

Die Protokolle der Internet Protokollfamilie (z.B. IP, UDP, TCP und ICMP) bilden ihre Prüfsumme stets nach dem gleichen Prinzip: Zuerst werden alle 16-Bit Halbworte der zu schützenden Daten nach den Regeln des Einerkomplements aufsummiert. Anschließend wird das Einerkomplement aus dieser Summe gebildet.

In RFC 1071 wird die Berechnung der Internet Prüfsumme ausführlich beschrieben und auch anhand von Beispiel-Implementierungen in verschiedenen Programmiersprachen veranschaulicht. Listing 7 zeigt die Implementierung in der Programmiersprache C.

```
1 unsigned short comp_chksum(unsigned short *addr, int len)
2 {
3     /**
4      * Quelle: RFC 1071
5      * Berechnet die Internet-Checksumme
6      * Gueltig fuer den IP, ICMP, TCP oder UDP Header
7      *
8      * *addr : Zeiger auf den Anfang der Daten
9      *         (Checksummenfeld muss Null sein)
10     * len   : Laenge der Daten (in Bytes)
11     *
12     * Rueckgabewert : Checksumme in Network Byte-Order
13     **/
14
15     long sum = 0;
16
17     while( len > 1 ) {
18         sum += *(addr++);
19         len -= 2;
20     }
21
22     if( len > 0 )
23         sum += * addr;
24
25     while (sum>>16)
26         sum = ((sum & 0xffff) + (sum >> 16));
27
28     sum = ~sum;
29
30     return ((u_short) sum);
31 }
```

Listing 7: Funktion `comp_chksum` zur Berechnung der Internet Prüfsumme

Obwohl sich die Prüfsummenberechnung für alle gängigen Internet Protokolle gleich gestaltet, variiert der Umfang der zu schützenden Daten von

Protokoll zu Protokoll. Die Prüfsumme im IP Header berechnet sich beispielsweise ausschließlich über die Daten des IP Headers, wobei die Nutzlast völlig außer Acht gelassen wird. Erst die Prüfsummen der Transport-Layer Protokolle integrieren auch die Nutzlast eines Pakets. So wird für die Berechnung der ICMP Prüfsumme der ICMP Header und die Nutzlast (hier: Random-Data) herangezogen. Für die TCP- und UDP-Prüfsumme wird zusätzlich noch ein sogenannter *Pseudo-Header* gebildet, der später noch genauer beleuchtet werden wird.

Die genauen Modalitäten der Prüfsummenbildung für ein bestimmtes Protokoll lassen sich nur in der zugehörigen Protokollspezifikation finden. Einen ersten Anlaufpunkt bieten die Kurzbeschreibungen ab Anhang A.

Das Verfahren zur Bildung der Prüfsumme ist vergleichsweise einfach, was zum Einen dazu führt, dass die Internet Prüfsumme keine optimale Sicherheit bietet. Zum Anderen ergeben sich jedoch durch die Einfachheit auch Möglichkeiten zur Performanzsteigerung beim ständigen Umgang mit der Internet Prüfsumme: Theoretisch müsste die Prüfsumme im IP Header bei jedem Hop, den das Paket zurücklegt, komplett neu berechnet werden, da der Time-To-Live Wert mit jedem Hop um Eins verringert wird. Das einfache Inkrementieren der Prüfsumme um Eins reicht jedoch aus, um wieder eine korrekte Prüfsumme zu erhalten.

Um die Berechnung der Prüfsumme weiter zu beschleunigen, lohnt sich ein Blick in die C Bibliotheken des Betriebssystems. Beispielsweise finden sich in `<asm/i486/checksum.h>` Funktionen wie `ip_compute_csum()` oder `ip_fast_csum()`, welche allein schon durch die Verwendung von Assemblercode einen Geschwindigkeitsvorteil bieten können. Durch die Verwendung von Assemblercode wird ein Programm allerdings abhängig von der zugrundeliegenden Rechnerarchitektur.

2.11 Schreibzugriff auf den IP Header mit IP_HDRINCL

Bei der Paketgenerierung mit Hilfe von normalen RAW Sockets können nur die Paketheader hinter dem IP Header von Hand generiert werden. Um auch auf den IP Header selbst zugreifen zu können, muss die Option `IP_HDRINCL` gesetzt werden. Dies geschieht über die Funktion `setsockopt()`, welche in `<sys/socket.h>` definiert ist und folgende Signatur besitzt:

```
int setsockopt(int s, int level, int optname,  
              const void *optval, socklen_t optlen);
```

Parameter `s` erwartet wie gewohnt den Socket Deskriptor. `level` gibt die Protokollebene an, auf die sich die gewünschte Socket-Option beziehen soll. Normalerweise wird hier die Konstante `SOL_SOCKET` angegeben, wel-

che die Ebene des verwendeten Socket Deskriptors bezeichnet. Im Falle der Socket-Option `IP_HDRINCL` muss hier jedoch `IPPROTO_IP` angegeben werden, da der normale RAW Socket noch nicht auf IP Ebene arbeitet. `optname` erwartet die gewünschte Socket-Option in Form einer Konstanten. Die Verfügbarkeit von Socket-Optionen ist stark betriebssystemabhängig. Unter Debian Linux finden sich mögliche Socket-Optionen beispielsweise in der Dokumentation des Betriebssystems oder in C Headerdateien⁶. Anhand der Präfixe der Optionskonstanten lassen sich auch Rückschlüsse auf den Level ziehen, auf dem die entsprechende Socket-Option Wirkung zeigt (`SO_*` Konstanten beziehen sich direkt auf einen Socket, `IP_*` Konstanten auf die IP Ebene usw.). In dieser Arbeit wird ausschließlich die Option `IP_HDRINCL` vorgestellt.

`optval` erwartet einen Zeiger auf den Wert, auf den die Socket-Option gesetzt werden soll. Zum Setzen (Aktivieren) der `IP_HDRINCL`-Option wird zuerst eine Integer-Variable mit dem Wert Eins gefüllt. Anschließend wird ein Zeiger auf diese Integer-Variable in dem Parameter `optval` an die Funktion `setsockopt()` übergeben. Der Parameter `optlen` erwartet die Länge des Wertes in `optval`.

Das Setzen der `IP_HDRINCL`-Option gestaltet sich schließlich wie folgt:

```
int on = 1;
setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
```

Nach dem Setzen der Socket-Option `IP_HDRINCL` erwartet der Socket ein beinahe vollständiges IP Paket inklusive IP Header als Input vor dem Senden (siehe Abbildung 6).

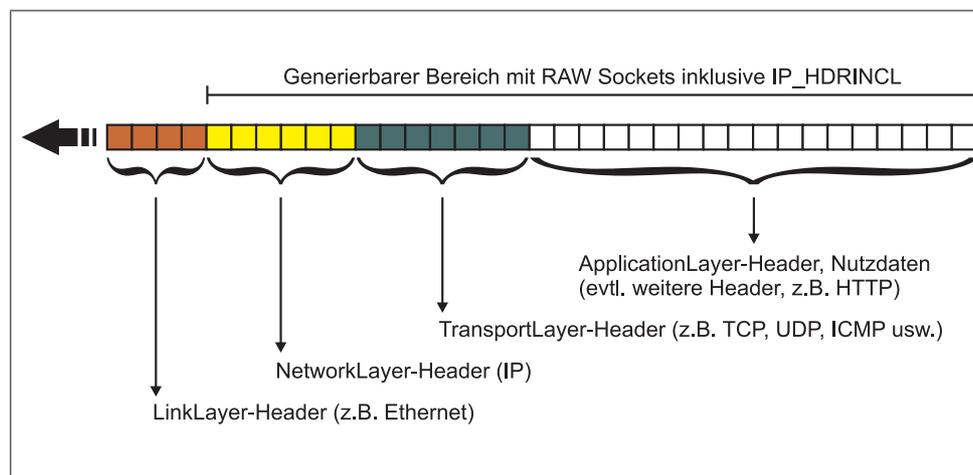


Abbildung 6: Paketgenerierung auf RAW Sockets inkl. `IP_HDRINCL`

⁶`SO_*` Konstanten finden sich z.B. unter „man 7 socket“, `IP_*` Optionen unter `<linux/in.h>` und `TCP_*` Optionen in `<linux/tcp.h>`

2.12 Beispiel: syn_packet_gen.c

Das Beispielprogramm `syn_packet_gen.c` in Anhang C zeigt, wie mit RAW Sockets und gesetzter `IP_HDRINCL` Option TCP SYN-Pakete inklusive IP Header generiert und gesendet werden können. Im Folgenden werden einige Ausschnitte aus diesem Programm vorgestellt.

Bei der Vorbereitung des Schreibpuffers muss darauf geachtet werden, dass der IP Header am Anfang des Schreibpuffers positioniert wird (siehe Listing 8). Hinter dem IP Header folgt der TCP Header. Da bei einem SYN Paket keine Nutzlast mitgesendet wird, endet das Paket nach dem TCP Header.

```
98     char                buffer[1024];
99     struct iphdr        *ip;
100    struct tcphdr       *tcp;
```

Listing 8: Ausschnitt aus `syn_packet_gen.c`: Variablendeklaration

```
133    // Vorbereiten des Schreibpuffers durch Type-Casting
134    ip = (struct iphdr*) buffer;
135    tcp = (struct tcphdr*) (buffer + sizeof(struct iphdr));
136
137    int iphdrlen = sizeof(struct iphdr);
138    int tcphdrlen = sizeof(struct tcphdr);
139    int datalen = 0;
```

Listing 9: Ausschnitt aus `syn_packet_gen.c`: Type Casting

Nun kann der IP Header über die Felder der Datenstruktur `struct iphdr` (siehe Anhang A.3.1) mit Inhalt gefüllt werden. Unter Verwendung normaler RAW Sockets und der `IP_HDRINCL` Option kann/muss man einige Felder automatisch vom Kernel ausfüllen lassen:

- Die IP Prüfsumme wird bei der Verwendung von normalen RAW Sockets grundsätzlich vom Kernel berechnet.
- Setzt man das ID Feld im IP Header auf Null, so setzt der Kernel automatisch einen gültigen Wert ein.
- Als Absenderadresse kann statt einer gültigen IP Adresse in Network Byte-Order die Konstante `INADDR_ANY` verwendet werden. In diesem Fall setzt der Kernel automatisch die korrekte IP Adresse des Host-Interfaces.

Listing 10 zeigt die Belegung des IP Headers im Programm `syn_packet_gen.c`:

```
142 // IP Header fuellen
143 memset(ip, 0, iphdrlen);
144 ip->version = 4;
145 ip->ihl = 5;
146 ip->tot_len = htons(iphdrlen + tcphdrlen);
147 // ID automatisch vom Kernel beziehen.
148 // Alternativ z.B.: ip->id = htons(1234);
149 ip->id = 0;
150 ip->ttl = 255;
151 ip->protocol = IPPROTO_TCP;
152 // Absenderadresse kann automatisch vom
153 // Kernel gesetzt werden: ip->saddr = INADDR_ANY;
154 // Vorsicht! TCP Pruefsumme wird dadurch evtl. falsch!
155 ip->saddr = inet_addr(argv[1]);
156 ip->daddr = inet_addr(argv[3]);
157 // IP Pruefsumme wird bei normalen RAW Sockets
158 // immer automatisch vom Kernel berechnet
159 ip->check = 0;
```

Listing 10: Ausschnitt aus `syn_packet_gen.c`: IP Header füllen

Der TCP Header wird auf die gleiche Weise generiert wie der IP Header. Die Datenstruktur `struct tcphdr` (siehe Anhang A.6.1) ermöglicht den Zugriff auf die einzelnen Felder des TCP Headers.

```
162 // TCP Header fuellen
163 memset(tcp, 0, tcphdrlen);
164 tcp->source = htons(atol(argv[2]));
165 tcp->dest = htons(atol(argv[4]));
166 tcp->seq = random();
167 tcp->doff = 5;
168 tcp->syn = 1;
169 tcp->window = htons(65535);
```

Listing 11: Ausschnitt aus `syn_packet_gen.c`: TCP Header füllen

2.13 Der TCP/UDP Pseudo-Header

Damit selbst generierte TCP-Pakete vom Kernel und von Netzwerk-Peers akzeptiert werden, muss die Prüfsumme des TCP Headers korrekt gebildet werden. Dazu muss zuerst ein sogenannter Pseudo-Header zusammengestellt werden, der sowohl Informationen aus dem IP als auch aus dem TCP Header integriert. Die Nutzung von Informationen aus dem IP Header (IP Adressen) soll die Erkennung von fehlerhaft gerouteten Paketen ermöglichen.

Die UDP Prüfsumme berechnet sich nach dem gleichen Schema wie die TCP Prüfsumme. Bei UDP ist die Prüfsumme optional und kann alternativ auch als Einer-Komplement von Null angegeben werden. Prinzipiell empfiehlt sich jedoch sowohl für TCP als auch für UDP immer die Bildung einer korrekten Prüfsumme.

Abbildung 7 zeigt den Aufbau des TCP/UDP Pseudo-Headers. Alle folgenden Darstellungen und Beschreibungen beziehen sich auf den Pseudo-Header unter Verwendung von IPv4. Unter IPv6 hat der Pseudo-Header einen anderen Aufbau.

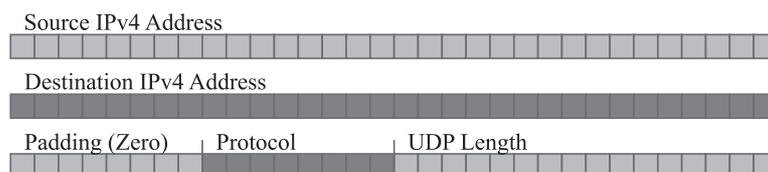


Abbildung 7: TCP/UDP Pseudo-Header (bei IPv4)

Die Felder *Source- bzw. Destination IPv4 Address* enthalten die Absender- bzw. Empfängeradresse aus dem IP Header. Bei IPv4 muss das Feld *Padding* mit Nullen gefüllt werden, damit die Länge des Pseudo-Headers ein Vielfaches von 16-Bit Worten erreicht.

Im Feld *Protocol* findet sich die Protokollnummer bzw. die `IPPROTO_*` Konstante des Transport-Layer Protokolls (z.B. TCP = `IPPROTO_TCP` = 6, UDP = `IPPROTO_UDP` = 17), die ebenfalls aus dem IP Header entnommen wird.

TCP/UDP Length enthält die Länge des TCP- oder UDP-Datagramms, also die Länge vom Beginn des TCP/UDP Headers bis zum Ende der Nutzlast, in Bytes. Der Pseudo-Header dient allein der Prüfsummenberechnung und wird zu keiner Zeit versendet.

Für die Generierung des Pseudo-Headers empfiehlt sich die Definition einer eigenen Datenstruktur:

```

1  #include <sys/types.h>
2  ...
3  struct pseudohdr
4  {
5      u_int32_t src_addr;
6      u_int32_t dst_addr;
7      u_int8_t padding;
8      u_int8_t proto;
9      u_int16_t length;
10 };

```

Listing 12: Datenstruktur für den TCP/UDP Pseudoheader: struct pseudohdr

Die TCP-/UDP-Prüfsumme wird über den Pseudo-Header, den Transport-Layer Header sowie über die Nutzdaten gebildet. Vor der Prüfsummenberechnung sollten daher alle benötigten Daten hintereinander im Speicher abgelegt werden. Im Programm `syn_packet_gen.c` (im Anhang C) wird dazu eine weitere Datenstruktur angelegt:

```

86  // Datenstruktur zur Zusammenfassung aller Daten,
87  // die zur Generierung der Prüfsumme notwendig sind:
88  // Pseudo-Header, TCP- bzw. UDP Header und Nutzlast
89  struct data_4_checksum
90  {
91      struct pseudohdr pshd;
92      struct tcphdr tcphd;
93      char payload[1024];
94  };

```

Listing 13: Ausschnitt aus `syn_packet_gen.c`: struct data_4_checksum

Die Gesamtlänge der notwendigen Daten muss nun einem Vielfachen von 16 Bit entsprechen. Andernfalls müssen am Ende noch weitere Nullen aufgefüllt werden. Die Prüfsumme wird am Ende als normale Internetprüfsumme gebildet (siehe Kapitel 2.10). Im Programm `syn_packet_gen.c` wird der TCP Pseudoheader ab Zeile 172 generiert. In Zeile 188f. folgt dann die Prüfsummenberechnung.

3 Programmierung mit PACKET Sockets

Unter Linux hält das Socket API neben den „normalen“ RAW Sockets noch einen weiteren Socket-Typ bereit: Den PACKET Socket.

Mit PACKET Sockets wird nun auch der Zugriff auf die bislang noch verborgen gebliebene zweite Schicht des TCP/IP Referenzmodells (vgl. Tabelle 1), den Data-Link-Layer, möglich.

3.1 Aufbau eines PACKET Sockets

Ursprünglich wurde ein PACKET Socket erstellt, indem der Socket-Typ `SOCK_PACKET` an die `socket()`-Funktion übergeben wurde. Ein `socket()`-Aufruf sah typischerweise wie folgt aus:

```
s = socket(AF_INET, SOCK_PACKET, htons(ETH_P_ALL));
```

Die Benutzung des Socket-Typs `SOCK_PACKET` gilt jedoch mittlerweile als veraltet, da hiermit keine Protokoll-Unabhängigkeit auf dem Physical-Layer gewährleistet ist. In dieser Arbeit wird zwar grundsätzlich die Verwendung von Ethernet auf dem Physical-Layer vorausgesetzt, jedoch ist die aktuell gültige Methode, einen PACKET Socket anzulegen, stets zu bevorzugen: Ein PACKET Socket wird dabei durch die Angabe der Address-Family `AF_PACKET` als Spezialfall eines RAW Sockets definiert.

```
s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

Es ist auch möglich, `AF_PACKET` zusammen mit `SOCK_DGRAM` zu verwenden, wodurch die Link-Layer Header bei jedem Paket abgeschnitten werden. Diese Möglichkeit wird in dieser Arbeit jedoch nicht weiter vorgestellt. Nähere Informationen dazu finden sich in der Linux Dokumentation (vgl. „man 7 packet“).

Der dritte Parameter der `socket()`-Funktion gibt an, mit welchen Protokollen der PACKET Socket arbeiten soll. Gültige Konstanten der Form `ETH_P_*` finden sich in `<linux/if_ether.h>`.

Bei der Verwendung von `ETH_P_ALL` akzeptiert der PACKET Socket beispielsweise alle einkommenden und ausgehenden Ethernet Frames. Mit `ETH_P_IP` oder `ETH_P_ARP` werden nur solche Frames akzeptiert, die ein IP- bzw. ARP-Paket beinhalten.

Da sich das Protokoll-Feld (oder „Ethertype“-Feld) im Ethernet-Header über zwei Bytes erstreckt, muss die Protokoll-Konstante bei der Übergabe an die `socket()`-Funktion in Network Byte-Order umgewandelt werden.

Durch die Verwendung von `AF_PACKET` auf Basis von `SOCK_RAW` Sockets sind nun sowohl beim Lese- als auch beim Schreibzugriff alle Paketdaten bis zum Data-Link-Layer Header zugreifbar. Bezogen auf Ether-

net Pakete handelt es sich dabei um den gesamten Ethernet Frame (siehe Anhang A.1) ohne die 4 Bytes lange CRC Prüfsumme am Ende. Die Präambel, der Start Frame Delimiter und die CRC Prüfsumme eines Ethernet-Pakets sind eher dem Physical-Layer zuzuordnen und werden vom Gerätetreiber grundsätzlich automatisch behandelt. Abbildung 8 liefert einen Überblick über die mit PACKET Sockets zugänglichen Bereiche eines Ethernet-Pakets.

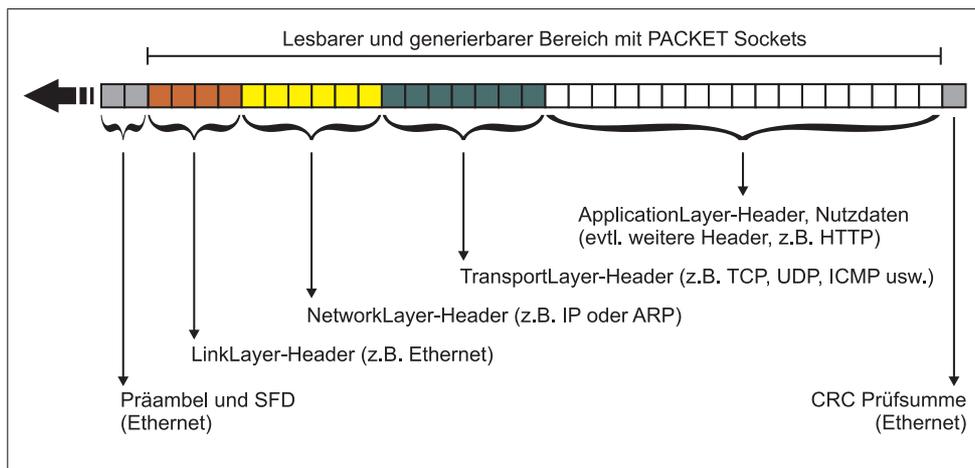


Abbildung 8: Zugänglicher Bereich von Ethernet-Paketen auf PACKET Sockets

3.2 Lesen von einem PACKET Socket

Das Lesen („Sniffen“) von einem PACKET Socket gestaltet sich ähnlich dem Lesen von einem normalen RAW Socket. Bei der Interpretation der gelesenen Daten muss jedoch beachtet werden, dass sich nun am Anfang der gelesenen Daten nicht mehr der IP Header sondern der Ethernet Header befindet.

Der Ethernet Header hat im Normalfall folgenden Aufbau:

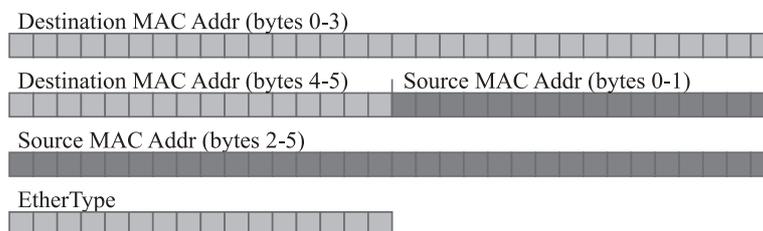


Abbildung 9: Headerdaten des Ethernet Protokolls

Der Aufbau von speziellen Ethernet-Paketen (z.B. VLAN tagged frames) weicht leicht von dem gezeigten Aufbau ab.

Beim Auslesen von Paketen über einen PACKET Socket kann mit Hilfe des *Protocol*-Parameters, der an die `socket()` - Funktion übergeben wurde, bestimmt werden, welche Pakete vom Socket angenommen werden. `ETH_P_ALL` sorgt dafür, dass jedes ankommende und ausgehende Paket jedes beliebigen Protokolls ausgelesen wird. Eine speziellere Protokollangabe, wie z.B. `ETH_P_IP`, führt jedoch dazu, dass nur noch ankommende Pakete des speziell vorgegebenen Protokolls (hier z.B. IP) an den Socket weitergegeben werden.

3.3 Beispiel: `show_packet_verbose.c`

Das Beispielprogramm `show_packet_verbose.c` im Begleitmaterial dieser Arbeit zeigt ausführlich, wie Ethernet-Pakete aus PACKET Sockets ausgelesen und mit Hilfe von vorgegebenen Datenstrukturen interpretiert werden können.

Das Programm ist eine Erweiterung des bereits in Kapitel 2.2 vorgestellten Programms `show_packet_binary.c`. Im Folgenden wird ein Überblick über die Arbeitsweise des Programmes gegeben, um einen schnellen Einstieg in den Programmcode zu ermöglichen. Weiterführende Informationen finden sich in den ausführlichen Kommentaren im Programmcode, sowie in den Protokollübersichten in Anhang A.

Nach der Deklaration zahlreicher Variablen in der `main()` Methode, z.B. zur Behandlung der verschiedenen Protokollheader, wird in Zeile 280 ein PACKET Socket mit Hilfe der `socket()` - Funktion erstellt (vgl. Kapitel 3.1).

Genau wie bei normalen RAW Sockets kann mit der `read()` - Funktion vom Socket gelesen werden. Momentan liest das Programm zu Demonstrationzwecken nur ein einziges Paket vom Socket. Dies kann jedoch leicht durch das Entfernen der Kommentarzeichen an der Endlos-`while` - Schleife (Zeile 293 und Zeile 481) geändert werden.

Nach dem erfolgreichen Auslesen eines Paketes wird zuerst die Datenstruktur `struct ether_header` mittels Type-Casting auf die Startposition des Lesepuffers gecastet.

```

324 // Der Data-Link-Layer beginnt a.d. Zeigeradr. "packet"
325 linkLayerPos = (int)packet;
326 linkLayerProt = "Ethernet";
327 linkLayerLgt = sizeof(struct ether_header);
328
329 // Type-Casting:
330 eth_hdr = (struct ether_header *)linkLayerPos;
331
332 // Pos. des Network-Layers im Speicher berechnen
333 netLayerPos = (int)packet + linkLayerLgt;

```

Listing 14: Ausschnitt aus show_packet_verbose.c: Type Casting des Data-Link-Layer Headers

Das Programm geht fest davon aus, dass es sich bei jedem gelesenen Paket um einen normalen Ethernet-Frame handelt. Da jeder (gewöhnliche) Ethernet Header eine feste Breite von 14 Bytes besitzt (`sizeof(struct ether_header)`), kann im gleichen Schritt auch schon die Startposition des Network-Layer Headers bestimmt werden (Zeile 332).

Nach dem Type-Casting sind die Informationen des Ethernet Headers direkt zugreifbar und werden nun mit Hilfe der Funktion `printEthernetInformation()` extrahiert und auf dem Bildschirm ausgegeben.

```

43 void printEthernetInformation (struct ether_header *eth,
44                               int hdr_len, int pkt_len)
45 {
46     printf("Data-Link-Layer Protokoll: Ethernet\n");
47     printf("MTU           : %i\n", ETHERMTU);
48     printf("Empfangen      : %i Bytes\n", pkt_len);
49     printf("Headerlaenge   : %i Bytes\n", hdr_len);
50
51
52
53     struct ether_addr *macaddr;
54     macaddr = (struct ether_addr *)eth->ether_shost;
55     printf("Absender MAC : %s\n", ether_ntoa(macaddr));
56     macaddr = (struct ether_addr *)eth->ether_dhost;
57     printf("EmpfaengerMAC: %s\n", ether_ntoa(macaddr));
58     printf("Ether-Type    : %04X\n\n", ntohs(eth->ether_type));
59 }

```

Listing 15: Ausschnitt aus show_packet_verbose.c: `printEthernetInformation()`

In der Funktion `printEthernetInformation()` wird auch gezeigt, wie sich die 48 Bit breiten MAC Adressen, die im Ethernet Header nur binär in Network Byte-Order vorliegen, in die gewohnte, menschenlesbare Hexadezimalform umwandeln lassen. Dazu kommt die Konvertierungsfunktion `ether_ntoa()` zum Einsatz. Da `ether_ntoa()` jedoch einen Zeiger auf die Datenstruktur `struct ether_addr` als Argument erwartet, müssen die Absender- und Empfängeradresse aus dem Ethernet Header zuerst noch in diesen Datentyp gecastet werden (Zeilen 53-57).

Zurück im Hauptprogramm wird nun das `ether_type` Feld aus dem Ethernet Header ausgewertet, um zu erkennen, welche Art von Protokollheader hinter dem Ethernet Header folgt (ab Zeile 346). Das Programm unterscheidet an dieser Stelle nur zwischen IP- und ARP-Paketen. Alle anderen Pakete meldet es als „unbekannt“.

Nun werden die einzelnen Paketheader Layer für Layer abgearbeitet. Dabei werden immer wieder passende Datenstrukturen auf den Lesepuffer gecastet. Anschließend werden über passende `print*()` - Funktionen jeweils eine kleine Auswahl an Headerinformationen auf dem Bildschirm ausgegeben. Das Programm unterscheidet momentan über eine `switch` Anweisung in der `main()` - Methode zwischen den IP Protokollen ICMP, TCP und UDP.

Nach der Abarbeitung der Protokollheader versucht die Funktion `printPayloadInformation()` die Daten der Payload als ASCII Zeichen zu interpretieren und auf dem Bildschirm auszugeben. Dabei werden nicht druckbare Zeichen durch ein `#` ersetzt.

Am Ende gibt das Programm die Inhalte des gesamten Pakets ähnlich wie `show_packet_binary.c` noch mal in binärer Darstellung aus. Diesmal werden die einzelnen Header jedoch sichtbar voneinander getrennt.

3.4 Schreiben auf einen PACKET Socket

Beim Senden über einen PACKET Socket werden Netzwerkpakete grundsätzlich inklusive Data-Link-Layer Header generiert. Der zu generierende Bereich entspricht also dem lesbaren Bereich eines PACKET Sockets. Auch hier entfällt die 4 Bytes lange CRC Prüfsumme am Ende des Ethernet-Frames (siehe Abbildung 8).

Die Generierung eines Netzwerkpakets erfolgt bei PACKET Sockets vollständig manuell. Es gibt hier keine Möglichkeiten mehr, Header oder Headerteile automatisch vom Betriebssystem generieren zu lassen. Die einzige Ausnahme bilden die Padding Bytes vor dem Ethernet CRC-Feld (siehe Anhang A.1): Obwohl das Ethernet Protokoll für einen Ethernet-Frame ei-

ne Mindestlänge von 64 Bytes (60 Bytes + 4 Bytes CRC Prüfsumme) vorschreibt, lassen sich auch kürzere Pakete über PACKET Sockets versenden. Die Treibersoftware des Netzwerk-Interfaces füllt in diesem Fall die fehlenden Bytes automatisch mit Nullen auf. Die CRC Prüfsumme ist, wie schon in Abbildung 8 gezeigt, nicht mit PACKET Sockets zugreifbar und wird somit grundsätzlich vom Gerätetreiber berechnet und eingesetzt.

3.5 Beispiel: arp_packet_gen.c

Das Beispielprogramm `arp_packet_gen.c` in Anhang E demonstriert das Versenden von ARP REPLY Paketen über einen PACKET Socket. Im Folgenden wird die Arbeitsweise des Programms skizziert. Es werden nur diejenigen Stellen im Programm genauer beschrieben, die sich von der bereits vorgestellten Paketgenerierung über RAW Sockets (vgl. Kapitel 2.8 und 2.11) unterscheiden. Ausführlichere Informationen zu den verwendeten Protokollen Ethernet und ARP finden sich in Anhang A.

Das ARP Protokoll setzt direkt auf dem Data-Link-Layer Protokoll auf. Aufgabe des Protokolls ist die Zuordnung von Hardware-Adressen (Data-Link-Layer Adressen) zu gegebenen Protokoll-Adressen (Network-Layer Adressen). In diesem Beispiel wird einer gegebenen IPv4-Adresse eine Ethernet-Adresse (MAC Adresse) zugeordnet. Normalerweise startet eine ARP Kommunikation mit einer Anfragenachricht (ARP REQUEST), welche mit einer ARP REPLY Nachricht beantwortet wird. Das Beispielprogramm `arp_packet_gen.c` versendet jedoch zu Demonstrationszwecken nur eine „unangefragte“ ARP REPLY Nachricht.

Das zu versendende ARP Paket setzt sich aus einem einfachen Ethernet Header (vgl. Abbildung 9) und einem auf IPv4 und Ethernet zurechtgeschnittenen ARP Header zusammen. Weitere Protokollheader oder eine Payload werden nicht benötigt.

Auf den in Zeile 57 angelegten Schreibpuffer *packet* werden daher in den Zeilen 100-102 die Datenstrukturen `struct ether_header` und `struct arp_header` gecastet. Die Definition von `struct ether_header` findet sich bereits in `<net/ethernet.h>`. `struct arp_header` wird hingegen direkt im Programmcode in den Zeilen 37-47 definiert, damit die Datenstruktur direkt an IPv4 und Ethernet angepasst werden kann.

In den Zeilen 69 bis 84 werden die Absender- und Empfängeradressen, sowie der Name der Netzwerkschnittstelle in menschenlesbarer Form festgelegt. Später werden diese Angaben mit Hilfe von Konvertierungsfunktionen in die entsprechende, binäre Form transformiert und in den Ethernet bzw. ARP Header eingetragen (Zeilen 105-133).

Ab Zeile 136 wird das Senden des Netzwerkpaketes vorbereitet. Zuerst wird ein PACKET Socket angelegt, der auf das ARP Protokoll festgelegt wird (`ETH_P_ARP`), dann wird die Nummer der Netzwerkschnittstelle ermittelt damit eine gültige Socket-Adresse generiert werden kann und schließlich wird das fertige Paket in Zeile 164 mittels `sendto()` versendet.

3.6 Adressierung von PACKET Sockets

Pakete, die über PACKET Sockets versendet werden, müssen vollständig generiert werden. Sämtliche Protokoll- oder Hardware-Adressen von Sender und Empfänger müssen folglich schon korrekt in den Paketheadern vorliegen. Trotzdem fordert das Socket API vor dem Versenden eines Paketes z.B. im Parameter *to* der Funktion `sendto()` eine Socket-Adresse.

Eine Socket-Adresse wird vom Socket API stets in der allgemeinen Form `struct sockaddr` verlangt. Für einen konkreten Anwendungsfall muss diese „Dummy“-Datenstruktur, wie auch schon in Kapitel 2.9 beschrieben, durch eine spezialisierte Datenstruktur ersetzt werden. AF_PACKET Sockets (also nicht die veralteten SOCK_PACKET Sockets) verwenden die Datenstruktur `struct sockaddr_ll`, die in `<linux/if_packet.h>` definiert wird (vgl. auch „man 7 packet“):

```
1 struct sockaddr_ll {
2     unsigned short sll_family;    /* Always AF_PACKET */
3     unsigned short sll_protocol; /* Physical layer protocol */
4     int sll_ifindex;             /* Interface number */
5     unsigned short sll_hatype;   /* Header type */
6     unsigned char sll_pkttype;   /* Packet type */
7     unsigned char sll_halen;     /* Length of address */
8     unsigned char sll_addr[8];   /* Physical layer address */
9 };
```

Listing 16: Datenstruktur: `struct sockaddr_ll` aus `<linux/if_packet.h>`

Beim Versenden von vollständigen Netzwerkpaketen ist nur noch die Bezeichnung der Netzwerkschnittstelle relevant, da alle übrigen Informationen aus `struct sockaddr_ll` bereits in den Paketheadern eingetragen wurden. `struct sockaddr_ll` sieht für die Angabe der Netzwerkschnittstelle das Feld `sll_ifindex` vor, in dem die Nummer des Interfaces abgelegt werden muss.

Um zu einem gegebenen Interfacenamen (z.B. „eth1“) die zugehörige Interfacennummer ermitteln zu können, existiert unter Linux das so-

nannte *Netdevice*-Interface. Das Netdevice-Interface beschreibt, wie mittels `ioctl()` - Systemcalls verschiedene Eigenschaften von Netzwerkschnittstellen abgefragt oder verändert werden können. Die Anfragen von Netdevice können auf beliebigen Sockets ausgeführt werden und basieren stets auf der Datenstruktur `struct ifreq`, die in `<net/if.h>` definiert wird. Interfacenummern lassen sich mit dem `ioctl()` - Callcode `SIOCGIFINDEX` ermitteln. Weitere Callcodes finden sich in der Linux-Dokumentation⁷.

Das Beispielprogramm `get_if_number_example.c` in Listing 17 demonstriert die vollständige Abfrage einer Interfacenummer zu einem gegebenen Interfacenamen. Die gleiche Methode wird auch im Beispielprogramm `arp_packet_gen.c` in Anhang E angewandt.

```
1 #include <stdio.h>           // Standard C Funktionen
2 #include <string.h>         // memset()
3 #include <sys/socket.h>     // socket()
4 #include <sys/ioctl.h>     // ioctl()
5 #include <net/if.h>        // struct ifreq
6
7 int main(int argc, char *argv[])
8 {
9     // Name der Netzwerkschnittstelle
10    char *dev = "eth1";
11
12    // Anfrage-Datenstruktur fuer Network Interfaces
13    struct ifreq ifr;
14    memset(&ifr, 0, sizeof(ifr));
15
16    // Interfacename in Anfrage-Datenstruktur kopieren
17    strncpy(ifr.ifr_name, dev, sizeof(ifr.ifr_name));
18
19    // Beliebigen Socket erstellen
20    int s = socket(AF_INET, SOCK_STREAM, 0);
21
22    // Indexnummer des Network Interfaces erfragen
23    ioctl(s, SIOCGIFINDEX, &ifr);
24
25    // Interfacenummer ausgeben
26    printf ("Interfacename   : %s\n", dev);
27    printf ("Interfacenumber: %i\n", ifr.ifr_ifindex);
28
29    return 0;
30 }
```

Listing 17: Beispielprogramm: `get_if_number_example.c`

⁷Linux-Manpages: `man 7 netdevice`

Ab Zeile 156 im Beispielprogramm `arp_packet_gen.c` in Anhang E wird gezeigt, wie die ermittelte Interfacennummer in die Socketadress-Datenstruktur `struct sockaddr_ll` übernommen wird, um dann in Zeile 164f. als Empfängeradresse zu dienen.

Alternativ kann die gleiche Socketadress-Datenstruktur auch in der Funktion `bind()` verwendet werden. Der Socket wird dadurch dauerhaft an das angegebene Interface gebunden. Statt `sendto()` kann dann auch `send()` oder `write()` ohne Angabe eines Empfängers verwendet werden, um Pakete über den PACKET Socket zu versenden. Diese Variante wird im erweiterten Beispielprogramm `icmp_packet_gen.c` aus dem folgenden Kapitel gezeigt.

3.7 Erweitertes Beispiel: `icmp_packet_gen.c`

Das Programm `icmp_packet_gen.c` in Anhang F bietet ein weiteres Beispiel für die Paketgenerierung mit PACKET Sockets. Das Programm versteht sich als Erweiterung des bereits vorgestellten Programms aus Anhang E (`arp_packet_gen.c`) und zeigt nun auch die Generierung von IP- bzw. ICMP-Paketen über PACKET Sockets.

Das Funktionsprinzip bleibt in dem erweiterten Beispiel erhalten, jedoch wird hier deutlich, dass die Automatismen, die noch bei der Arbeit mit normalen RAW Sockets genutzt werden konnten (vgl. Kapitel 2.11), hier nicht mehr zum Tragen kommen. So muss bei der Arbeit mit PACKET Sockets der komplette IP Header (inklusive Prüfsumme, Identifikationsnummer und IP-Adressen) immer von Hand generiert werden.

4 Über die Arbeit mit RAW Sockets

Das Generieren von Paketen mittels RAW Sockets oder PACKET Sockets ist eine diffizile Angelegenheit. Zum Einen werden fehlerhaft konstruierte Pakete vom eigenen Betriebssystem in der Regel gar nicht erst abgesendet. Zum Anderen reagieren die Gegenstellen im Netzwerk, je nach zugrundeliegendem Betriebssystem oder angewandter Sicherheitspolicy auch auf formal korrekte Pakete sehr unterschiedlich oder sogar überhaupt nicht. So sorgen die je nach Betriebssystem unterschiedlichen Implementierungen des TCP/IP Protokoll-Stacks dafür, dass typische Protokollabläufe im Detail unterschiedlich interpretiert werden und der Aufbau von Netzwerk-Paketen leicht variiert.

Hauptsächlich zeigen sich solche Unterschiede an Stellen, die von Seiten der Protokollspezifikationen absichtlich variabel gehalten wurden (z.B. Random-Data Felder, ID Nummern, optionale Felder usw.) oder an Stellen, an denen es um die Einschätzung von Plausibilität, Sicherheit oder Performanz geht: Ein ARP_REPLY Paket, das ohne vorherige Anfrage per ARP_REQUEST an einem Ubuntu 7.10 Rechner eingeht wird ignoriert. Wurde jedoch zuvor ein ARP_REQUEST versendet, so akzeptiert Ubuntu ein ARP_REPLY nicht nur wenn das ARP_REPLY Paket direkt an die eigene MAC Adresse adressiert wurde, sondern auch dann, wenn es an die Ethernet-Broadcast Adresse geschickt wurde. Die Target IP Adresse, die in einem ARP_REPLY durchaus angegeben wird, wird von Ubuntu in diesem Fall völlig ignoriert. Eine andere ARP Implementierung könnte hier völlig anders reagieren.

In Rechnernetzen existieren unterschiedliche Vorstellungen von „korrekten“ Protokollabläufen, einem „korrekten“ Aufbau von Netzwerkpaketen und einer Toleranz gegenüber Abweichungen. Die zum Teil unvorhersehbaren Effekte, die sich daraus ergeben, erschweren zwar die Fehlersuche bei der Arbeit mit RAW Sockets, jedoch erweist sich die RAW Socket Programmierung gleichzeitig auch als flexibles Werkzeug, um verschiedenste Experimente in einem Rechnernetz durchzuführen oder das Verhalten von Betriebssystemen und Netzwerksoftware gezielt zu analysieren und zu testen.

Vorschläge für weitere Arbeiten mit RAW Sockets sind: Die Analyse des Verhaltens unterschiedlicher Netzwerksoftware (z.B. Serversoftware oder Protokollstacks verschiedener Betriebssysteme), die Analyse oder der Aufbau von Intrusion Detection - bzw. Intrusion Prevention Systemen, die Nutzung von IPv6 mit RAW Sockets, die Untersuchung von typischen Hacker-Angriffsstrategien (z.B. Denial of Service oder Man-in-the-Middle Attacken),

die experimentelle Entwicklung neuer Protokolle auf Basis von IP oder Ethernet oder die Benutzung alternativer Data-Link-Layer Techniken (z.B. WLAN, o.ä.) mit PACKET Sockets.

Abschließend sei auf die Betrachtung von universellen Packet Capturing Bibliotheken (wie z.B. pcap⁸ oder libnet⁹) als besonders sinnvolle Ergänzung für den Zugang zu einer tiefschichtigen Netzwerk-Programmierung hingewiesen. Mit diesen Bibliotheken wird das Capturing und die Paketgenerierung gekapselt und somit auf einer höheren Programmierenebene ausgeführt. Betriebssystemunabhängige Programme werden somit ermöglicht.

⁸<http://www.tcpdump.org/>

⁹<http://www.packetfactory.net/projects/libnet/>

A Kurzübersich über Protokolldaten

Im Folgenden werden einige für die Socket Programmierung wichtige Aspekte verschiedener Protokolle anhand von Schemata, Beschreibungen und Codefragmenten kurz vorgestellt. Der gesamte Anhang A versteht sich als Nachschlagewerk bei der Arbeit mit RAW Sockets.

Eine noch kompaktere Darstellung findet sich z.B. im *TCP/IP and tcp-dump Pocket Reference Guide*¹⁰ des SANS Institutes.

¹⁰<http://www.sans.org/resources/tcpip.pdf>

A.1 Das Ethernet Protokoll

Der Aufbau eines Ethernet-II-Paketes (nach IEEE 802.3) wird in Abbildung 10 gezeigt (1 Kästchen entspricht einem Byte):

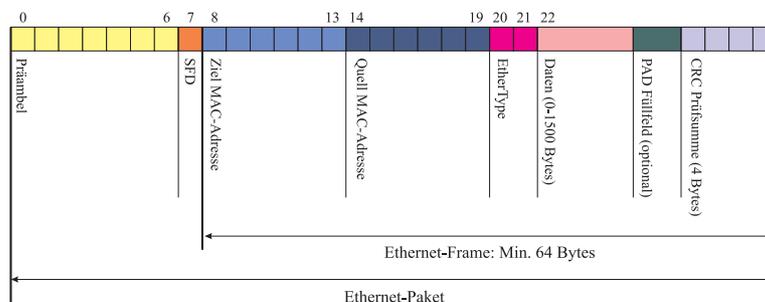


Abbildung 10: Ein Ethernet-Paket

Es wird unterschieden zwischen einem Ethernet-*Paket* und einem Ethernet-*Frame*.

Präambel und *Start Frame Delimiter (SFD)* werden bei der Socket Programmierung ebensowenig berücksichtigt wie die anhängende *CRC Prüfsumme*. Die entsprechenden Bytes werden beim Empfang eines Ethernet-Paketes bereits durch den Gerätetreiber abgetrennt bzw. vor dem Versand hinzugefügt.

Die erste relevante (sichtbare) Information beginnt folglich mit dem Header des Ethernet Frames (im Folgenden nur noch *Ethernet-Header* genannt).

Der Ethernet Header hat praktisch eine feste Größe von 14 Bytes (siehe Abbildung 11). Weitere Headerdaten (z.B. zum *VLAN Tagging*) werden durch einen entsprechenden Wert im Feld *EtherType* angekündigt und können genauso wie höhere Protokoll Daten behandelt werden.

Der gesamte Ethernet-Frame muss mindestens 64 Bytes lang sein. Sendet man kleinere Pakete über einen Socket, so hängt der Gerätetreiber automatisch weitere Padding Bytes (PAD Füllfeld) an das Paket an. Diese automatisch aufgefüllten Bytes sind beim gleichzeitigen „sniffen“ der gesendeten Pakete noch nicht sichtbar.

Die Nutzdaten nach dem *EtherType* (also inkl. aller weiteren Header von höheren Protokollen, jedoch ohne die CRC Prüfsumme) dürfen höchstens 1500 Bytes belegen.

A.1.1 Bit-Schema

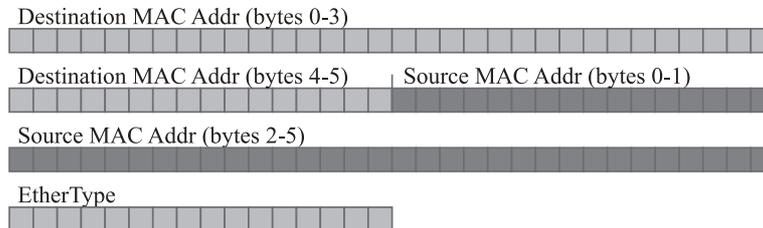


Abbildung 11: Header Daten des Ethernet Protokolls

Destination MAC Addr: MAC Adresse des Empfängers

Source MAC Addr: MAC Adresse des Senders

EtherType: Das verwendete Network-Layer Protokoll
(z.B.: IPv4 = 0x0800, ARP = 0x0806)
siehe `ETHERTYPE_*` Konstanten in `<net/ethernet.h>`

A.1.2 Datenstruktur

```
1 struct ether_header
2 {
3     u_int8_t ether_dhost[ETH_ALEN]; /* destination eth addr */
4     u_int8_t ether_shost[ETH_ALEN]; /* source ether addr */
5     u_int16_t ether_type;           /* packet type ID field */
6 };
```

Listing 18: struct ether_header aus `<net/ethernet.h>`

A.1.3 Weitere Informationen / Quellen

- IEEE 802.3: <http://www.ieee802.org/3/>
- Ethernet Numbers: <http://www.iana.org/assignments/ethernet-numbers>
- Ethernet (Wikipedia): <http://de.wikipedia.org/wiki/Ethernet>

A.2 Das ARP Protokoll

Mit dem *Address Resolution Protocol* (ARP) lässt sich zu einer bereits bekannten *Protokolladresse* (Network-Layer Adresse, z.B. IP-Adresse) die zugehörige *Hardwareadresse* (Data-Link-Layer Adresse, z.B. MAC-Adresse) erfragen.

A.2.1 Bit-Schema

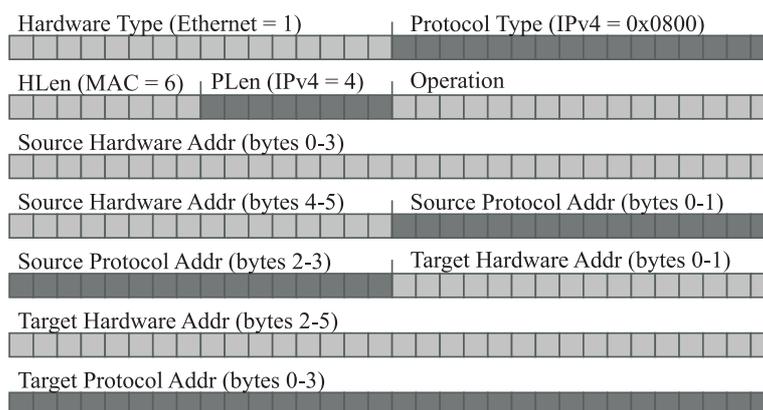


Abbildung 12: Header Daten des ARP Protokolls

Hardware Type: Typ der Hardwareadresse (Data-Link-Layer Adresse)
(z.B.: Ethernet = 1)
siehe `ARPHRD_*` Konstanten in `<net/if_arp.h>`

Protocol Type: Typ der Protokolladresse (Network-Layer Adresse)
(z.B.: IPv4 = 0x0800)
siehe `ETHERTYPE_*` Konstanten in `<net/ethernet.h>`

HLen: Länge der Hardwareadresse (Data-Link-Layer) in Bytes
(z.B.: MAC Adresse = 6 Bytes)

PLen: Länge der Protokolladresse (Network-Layer) in Bytes
(z.B.: IPv4 Adresse = 4 Bytes)

Operation: ARP Operation
(z.B.: ARP Request = 1, ARP Reply = 2)
siehe `ARPOP_*` Konstanten in `<net/if_arp.h>`

Source Hardware Addr: Data-Link-Layer Adresse des Senders
(z.B. die 48 Bits lange MAC Adresse des Senders)

Source Protocol Addr: Network-Layer Adresse des Senders
(z.B. die 32 Bits lange IPv4 Adresse des Senders)

Target Hardware Addr: Data-Link-Layer Adresse des Empfängers
(im Falle eines ARP Requests nur Nullen, da diese Adresse erfragt werden soll)

Target Protocol Addr: Network-Layer Adresse des Empfängers

A.2.2 Datenstruktur

Die Datenstruktur in `<net/if_arp.h>` definiert den ARP Header nur bis zum Operation-Feld, da sich die nachfolgenden Adressfelder je nach eingesetztem Protokoll und eingesetzter Hardware in der Länge unterscheiden. Hier findet sich eine für IPv4 und Ethernet modifizierte Variante, die direkt in ein C Programm übernommen werden kann.

```
1 struct arp_header {
2     unsigned short int ar_hrd; // Format of hardware address
3     unsigned short int ar_pro; // Format of protocol address
4     unsigned char  ar_hln;    // Length of hardware address
5     unsigned char  ar_pln;    // Length of protocol address
6     unsigned short int ar_op; // ARP opcode (command).
7     unsigned char  ar_sha[6]; // Sender hardware addr (MAC)
8     unsigned char  ar_sip[4]; // Sender IP address
9     unsigned char  ar_tha[6]; // Target hardware addr (MAC)
10    unsigned char  ar_tip[4]; // Target IP address
11 };
```

Listing 19: struct arphdr aus `<net/if_arp.h>` (modifiziert für IPv4 und Ethernet)

A.2.3 Weitere Informationen / Quellen

- RFC 826: <http://tools.ietf.org/html/rfc826>
- ARP Parameters: <http://www.iana.org/assignments/arp-parameters>
- Adress Resolution Protocol (Wikipedia):
http://de.wikipedia.org/wiki/Address_Resolution_Protocol
- Linux-Manpage: `man 7 arp`

A.3 Internet Protokoll (IP) Header

A.3.1 Bit-Schema

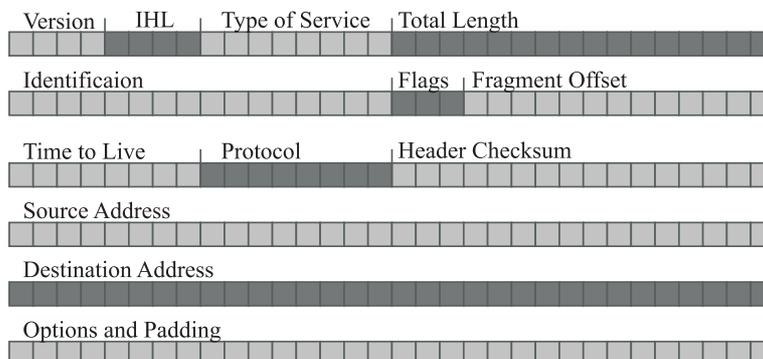


Abbildung 13: Header Daten des Internet Protokolls (IP)

Version: Version des verwendeten IP Protokolls

(z.B.: IPv4 = 4, IPv6 = 6)

siehe `<netinet/ip.h>` oder `<netinet/ip6.h>`

IHL: Internet Header Length

Länge des IP Headers in 32-Bit Worten (mindestens 5).

Type of Service (TOS): Güte des IP Dienstes.

In diesem Feld kann ein Wunsch nach Priorisierung des Paketes eingetragen werden.

(z.B.: Keine Angabe = 0)

Für die Kodierungsvorschrift siehe z.B. [5], S.28.

siehe `IP_TOS_*` Konstanten in `<netinet/ip.h>`

Total Length: Gesamtlänge des IP Pakets (incl. IP Header) in Bytes.

Identification: Nummer zur „eindeutigen“ Identifikation eines Paketes.

Dient dem Wiederausammensetzen fragmentierter Pakete.

Aufgrund des eingeschränkten Speicherplatzes für die Identification Nummer (16 Bit) können sich die Zahlen jedoch nach einer gewissen Zeit wiederholen.

Flags: Bitmaske für Fragmentierungs-Informationen.

Die angegebenen Konstanten sind in `<netinet/ip.h>` vordefiniert.

Bit0: **reserved** (`IP_RF`): Muss immer Null sein.

Bit1: **Don't fragment** (`IP_DF`): 1 bedeutet, dass das Paket nicht fragmentiert werden darf.

Bit2: **More Fragments** (`IP_MF`): 1 bedeutet, dass diesem Fragment

noch weitere Fragmente des gleichen Paketes folgen.

In der Datenstruktur `struct iphdr` (siehe unten) werden die Flags und der Fragment Offset in einem Feld zusammengefasst.

Fragment Offset: Der Wert in diesem Feld multipliziert mit 8 ergibt die Position (in Byte) des aktuellen Fragments im Gesamtdatagramm.

Time to Live: Zähler, der bei jedem Hop, den das Paket auf seinem Weg zum Ziel überquert, um 1 heruntergezählt wird. Wenn der Wert Null erreicht wird, wird das Paket verworfen.
Übliche Startwerte sind z.B. 255, 128 oder 64.

Protocol: Nummer des nächst höheren Protokolls, das in diesem IP Paket transportiert wird.
(z.B. ICMP = 1, TCP = 6, UDP = 17)
siehe `IPPROTO_*` Konstanten in `<netinet/in.h>`

Header Checksum: Prüfsumme, die über den IP Header gebildet wird, um Datenintegrität zu gewährleisten.
Vor Berechnung der Prüfsumme muss das Checksum-Feld mit Nullen gefüllt werden.
Ein IP Paket mit falscher Prüfsumme wird entweder gar nicht erst abgesendet oder spätestens beim Empfänger verworfen.
Eine genaue Beschreibung der Prüfsummenbildung findet sich z.B. in „RFC 1071“.

Source Address: IP Adresse des Absenders in Network Byte-Order.
siehe dazu `struct in_addr` in `<netinet/in.h>`

Destination Address: IP Adresse des Empfängers in Network Byte-Order.
siehe dazu `struct in_addr` in `<netinet/in.h>`

Options: Dieses Feld ist optional und kann komplett ausgelassen werden.
siehe dazu `IPOPT_*` Konstanten in `<netinet/ip.h>`

Padding: Die Länge des Options-Feldes muss ein Vielfaches von 32-Bit betragen. Reicht die Länge nicht aus, so wird der freie Platz mit Nullen aufgefüllt (Padding).

A.3.2 Datenstruktur

```
1 struct iphdr
2 {
3     #if __BYTE_ORDER == __LITTLE_ENDIAN
4         unsigned int ihl:4;
5         unsigned int version:4;
6     #elif __BYTE_ORDER == __BIG_ENDIAN
7         unsigned int version:4;
8         unsigned int ihl:4;
9     #else
10    # error "Please fix <bits/endian.h>"
11    #endif
12    u_int8_t tos;
13    u_int16_t tot_len;
14    u_int16_t id;
15    u_int16_t frag_off;
16    u_int8_t ttl;
17    u_int8_t protocol;
18    u_int16_t check;
19    u_int32_t saddr;
20    u_int32_t daddr;
21    /*The options start here. */
22 };
```

Listing 20: struct iphdr aus <netinet/ip.h>

A.3.3 Weitere Informationen / Quellen

- RFC 791 (IP): <http://tools.ietf.org/html/rfc791>
- RFC 1071 (IP Checksum): <http://tools.ietf.org/html/rfc1071>
- Protocol Numbers: <http://www.iana.org/assignments/protocol-numbers>
- Linux-Manpage: `man 7 ip`

A.4 ICMP Protokoll Header

A.4.1 Bit-Schema

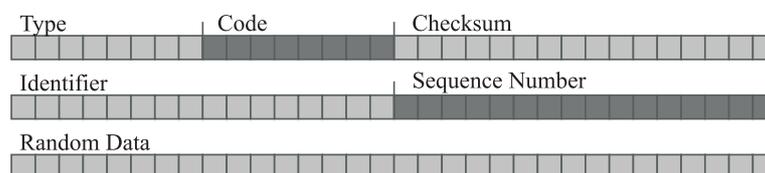


Abbildung 14: Headerdaten des ICMP Protokolls (ECHO)

Pflichtfelder eines jeden ICMP Paketes sind nur die Felder `Type`, `Code` und `Checksum`. Der übrige Paketinhalt variiert je nach Paket-Typ. Hier werden beispielhaft alle Felder der ICMP `ECHO_REQUEST` und `ECHO_REPLY` Pakete beschrieben.

Type: Typ des ICMP Paketes (`ECHO_REQUEST` = 8, `ECHO_REPLY` = 0)

Eine Übersicht über alle (ursprünglich) möglichen ICMP Pakettypen findet sich z.B. in RFC 792.

siehe auch `ICMP_*` Konstanten in `<netinet/ip_icmp.h>`

Code: Spezifiziert das ICMP Paket weiter.

Nur für `DESTINATION_UNREACHABLE`, `REDIRECT` oder `TIME_EXCEEDED` Nachrichten relevant. (Ansonsten = 0.)

siehe auch `ICMP_*` Konstanten in `<netinet/ip_icmp.h>`

Checksum: Prüfsumme nach den Regeln der Internet-Prüfsummenbildung.

Die Prüfsumme wird über das gesamte ICMP Paket gebildet, vom ICMP Typenfeld bis zum Ende der Random Data. Vor Berechnung der Prüfsumme muss das Checksum-Feld mit Nullen gefüllt werden. Näheres zur Prüfsummenberechnung sowie ein Implementierungs-Beispiel findet sich in Kapitel 2.10.

Identifier: (nur bei `ECHO_REQUEST` und `ECHO_REPLY` Paketen) Identifikationsnummer zur besseren Zuordbarkeit von `ECHO_REPLY` zu

`ECHO_REQUEST` Nachrichten. (Darf = 0 sein.)

In der Antwort (`ECHO_REPLY`) muss der gleiche Wert verwendet werden wie in der Anfrage (`ECHO_REQUEST`).

Sequence Number: (nur bei `ECHO_REQUEST` und `ECHO_REPLY` Paketen)

Kann als Zähler verwendet werden, wobei mit jedem `ECHO_REQUEST` hochgezählt wird. (Darf = 0 sein.)

In der Antwort (`ECHO_REPLY`) muss der gleiche Wert verwendet werden wie in der Anfrage (`ECHO_REQUEST`).

Random Data: (nur bei ECHO_REQUEST und ECHO_REPLY Paketen) Kann beliebige Daten in beliebiger Länge enthalten.
In der Antwort (ECHO_REPLY) müssen die gleichen Daten verwendet werden wie in der Anfrage (ECHO_REQUEST).

A.4.2 Datenstruktur

```
1 struct icmp_hdr
2 {
3     u_int8_t type;           /* message type */
4     u_int8_t code;         /* type sub-code */
5     u_int16_t checksum;
6     union
7     {
8         struct
9         {
10            u_int16_t id;
11            u_int16_t sequence;
12        } echo;             /* echo datagram */
13        u_int32_t gateway;  /* gateway address */
14        struct
15        {
16            u_int16_t __unused;
17            u_int16_t mtu;
18        } frag;             /* path mtu discovery */
19    } un;
20 };
```

Listing 21: struct icmp_hdr aus <netinet/ip_icmp.h>

A.4.3 Weitere Informationen / Quellen

- RFC 792 (ICMP): <http://tools.ietf.org/html/rfc792>
- Linux-Manpage: `man 7 icmp`

A.5 UDP Protokoll Header

A.5.1 Bit-Schema



Abbildung 15: Headerdaten des UDP Protokolls

Source Port: (optional) Beschreibt die Portnummer des sendenden Prozesses. Weist den Empfänger im Allgemeinen darauf hin, an welche Portnummer die Antwort zu richten ist. (Darf = 0 sein.)

Destination Port: Lässt den Empfänger erkennen, an welchen Prozess das Paket gerichtet ist.

Length: Länge des gesamten restlichen Paketes (in Bytes) beginnend ab dem Source Port Feld.

Checksum: (optional) Prüfsumme über den sogenannten *Pseudo-Header*, den UDP Header und die Nutzdaten zur Sicherung der Datenintegrität des gesamten Paketes. Der Pseudo-Header wird aus Teilen des IP Headers und des UDP Headers gebildet (vgl. Kapitel 2.13).

Zur Bildung der Prüfsumme werden der Pseudo-Header, der UDP Header und die Nutzdaten aneinandergereiht. Die so entstandene Bitfolge wird am Ende mit Nullen aufgefüllt, so dass die Gesamtlänge ein Vielfaches von 16 Bit ergibt. Die UDP Prüfsumme wird nach den Regeln der Internet-Prüfsummenbildung (vgl. Kapitel 2.10) gebildet.

Die Prüfsumme darf bei UDP auch Null sein. Dann wird sie jedoch als 16 Einsen kodiert (Einer-Komplement).

A.5.2 Datenstruktur

```
1 struct udphdr
2 {
3     u_int16_t source;
4     u_int16_t dest;
5     u_int16_t len;
6     u_int16_t check;
7 };
```

Listing 22: struct udphdr aus <netinet/udp.h>

A.5.3 Weitere Informationen / Quellen

- RFC 768 (UDP): <http://tools.ietf.org/html/rfc768>
- Linux-Manpage: `man 7 udp`

A.6 TCP Protokoll Header

A.6.1 Bit-Schema

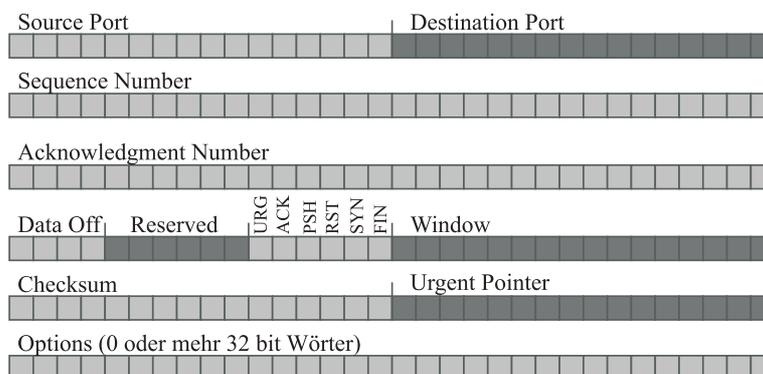


Abbildung 16: Header Daten des TCP Protokolls

Source Port: Port Nummer des Absenders (siehe UDP, hier jedoch nicht optional).

Destination Port: Port Nummer des Empfängers (siehe UDP, hier jedoch nicht optional).

Sequence Number: Sequenznummer des ersten Bytes in diesem TCP Paket.

Dient der Sortierung der ankommenden Pakete.

Wenn das *SYN-Flag* (siehe unten) gesetzt wurde, wird in diesem Feld die *Initiale Sequenznummer* übertragen.

Acknowledgement Number: Hat nur Gültigkeit, wenn das *ACK-Flag* (siehe unten) gesetzt ist.

Das Feld beinhaltet die Sequenznummer des Segments, das der Sender als nächstes erwartet.

Data Offset: Länge des TCP Headers in 32-Bit Worten. Hiermit wird deutlich ab wann die Nutzdaten beginnen.

Reserved: War ursprünglich reserviert für zukünftige Funktionen.

Das Feld wird nun gar nicht verwendet und muss daher Null sein.

Flags: (auch: Control Bits) Können jeweils 0 (nicht gesetzt, deaktiviert) oder 1 (gesetzt, aktiviert)

URG: (Urgent = Dringend) Ist dieses Feld gesetzt, so wird die Anwendung angewiesen, sofort die Daten zu bearbeiten, auf die der *Urgent Pointer* (siehe unten) zeigt.
Wird nur selten benutzt.

ACK: (Acknowledgement = Bestätigung) Wird gesetzt, wenn der Empfang bestimmter Segmente (siehe *Acknowledgement Number*) durch dieses TCP Paket bestätigt werden soll.

Bei gleichzeitig gesetztem *SYN-Flag* ist die Bestätigung im 3-Wege Handshake gemeint.

PSH: (Push) Bei gesetztem PSH-Flag, wird das Paket ohne Umweg über einen Puffer an die Applikation weitergereicht.

RST: (Reset) Wird z.B. gesetzt um unerwünschte Verbindungen abzuweisen oder um Verbindungen bei technischen Problemen abzubrechen.

SYN: (Synchronization) Wird verwendet um einen Verbindungsaufbau zu initialisieren (3-Wege Handshake) bzw. die Initialisierung zu bestätigen (mit gleichzeitig gesetztem *ACK-Flag*).
Dient der Synchronisierung von Sequenznummern.

FIN: Wird gesetzt, um TCP Verbindungen zu Beenden.

Window: Die Größe des Empfangsfensters.

Anzahl der Bytes (ab dem Byte mit der Nummer im Feld *Acknowledgement Number*), die der Sender dieses Paketes bereit ist zu empfangen.

Checksum: (erforderlich) Prüfsumme über den sogenannten *Pseudo-Header*, den TCP Header und die Nutzdaten zur Sicherung der Datenintegrität des gesamten Pakets. Der Pseudo-Header wird aus Teilen des IP Headers und des TCP Headers gebildet (vgl. Kapitel 2.13).

Zur Bildung der Prüfsumme werden der Pseudo-Header, der TCP Header und die Nutzdaten aneinandergereiht. Die so entstandene Bitfolge wird am Ende mit Nullen aufgefüllt, so dass die Gesamtlänge ein Vielfaches von 16 Bit ergibt. Die TCP Prüfsumme wird nach den Regeln der Internet-Prüfsummenbildung (vgl. Kapitel 2.10) gebildet.

Urgent Pointer: Nur gültig, wenn das *URG-Flag* gesetzt ist.

Gibt zusammen mit der Sequenznummer die genaue position der dringenden Daten im Datenstrom an.

Options: Ermöglicht das Unterbringen weiterer TCP-Optionen.

Die Länge des Options-Feldes muss ein Vielfaches von 32-Bit sein (ggf. müssen Nullen am Ende aufgefüllt werden).

A.6.2 Datenstruktur

```
1 struct tcphdr
2 {
3     u_int16_t source;
4     u_int16_t dest;
5     u_int32_t seq;
6     u_int32_t ack_seq;
7 # if __BYTE_ORDER == __LITTLE_ENDIAN
8     u_int16_t res1:4;
9     u_int16_t doff:4;
10    u_int16_t fin:1;
11    u_int16_t syn:1;
12    u_int16_t rst:1;
13    u_int16_t psh:1;
14    u_int16_t ack:1;
15    u_int16_t urg:1;
16    u_int16_t res2:2;
17 # elif __BYTE_ORDER == __BIG_ENDIAN
18    u_int16_t doff:4;
19    u_int16_t res1:4;
20    u_int16_t res2:2;
21    u_int16_t urg:1;
22    u_int16_t ack:1;
23    u_int16_t psh:1;
24    u_int16_t rst:1;
25    u_int16_t syn:1;
26    u_int16_t fin:1;
27 # else
28 #   error "Adjust your <bits/endian.h> defines"
29 # endif
30    u_int16_t window;
31    u_int16_t check;
32    u_int16_t urg_ptr;
33 };
```

Listing 23: struct tcphdr aus <netinet/tcp.h>

A.6.3 Weitere Informationen / Quellen

- RFC 793 (TCP): <http://tools.ietf.org/html/rfc793>
- Linux-Manpage: `man 7 tcp`

B Programmbeispiel show_packet_binary.c

```
1  /***
2  *   Author: Andre Volk, 2008
3  *
4  *   Dieses Programm erstellt einen RAW Socket, liest das
5  *   erstbeste ICMP Paket auf einem beliebigen Interface
6  *   und gibt dessen Inhalt (ohne Link-Layer Header) in
7  *   binaerer Darstellung aus.
8  *
9  *   Programm kompilieren mit:
10 *
11 *       gcc show_packet_binary.c -o show_packet_binary
12 *
13 ***/
14
15
16 /*** show_packet_binary.c ***/
17
18 #include <stdio.h>           // Standard C Funktionen
19 #include <stdlib.h>         // Std. Fkt. wie exit() und malloc()
20 #include <unistd.h>         // Std. Fkt. wie getuid() und read()
21 #include <string.h>         // String u Memory Fkt (strcmp(), memset())
22
23 #include <sys/ioctl.h>      // Ankunftszeit des Pakets
24 #include <time.h>
25
26 #include <netinet/in.h>     // Socket Fkt. (incl. <sys/socket.h>)
27 #include <netinet/ether.h>  // Ethernet Fkt. u Konst. (z.B. ETH_P_ALL)
28
29
30 void print_binary_byte (unsigned char byte) {
31
32     char binary_byte[10];
33     int  i;
34
35     binary_byte[8] = ' '; // Leerraum zw den Bytes
36     binary_byte[9] = '\0'; // Null Terminierung
37
38     //Umwandlung in binaere Darstellung
39     for (i = 0; i < 8; i++) {
40         if ( byte&(1<<i)) binary_byte[7-i] = '1';
41         else binary_byte[7-i] = '0';
42     }
43
44     printf (binary_byte);
45
46 }
47
```

```

48
49
50 int main(int argc, char *argv[])
51 {
52     struct timeval tv;
53     time_t curtime;
54     char time_buffer[30];
55
56     int s, bytes, microseconds;
57     unsigned char packet[ETHERMTU]; // ETHERMTU = 1500
58
59
60     /* root-Rechte pruefen */
61     if(getuid() != 0) {
62         printf("\nSie muessen als root angemeldet sein!\n");
63         exit(1);
64     }
65
66
67     /**
68     * Hier wird der SOCK_RAW Socket fuer ICMP Pakete angelegt.
69     * Um weitere Protokolle auszulesen muessen die Parameter
70     * entsprechend angepasst werden.
71     ***/
72
73     s = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
74
75     /**
76     * Durch Aktivierung der naechsten Zeile wird statt eines
77     * normalen RAW Sockets ein PACKET Socket verwendet.
78     * Dadurch werden auch die Link Layer Informationen sichtbar.
79     ***/
80
81     // s = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
82
83
84
85     if (s == -1)
86     {
87         perror("socket() failed");
88         return 1;
89     }
90
91
92
93
94
95
96

```

```

97  /**
98  *   Durch Aktivierung der While-Schleife werden statt nur
99  *   einem Paket alle folgenden Pakete eingelesen.
100  */
101
102  //while (1) {
103
104
105      /**
106      *   Mit read werden Daten vom Socket gelesen.
107      *
108      *   bytes           : Zahl der gelesenen Bytes
109      *   s               : Socket, von dem gelesen werden soll
110      *   packet         : Speicherplatz in den die gelesenen
111      *                   Daten geschrieben werden sollen
112      *   sizeof(packet) : Laenge des Speicherbereichs, in den
113      *                   geschrieben werden darf
114      *   0               : Keine besonderen Flags
115      */
116
117      bytes = read(s, packet, sizeof(packet));
118
119
120      /**
121      *   Echte Ankunftszeit des Pakets ausgeben
122      */
123
124      ioctl(s, SIOCGSTAMP, &tv);
125      curtime=tv.tv_sec;
126      microseconds=(int)tv.tv_usec;
127      strftime (time_buffer, 30, "%Y-%m-%d, %T", localtime(&curtime));
128      printf("\nZeit: %s.%06d\n", time_buffer, microseconds);
129
130
131      /**
132      *   Durchlaufe die gelesenen Daten Byte fuer Byte und
133      *   gebe die Daten binaer als Nullen und Einsen aus.
134      */
135
136      printf("\nBinaere Darstellung:\n\n");
137
138      int j, k=0;
139
140      for (j = 0; j < bytes; j++){
141
142          k++; // Zaehlt die Bytes in einer Zeile
143
144          // Gibt das Byte in 8-stelliger Binaerdarstellung aus
145          print_binary_byte(packet[j]);

```

```
146
147         // Immer 4 Bytes pro Zeile
148         if ((k % 4) == 0) { printf ("\n"); }
149
150     }
151
152     printf("\n");
153
154 //}
155
156
157     return 0;
158 }
159
160 /*** Ende von show_packet_binary.c ***/
```

Listing 24: Beispielprogramm: show_packet_binary.c

C Programmbeispiel syn_packet_gen.c

```
1  /**
2  *   Author: Andre Volk, 2008
3  *
4  *   Dieses Programm generiert ein TCP SYN Paket
5  *   auf Basis eines RAW Sockets mit gesetztem
6  *   IP_HDRINCL. Der IP Header wird vollstaendig
7  *   von Hand generiert.
8  *
9  *   Basiert auf syn_flood.c von Felix Opatz
10 *   (www.zotteljedi.de)
11 *
12 *   Programm kompilieren mit:
13 *
14 *   gcc syn_packet_gen.c -o syn_packet_gen
15 *
16 ***/
17
18 /** syn_packet_gen.c ***/
19
20 #include <sys/types.h>
21 #include <stdio.h>
22 #include <string.h>
23 #include <stdlib.h>
24 #include <unistd.h>
25 #include <arpa/inet.h>
26 #include <netinet/in.h>
27 #include <netinet/ip.h>
28 #include <netinet/tcp.h>
29
30
31
32
33 unsigned short comp_chksum(unsigned short *addr, int len)
34 {
35
36     /**
37      *   Quelle: RFC 1071
38      *   Berechnet die Internet-Checksumme
39      *   Gueltig fuer den IP-, ICMP-, TCP- oder UDP-Header
40      *
41      *   *addr : Zeiger auf den Anfang der Daten
42      *           (Checksummenfeld muss Null sein)
43      *   len   : Laenge der Daten (in Bytes)
44      *
45      *   Rueckgabewert : Checksumme in Network Byte Order
46      */
47
```

```

48     long sum = 0;
49
50     while( len > 1 ) {
51         sum += *(addr++);
52         len -= 2;
53     }
54
55     if( len > 0 )
56         sum += * addr;
57
58     while (sum>>16)
59         sum = ((sum & 0xffff) + (sum >> 16));
60
61     sum = ~sum;
62
63     return ((u_short) sum);
64 }
65
66
67
68
69 int main(int argc, char *argv[])
70 {
71
72     // Hilfs-Datenstrukturen zur Bildung der Pruefsumme
73
74     // TCP-/UDP-Pseudoheader
75     struct pseudohdr
76     {
77         u_int32_t src_addr;
78         u_int32_t dst_addr;
79         u_int8_t padding;
80         u_int8_t proto;
81         u_int16_t length;
82     };
83
84
85
86     // Datenstruktur zur Zusammenfassung aller Daten
87     // die zur Generierung der Pruefsumme notwendig sind:
88     // Pseudoheader, TCP- bzw. UDP-Header und Nutzlast
89     struct data_4_checksum
90     {
91         struct pseudohdr pshd;
92         struct tcphdr tcphd;
93         char payload[1024];
94     };
95
96

```

```

97     int                s, bytes, on = 1;
98     char               buffer[1024];
99     struct iphdr       *ip;
100    struct tcphdr      *tcp;
101    struct sockaddr_in  to;
102    struct pseudohdr    pseudoheader;
103    struct data_4_checksum tcp_chk_construct;
104
105
106    // Anzahl der Programm-Parameter pruefen
107    if (argc != 5)
108    {
109        fprintf(stderr, "Usage: %s <src-addr> ", argv[0]);
110        fprintf(stderr, "<src-port> <dest-addr> <dest-port>\n");
111        return 1;
112    }
113
114
115    // RAW Socket aufbauen
116    s = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
117    if (s == -1)
118    {
119        perror("socket() failed");
120        return 1;
121    }
122
123
124    // Socket Option IP_HDRINCL setzen, damit der IP Header
125    // manuell generiert werden kann.
126    if (setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) == -1)
127    {
128        perror("setsockopt() failed");
129        return 2;
130    }
131
132
133    // Vorbereiten des Schreibpuffers durch Type-Casting
134    ip = (struct iphdr*) buffer;
135    tcp = (struct tcphdr*) (buffer + sizeof(struct iphdr));
136
137    int iphdrlen = sizeof(struct iphdr);
138    int tcphdrln = sizeof(struct tcphdr);
139    int datalen = 0;
140
141
142    // IP Header fuellen
143    memset(ip, 0, iphdrlen);
144    ip->version = 4;
145    ip->ihl = 5;

```

```

146 ip->tot_len = htons(iphdrlen + tcphdrLen);
147 // ID automatisch vom Kernel beziehen.
148 // Alternativ z.B.: ip->id = htons(1234);
149 ip->id = 0;
150 ip->tTL = 255;
151 ip->protocol = IPPROTO_TCP;
152 // Absenderadresse kann automatisch vom
153 // Kernel gesetzt werden: ip->saddr = INADDR_ANY;
154 // Vorsicht! TCP Pruefsumme wird dadurch evtl. falsch!
155 ip->saddr = inet_addr(argv[1]);
156 ip->daddr = inet_addr(argv[3]);
157 // IP Pruefsumme wird bei normalen RAW Sockets
158 // immer automatisch vom Kernel berechnet
159 ip->check = 0;
160
161
162 // TCP Header fuellen
163 memset(tcp, 0, tcphdrLen);
164 tcp->source = htons(atol(argv[2]));
165 tcp->dest = htons(atol(argv[4]));
166 tcp->seq = random();
167 tcp->doff = 5;
168 tcp->syn = 1;
169 tcp->window = htons(65535);
170
171
172 // Pseudoheader erstellen
173 pseudoheader.src_addr = ip->saddr;
174 pseudoheader.dst_addr = ip->daddr;
175 pseudoheader.padding = 0;
176 pseudoheader.proto = ip->protocol;
177 pseudoheader.length = htons(tcphdrLen+datalen);
178
179 // Pseudoheader, TCP Header und Nutzlast kombinieren
180 tcp_chk_construct.pshd = pseudoheader;
181 tcp_chk_construct.tcphd = *tcp;
182 //tcp_chk_construct.payload muesste per memcpy
183 //gefuellt werden wenn Nutzlast vorhanden waere.
184
185
186 // Pruefsumme ueber Pseudoheader,
187 // TCP Header und Nutzlast berechnen
188 int checksum = comp_chksum((unsigned short*)&tcp_chk_construct,
189                             sizeof(struct pseudohdr)+tcphdrLen+datalen);
190
191 // TCP Pruefsumme im Header eintragen
192 tcp->check = checksum;
193
194

```

```

195     printf ("TCP Pruefsumme: %i\n", checksum);
196     printf ("Ziel   : %s:%i\n", argv[3], ntohs(tcp->dest));
197     printf ("Quelle: %s:%i\n", argv[1], ntohs(tcp->source));
198
199
200     // Socket Adress-Datenstruktur fuer sendto()
201     to.sin_addr.s_addr = ip->daddr;
202     to.sin_family      = AF_INET;
203     to.sin_port        = tcp->dest;
204
205
206     // Paket versenden
207     bytes = sendto(s, buffer, ntohs(ip->tot_len), 0,
208                  (struct sockaddr*)&to, sizeof(to));
209
210     if (bytes == -1)
211     {
212         perror("sendto() failed");
213         return 1;
214     }
215
216     return 0;
217 }
218
219 /*** Ende von syn_packet_gen.c ***

```

Listing 25: Beispielprogramm: syn_packet_gen.c

D Programmbeispiel show_packet_verbose.c

```
1  /***
2  *   Author: Andre Volk, 2008
3  *
4  *   Dieses Programm erstellt einen PACKET Socket, der
5  *   ein Netzwerkpaket einliest.
6  *   Mit Hilfe der Datenstrukturen fuer die verschiedenen
7  *   Packetheader werden wichtige Informationen aus dem
8  *   Paket ausgelesen.
9  *
10 *   Programm kompilieren mit:
11 *
12 *       gcc show_packet_verbose.c -o show_packet_verbose
13 *
14 ***/
15
16 /*** show_packet_verbose.c ***/
17
18 #include <stdio.h>           // Standard C Funktionen
19 #include <stdlib.h>         // Std. Fkt. wie exit() und malloc()
20 #include <unistd.h>         // Std. Fkt. wie getuid() und read()
21 #include <string.h>         // String/Memory Fkt (strcmp(), memset())
22 #include <ctype.h>
23
24 #include <sys/ioctl.h>      // Ankunftszeit des Pakets
25 #include <time.h>
26
27 #include <netinet/in.h>     // Socket Fkt. (incl. <sys/socket.h>)
28 #include <netinet/ether.h> // Ethernet Fkt. u. Konst.
29                             // (z.B. ETH_P_ALL oder ether_ntoa())
30 #include <arpa/inet.h>     // Datenstrukt. u. Fkt fuer IP Adressen
31                             // (z.B. struct in_addr od. inet_ntoa())
32
33
34 #include <net/ethernet.h>   // Ethernet Header Struktur
35 #include <net/if_arp.h>     // ARP Header Struktur
36 #include <netinet/ip.h>     // IPv4 Header Struktur
37 #include <netinet/ip_icmp.h> // ICMPv4 Header Struktur
38 #include <netinet/udp.h>    // UDP Header Struktur
39 #include <netinet/tcp.h>    // TCP Header Struktur
40
41
42
43 void printEthernetInformation (struct ether_header *eth,
44                               int hdr_len, int pkt_len)
45 {
46     printf("LinkLayer Protokoll: Ethernet\n");
47     printf("    MTU           : %i\n",ETHERMTU);
```

```

48     printf("    Empfangen      : %i Bytes\n", pkt_len);
49     printf("    Headerlaenge   : %i Bytes\n", hdr_len);
50
51
52
53     struct ether_addr *macaddr;
54     macaddr = (struct ether_addr *)eth->ether_shost;
55     printf("    Absender MAC    : %s\n", ether_ntoa(macaddr));
56     macaddr = (struct ether_addr *)eth->ether_dhost;
57     printf("    Empfaenger MAC  : %s\n", ether_ntoa(macaddr));
58     printf("    Ether-Type     : %04X\n\n", ntohs(eth->ether_type));
59 }
60
61
62 void printARPInformation(struct arphdr *arp, int hdr_len)
63 {
64     char *arp_operation;
65
66     printf("LinkLayer Protokoll: ARP\n");
67     printf("    Headerlaenge   : %i Bytes\n", hdr_len);
68
69     int arpop = ntohs(arp->ar_op);
70     switch(arpop)
71     {
72         case ARPOP_REQUEST: arp_operation = "REQUEST (WHO-HAS)";
73                             break;
74         case ARPOP_REPLY   : arp_operation = "REPLY (IS-AT)";
75                             break;
76         default           : arp_operation = "Unbekannt";
77     }
78
79     printf("    Operation      : (%i) %s\n", arpop, arp_operation);
80 }
81
82
83
84 void printIPv4Information (struct iphdr *ip, int hdr_len)
85 {
86     printf("NetworkLayer Protokoll: IPv4\n");
87     printf("    Headerlaenge   : %i Bytes\n", hdr_len);
88
89     printf("    Type of Service: %i\n", ip->tos);
90     printf("    Total Length   : %i Bytes\n", ntohs(ip->tot_len));
91     printf("    ID             : %i\n", ntohs(ip->id));
92     printf("    Time To Live   : %i\n", ip->ttl);
93     printf("    Checksum       : %04X\n", ntohs(ip->check));
94
95     struct in_addr ipaddr;
96     ipaddr.s_addr = ip->saddr;

```

```

97     printf("    Absender      : %s\n", inet_ntoa(ipaddr));
98
99     ipaddr.s_addr = ip->daddr;
100    printf("    Empfaenger    : %s\n", inet_ntoa(ipaddr));
101 }
102
103
104 void printICMPInformation(struct icmp_hdr *icmp, int hdr_len)
105 {
106     printf("TransportLayer Protokoll: ICMP\n");
107     printf("    Headerlaenge    : %i Bytes\n", hdr_len);
108     printf("    Type            : ");
109
110     int icmp_code = (int)icmp->type;
111
112     switch(icmp_code)
113     {
114         case ICMP_ECHOREPLY:
115             printf ("ECHO REPLY (%i)", icmp_code);
116             break;
117
118         case ICMP_ECHO:
119             printf ("ECHO REQUEST (%i)", icmp_code);
120             break;
121
122         case ICMP_DEST_UNREACH:
123             printf ("DESTINATION UNREACHABLE (%i)", icmp_code);
124             break;
125
126         case ICMP_REDIRECT:
127             printf ("REDIRECT (%i)", icmp_code);
128             break;
129
130         case ICMP_TIME_EXCEEDED:
131             printf ("TIME EXCEEDED (%i)", icmp_code);
132             break;
133
134         default:
135             printf ("SONSTIGE NACHRICHT (%i)", icmp_code);
136     }
137
138     printf("\n");
139
140 }
141
142
143
144
145

```

```

146 int printTCPInformation(struct tcphdr *tcp, int hdr_len)
147 {
148     /**
149     * Gibt verschiedene Daten des TCP Headers aus.
150     * Rueckgabewert 1: Daten in der Payload sind ASCII.
151     * Rueckgabewert -1: Daten in der Payload sind nicht ASCII.
152     */
153
154     printf("TransportLayer Protokoll: TCP\n");
155     printf("    Headerlaenge    : %i Bytes\n", hdr_len);
156
157     printf("    Quellport        : %i\n", ntohs(tcp->source));
158     printf("    Zielport         : %i\n", ntohs(tcp->dest));
159
160     int ret = -1;
161
162     // Bei FTP, Telnet und HTTP werden ASCII Daten erwartet
163
164     if ((ntohs(tcp->dest) == 21) | (ntohs(tcp->source) == 21) |
165         (ntohs(tcp->dest) == 23) | (ntohs(tcp->source) == 23) |
166         (ntohs(tcp->dest) == 80) | (ntohs(tcp->source) == 80))
167         { ret = 1; }
168
169     return ret;
170 }
171
172
173 int printUDPInformation(struct udphdr *udp, int hdr_len)
174 {
175
176     /**
177     * Gibt verschiedene Daten des UDP Headers aus.
178     * Rueckgabewert 1: Daten in der Payload sind ASCII.
179     * Rueckgabewert -1: Daten in der Payload sind nicht ASCII.
180     */
181
182     printf("TransportLayer Protokoll: UDP\n");
183     printf("    Headerlaenge    : %i Bytes\n", hdr_len);
184
185     printf("    Quellport        : %i\n", ntohs(udp->source));
186     printf("    Zielport         : %i\n", ntohs(udp->dest));
187     printf("    Laenge          : %i Bytes\n", ntohs(udp->len));
188
189     int ret = -1;
190
191     return ret;
192 }
193
194

```

```

195 void printPayloadInformation(char *payloadData, int length, int readable)
196 {
197     /**
198     *  Gibt Informationen zur Payload aus.
199     *  readable ist 1, wenn in der Payload vermutlich
200     *  ASCII Daten hinterlegt sind.
201     ***/
202
203     printf("ApplicationLayer Daten (Payload):\n");
204     printf("    Laenge          : %i Bytes\n", length);
205
206     if (readable == 1){
207     printf("    Daten (ASCII)   : \n\n");
208
209         // Gebe alle druckbaren Zeichen aus, sonst '#'
210         int i;
211         for (i=0; i<length; i++){
212             if (((payloadData[i] >= 32) && (payloadData[i] <= 125)) |
213                 (payloadData[i] == '\n') | (payloadData[i] == '\r')) {
214                 putchar(payloadData[i]);
215             } else {putchar('#');}
216         }
217
218     printf("\n");
219     }
220 }
221
222
223 void print_binary_byte (unsigned char byte) {
224
225     char binary_byte[10];
226     int i;
227
228     binary_byte[8] = ' '; // Leerraum zw den Bytes
229     binary_byte[9] = '\0'; // Null Terminierung
230
231     //Umwandlung in binaere Darstellung
232     for (i = 0; i < 8; i++) {
233         if ( byte&(1<<i)) binary_byte[7-i] = '1';
234         else binary_byte[7-i] = '0';
235     }
236
237     printf (binary_byte);
238
239 }
240
241
242
243

```

```

244 int main(int argc, char *argv[])
245 {
246
247     int    s, bytes;
248     char   packet[ETHERMTU];
249
250     //      Zeiger fuer das Type-Casting
251     struct ether_header    *eth_hdr;
252     struct arphdr          *arp_hdr;
253     struct iphdr           *ip_hdr;
254     struct icmp_hdr       *icmp_hdr;
255     struct udphdr         *udp_hdr;
256     struct tcphdr         *tcp_hdr;
257
258     char   *payload;
259     int    payloadReadable = -1, payloadLgt=0;;
260
261     //      Namen der eingesetzten Protokolle
262     char   *linkLayerProt , *netLayerProt, *transLayerProt;
263     int    linkLayerLgt=0, netLayerLgt=0, transLayerLgt=0;
264     int    linkLayerPos=0, netLayerPos=0, transLayerPos=0;
265
266     //      Variablen fuer Zeitangaben
267     time_t curtime;
268     struct timeval tv;
269     int    milliseconds;
270     char   time_buffer[30];
271
272
273     /**
274     *   Hier wird der PACKET Socket angelegt.
275     *   ETH_P_ALL sieht alle Pakete, die empfangen od gesendet werden.
276     *   ETH_P_IP  sieht nur IP Pakete, die empfangen werden.
277     *   ETH_P_IP  sieht nur ARP Pakete, die empfangen werden.
278     */
279
280     s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
281
282     if (s == -1){
283         perror("socket() failed");
284         return 1;
285     }
286
287
288
289
290
291
292

```

```

293 //while (1) {
294
295
296     /**
297     * Mit read werden Daten vom Socket gelesen.
298     ***/
299
300     bytes = read(s, packet, sizeof(packet));
301
302     if (bytes == -1){
303         perror("read() failed");
304         return 1;
305     }
306
307
308     /**
309     * Ankunftszeit des Pakets ausgeben.
310     ***/
311
312     ioctl(s, SIOCGSTAMP, &tv);
313     curtime=tv.tv_sec;
314     milliseconds=(int)tv.tv_usec;
315     strftime (time_buffer, 30, "%Y-%m-%d, %T", localtime(&curtime));
316     printf("\nZeit: %s.%06d\n", time_buffer, milliseconds);
317
318
319     /**
320     * Informationen zum Link Layer extrahieren.
321     * Hier: Festlegung auf Ethernet.
322     ***/
323
324     // Der LinkLayer beginnt a.d. Zeigeradr. "packet"
325     linkLayerPos = (int)packet;
326     linkLayerProt = "Ethernet";
327     linkLayerLgt = sizeof(struct ether_header);
328
329     // Type-Casting:
330     eth_hdr = (struct ether_header *)linkLayerPos;
331
332     //Pos. des NetworkLayers im Speicher berechnen
333     netLayerPos = (int)packet + linkLayerLgt;
334
335     printf("\n");
336     printEthernetInformation(eth_hdr, linkLayerLgt, bytes);
337
338
339
340
341

```

```

342  /**
343   * Informationen zum Network Layer auswerten
344   ***/
345
346   if (ntohs(eth_hdr->ether_type) == ETHERTYPE_IP)
347   {
348       netLayerProt = "IPv4";
349       ip_hdr = (struct iphdr *)netLayerPos;
350
351       netLayerLgt = ((int)ip_hdr->ihl) * 4;
352       transLayerPos = (int)packet+linkLayerLgt+netLayerLgt;
353
354       printf("\n");
355       printIPv4Information(ip_hdr, netLayerLgt);
356
357       switch (ip_hdr->protocol)
358       {
359           case IPPROTO_ICMP:
360               transLayerProt = "ICMPv4";
361               transLayerLgt = sizeof(struct icmphdr);
362
363               // Type-Casting:
364               icmp_hdr = (struct icmphdr *)transLayerPos;
365
366               printf("\n");
367               printICMPInformation(icmp_hdr, transLayerLgt);
368               break;
369
370           case IPPROTO_TCP:
371               transLayerProt = "TCP";
372
373               // Type-Casting:
374               tcp_hdr = (struct tcphdr *)transLayerPos;
375
376               // Achtung: TCP hat eine variable Headergroesse
377               // daher darf nicht sizeof(struct tcphdr)
378               // verwendet werden!!
379               transLayerLgt = ((int)tcp_hdr->doff) * 4;
380
381               printf("\n");
382               payloadReadable =
383                   printTCPInformation(tcp_hdr, transLayerLgt);
384               break;
385
386           case IPPROTO_UDP:
387               transLayerProt = "UDP";
388               transLayerLgt = sizeof(struct udphdr);
389
390

```

```

391         // Type-Casting:
392         udp_hdr = (struct udphdr *)transLayerPos;
393
394         printf("\n");
395         payloadReadable =
396             printUDPInformation(udp_hdr, transLayerLgt);
397         break;
398
399     default:
400         transLayerProt = "Unbekannt";
401         transLayerLgt = 0;
402         printf("\n");
403         printf("TransportLayer Protokoll: Unbekannt\n");
404         printf("    Protokollnr: %i\n", ip_hdr->protocol);
405
406     }
407 }
408
409 else if (ntohs(eth_hdr->ether_type) == ETHERTYPE_ARP)
410 {
411     netLayerProt = "ARP";
412     netLayerLgt = sizeof(struct arphdr);
413
414     arp_hdr = (struct arphdr *)netLayerPos;
415
416     transLayerPos = (int)packet+linkLayerLgt+netLayerLgt;
417     transLayerProt = "Kein Transportprotokoll";
418     transLayerLgt = 0;
419
420     printf("\n");
421     printARPInformation(arp_hdr, netLayerLgt);
422 }
423
424 else {
425
426     printf("\n");
427     printf ("Unbekanntes NetworkLayer Protokoll.\n");
428     printf ("Protokollnr: %i\n",ntohs(eth_hdr->ether_type));
429     exit(1);
430
431 }
432
433
434 /**
435  * Die Payload:
436  /**/
437
438 payload = packet + linkLayerLgt + netLayerLgt + transLayerLgt;
439 payloadLgt = bytes - (linkLayerLgt + netLayerLgt + transLayerLgt);

```

```

440     printf("\n");
441     printPayloadInformation(payload, payloadLgt, payloadReadable);
442
443     /**
444     * Durchlaufe das Paket Byte fuer Byte und
445     * gebe die Daten binaer als Nullen und Einsen aus.
446     ***/
447
448     printf("\n");
449     printf("Binaere Darstellung:\n\n");
450     printf("%s (%i):\n", linkLayerProt, linkLayerLgt);
451
452     int j,k;
453
454     k = 0;
455     for (j = 0; j < bytes; j++){
456
457         k++; // Zaehlt die Pakete in einer Zeile
458
459         if (j == linkLayerLgt) {
460             printf("\n\n%s (%i):\n", netLayerProt, netLayerLgt);
461             k=1;
462         }
463
464         if (j == linkLayerLgt+netLayerLgt) {
465             printf("\n\n%s (%i):\n", transLayerProt, transLayerLgt);
466             k=1;
467         }
468
469         if (j == linkLayerLgt+netLayerLgt+transLayerLgt) {
470             printf("\n\nNutzlast (%i):\n", payloadLgt);
471             k=1;
472         }
473
474         print_binary_byte(packet[j]);
475
476         if ((k % 4) == 0) { printf ("\n"); }
477     }
478
479     printf("\n===== \n\n");
480
481 // }
482
483     return 0;
484 }
485 /**/ Ende von show_packet_verbose.c ***/

```

Listing 26: Beispielprogramm: show_packet_verbose.c

E Programmbeispiel arp_packet_gen.c

```
1  /***
2  *   Author: Andre Volk, 2008
3  *
4  *   Dieses Programm generiert ein ARP REPLY Paket
5  *   auf Basis eines PACKET Sockets.
6  *
7  *   Programm kompilieren mit:
8  *
9  *   gcc arp_packet_gen.c -o arp_packet_gen
10 *
11 ***/
12
13 /*** arp_packet_gen.c ***/
14
15 #include <stdio.h>           // Standard C Funktionen
16 #include <stdlib.h>         // Std. Fkt. wie exit() und malloc()
17 #include <unistd.h>         // Std. Fkt. wie getuid() und read()
18 #include <string.h>         // String u Memory Fkt (strcmp(), memset())
19 #include <errno.h>          // Detailliertere Fehlermeldungen
20
21 #include <sys/socket.h>     // Socket Fkt wie socket(), bind(), listen()
22 #include <arpa/inet.h>      // Wichtige Funktionen wie htons() etc.
23 #include <net/ethernet.h>   // struct ether_header, ETHER_ADDR_LEN
24 #include <netinet/ether.h> // ether_aton()
25 #include <net/if_arp.h>    // struct arphdr, ARPOP_REPLY, ARPHRD_ETHER
26
27 // (siehe "man 7 packet")
28 #include <linux/if_packet.h> // struct sockaddr_ll
29
30 // Netzwerk Interface: (siehe "man 7 netdevice")
31 #include <sys/ioctl.h>      // ioctl(): Abfrage des Netw Interf Treibers
32 #include <net/if.h>        // struct ifreq
33
34
35 // Definition einer Datenstruktur fuer ARP Header
36 // (struct arphdr in <net/if_arp.h> ist nur bis ar_op definiert)
37 struct arp_header {
38     unsigned short int ar_hrd; // Format of hardware address
39     unsigned short int ar_pro; // Format of protocol address
40     unsigned char   ar_hln;    // Length of hardware address
41     unsigned char   ar_pln;    // Length of protocol address
42     unsigned short int ar_op;  // ARP opcode (command).
43     unsigned char   ar_sha[6]; // Sender hardware addr (MAC)
44     unsigned char   ar_sip[4]; // Sender IP address
45     unsigned char   ar_tha[6]; // Target hardware addr (MAC)
46     unsigned char   ar_tip[4]; // Target IP address
47 };
```

```

48
49
50 int main(int argc, char *argv[])
51 {
52     int s, uid;
53
54     unsigned int packetsize = sizeof(struct ether_header) +
55                               sizeof(struct arp_header);
56
57     unsigned char packet[packetsize];
58
59     struct ether_header *eth_hdr;
60     struct arp_header  *arp_hdr;
61
62     // Socket Adress-Datenstruktur fuer PACKET Sockets
63     struct sockaddr_ll To_sock_addr;
64
65     // Informationen zum Network Interface
66     struct ifreq ifr;
67
68
69     // Adressen festlegen...
70
71     // Sender IP Adresse
72     char *sip = "192.168.1.69";
73
74     // Sender MAC Adresse
75     char *smac = "12:23:34:45:56:67";
76
77     // Empfaenger IP Adresse (hier: Broadcast)
78     char *dip = "255.255.255.255";
79
80     // Empfaenger MAC Adresse (hier: Broadcast)
81     char *dmac = "FF:FF:FF:FF:FF:FF";
82
83     // Name der Netzwerkschnittstelle
84     char *dev = "eth1";
85
86
87     // root-Rechte vorhanden?
88     uid = getuid();
89     if(uid != 0)
90     {
91         printf("You must be root!\n");
92         exit(1);
93     }
94
95
96

```

```

97 // Packet Buffer initialisieren
98 memset(packet,0,packetsize);
99
100 // Type-Casting
101 eth_hdr = (struct ether_header *)packet;
102 arp_hdr = (struct arp_header *)(packet + sizeof(struct ether_header));
103
104
105 // Ethernet Header fuellen
106
107 // Destination MAC
108 memcpy(eth_hdr->ether_dhost,(u_char *)ether_aton(dmac),ETHER_ADDR_LEN);
109
110 // Source MAC
111 memcpy(eth_hdr->ether_shost,(u_char *)ether_aton(smac),ETHER_ADDR_LEN);
112 eth_hdr->ether_type = htons(ETHERTYPE_ARP); // ARP Protokoll
113
114
115 // ARP Header Optionen
116 arp_hdr->ar_hrd = htons(ARPHRD_ETHER); // Hardware Addr Typ
117 arp_hdr->ar_pro = htons(ETH_P_IP); // Protokoll Addr Typ
118 arp_hdr->ar_hln = ETHER_ADDR_LEN; // Hardware Addr Lg
119 arp_hdr->ar_pln = 4; // Protokoll Addr Lg
120 arp_hdr->ar_op = htons(ARPOP_REPLY); // ARP Operation Typ
121
122
123 // Sender MAC im ARP Header eintragen
124 memcpy(arp_hdr->ar_sha, (u_char *)ether_aton(smac), ETHER_ADDR_LEN);
125
126 // Sender IP im ARP Header eintragen
127 *(u_long *)arp_hdr->ar_sip = inet_addr(sip);
128
129 // Empfaenger MAC im ARP Header eintragen
130 memcpy(arp_hdr->ar_tha, (u_char *)ether_aton(dmac), ETHER_ADDR_LEN);
131
132 // Empfaenger IP im ARP Header eintragen
133 *(u_long *)arp_hdr->ar_tip = inet_addr(dip);
134
135
136 // Erstelle einen PACKET Socket
137 if((s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ARP))) == -1 )
138 {
139     perror("Socket() failed.");
140     exit(1);
141 }
142
143
144
145

```

```

146 // Indexnummer des Network Interfaces erfragen
147 strncpy(ifr.ifr_name, dev, sizeof(ifr.ifr_name));
148
149 if (ioctl(s, SIOCGIFINDEX, &ifr) != 0)
150 {
151     perror("ioctl(SIOCGIFINDEX) failed");
152     exit(1);
153 }
154
155 // Socket-Adresse definieren
156 memset(&To_sock_addr, 0, sizeof(struct sockaddr_ll));
157 To_sock_addr.sll_ifindex = ifr.ifr_ifindex;
158
159 // Paket versenden
160 printf ("Sending ARP REPLY on Interface %s (%i)...\n",
161         dev, ifr.ifr_ifindex);
162
163
164 if ((sendto(s,packet,packetsize,0,(struct sockaddr*)&To_sock_addr,
165            sizeof(struct sockaddr_ll)))==-1)
166 {
167     perror("Sendto() failed.");
168     exit(1);
169 }
170
171 printf ("Done.\n");
172
173 return 0;
174 }
175
176 /*** Ende von arp_packet_gen.c ***/

```

Listing 27: Beispielprogramm: arp_packet_gen.c

F Programmbeispiel icmp_packet_gen.c

```
1  /***
2  *   Author: Andre Volk, 2008
3  *
4  *   Dieses Programm generiert ein ICMP REQUEST Paket
5  *   auf Basis eines PACKET Sockets.
6  *
7  *   Programm kompilieren mit:
8  *
9  *   gcc icmp_packet_gen.c -o icmp_packet_gen
10 *
11 ***/
12
13
14
15 /*** icmp_packet_gen.c ***/
16
17
18 #include <stdio.h>           // Standard C Funktionen
19 #include <stdlib.h>         // Std. Fkt. wie exit() und malloc()
20 #include <unistd.h>         // Std. Fkt. wie getuid() und read()
21 #include <string.h>         // String u Memory Fkt (strcmp(), memset())
22 #include <errno.h>          // Detailliertere Fehlermeldungen
23
24 #include <sys/socket.h>     // Socket Fkt wie socket(), bind(), listen()
25 #include <arpa/inet.h>     // Wichtige Funktionen wie htons() etc.
26 #include <net/ethernet.h>  // struct ether_header, ETHER_ADDR_LEN
27 #include <netinet/ether.h> // ether_aton()
28
29 #include <netinet/ip.h>     // struct iphdr
30 #include <netinet/ip_icmp.h> // struct icmphdr
31
32 // (siehe "man 7 packet")
33 #include <linux/if_packet.h> // struct sockaddr_ll
34
35 // Netzwerk Interface: (siehe "man 7 netdevice")
36 #include <sys/ioctl.h>     // ioctl(): Abfrage des Netw Interf Treibers
37 #include <net/if.h>        // struct ifreq
38
39
40 #define SIZE_ETHERHDR      sizeof(struct ether_header)
41 #define SIZE_IPHDR         sizeof(struct iphdr)
42 #define SIZE_ICMPHDR       sizeof(struct icmphdr)
43 #define SIZE_RANDOMDATA   20
44
45
46
47
```

```

48
49 unsigned short comp_chksum(unsigned short *addr, int len)
50 {
51
52     /**
53     * Quelle: RFC 1071
54     * Berechnet die Internet-Checksumme
55     * Gueltig fuer den IP-, ICMP-, TCP- oder UDP-Header
56     *
57     * *addr : Zeiger auf den Anfang der Daten
58     *         (Checksummenfeld muss Null sein)
59     * len   : Laenge der Daten (in Bytes)
60     *
61     * Rueckgabewert : Checksumme in Network Byte Order
62     **/
63
64     long sum = 0;
65
66     while( len > 1 ) {
67         sum += *(addr++);
68         len -= 2;
69     }
70
71     if( len > 0 )
72         sum += * addr;
73
74     while (sum>>16)
75         sum = ((sum & 0xffff) + (sum >> 16));
76
77     sum = ~sum;
78
79     return ((u_short) sum);
80 }
81
82
83 int main(int argc, char *argv[])
84 {
85
86     int sock, bytes;
87
88     // Socket Adress-Datenstruktur fuer PACKET Sockets
89     struct sockaddr_ll Sock_Addr;
90
91     // Informationen zum Network Interface
92     struct ifreq ifr;
93
94     //Die Groesse des gesamten Pakets
95     unsigned int SIZE_PACKET = SIZE_ETHERHDR + SIZE_IPHDR +
96                               SIZE_ICMPHDR + SIZE_RANDOMDATA;

```

```

97
98 //Der Puffer fuer das gesamte Paket
99 unsigned char packet[SIZE_PACKET];
100
101 // Die Zeiger der Header struct Typen werden hier direkt
102 // auf den Puffer gecastet
103 struct ether_header *etherheader = (struct ether_header *)packet;
104 struct iphdr *ipheader = (struct iphdr*)(packet + SIZE_ETHERHDR);
105 struct icmp_hdr *icmpheader = (struct icmp_hdr*)(packet +
106                               SIZE_ETHERHDR + SIZE_IPHDR);
107 char *icmp_randomdata = (char*)(packet + SIZE_ETHERHDR +
108                               SIZE_IPHDR + SIZE_ICMPHDR);
109
110
111 // Are you root?
112 if(getuid() != 0) { printf("You must be root!\n"); exit(1); }
113
114
115 // packet Buffer mit Nullen fuellen
116 memset(packet, 0, SIZE_PACKET);
117
118 // Ethernet Header Daten setzen
119 memcpy(etherheader->ether_dhost,
120        (u_char *)ether_aton("00:04:0e:4b:4c:45"), ETHER_ADDR_LEN);
121
122 memcpy(etherheader->ether_shost,
123        (u_char *)ether_aton("12:23:34:45:56:67"), ETHER_ADDR_LEN);
124
125 etherheader->ether_type = htons(ETHERTYPE_IP);
126
127 // IP Header Daten setzen
128 ipheader->ihl = 5; // 5 * 32 Bit
129 ipheader->version = 4;
130 ipheader->tos = 0;
131 ipheader->tot_len = htons(SIZE_PACKET - SIZE_ETHERHDR);
132 ipheader->id = htons(random());
133 ipheader->frag_off = htons(IP_DF); // Don't fragment
134 ipheader->ttl = 64;
135 ipheader->protocol = IPPROTO_ICMP;
136 ipheader->check = 0; // Muss Null sein. Berechnung folgt.
137 ipheader->saddr = inet_addr("192.168.100.24");
138 ipheader->daddr = inet_addr("192.168.100.1");
139
140 // ICMP Header Daten setzen
141 icmpheader->type = ICMP_ECHO;
142 icmpheader->code = 0;
143 icmpheader->checksum = 0; // Muss Null sein. Berechnung folgt
144 icmpheader->un.echo.id = htons(12345);
145 icmpheader->un.echo.sequence = htons(1);

```

```

146
147 // Random Data im ICMP Paket
148 char icmp_random_data[SIZE_RANDOMDATA] = "HALLO WELT";
149 memcpy(icmp_randomdata, &icmp_random_data, SIZE_RANDOMDATA);
150 printf ("Random Data: %s\n",icmp_randomdata);
151
152
153 // Berechnung der IP Checksumme:
154 ipheader->check = comp_chksum((unsigned short*)ipheader, SIZE_IPHDR);
155 printf ("IP Checksumme: %x \n", ipheader->check);
156
157 // Berechnung der ICMP Checksumme:
158 icmpheader->checksum = comp_chksum((u_short*)icmpheader,
159                                     SIZE_ICMPHDR + SIZE_RANDOMDATA);
160
161
162 // PACKET Socket aufbauen
163 if ((sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) == -1 )
164 {
165     perror("socket() failed");
166     exit(1);
167 }
168
169
170 // Indexnummer des Network Interfaces erfragen
171 strncpy(ifr.ifr_name, "eth1", sizeof(ifr.ifr_name));
172
173 if (ioctl(sock, SIOCGIFINDEX, &ifr) != 0)
174 {
175     perror("ioctl(SIOCGIFINDEX) failed");
176     exit(1);
177 }
178
179
180 // Informationen zum Netzwerk-Interface des Absenders
181 // werden in Sock_Addr festgelegt und mit bind() an den socket
182 // gebunden.
183
184 // Sock_Addr mit Null initialisieren
185 memset(&Sock_Addr, 0, sizeof(struct sockaddr_ll));
186
187 // Auskommentierte Felder sind fuer den Versand nicht noetig...
188
189 // LinkLayer Adressfamilie:
190 Sock_Addr.sll_family      = AF_PACKET;
191
192 // Interface Number
193 Sock_Addr.sll_ifindex    = ifr.ifr_ifindex;
194

```

```

195 // LinkLayer Protokoll: siehe if_ether.h
196 // Sock_addr.sll_protocol = htons(ETH_P_802_3);
197
198 // LinkLayer Adresstyp (MAC): siehe if_arp.h
199 // Sock_Addr.sll_hatype = htons(ARPHRD_ETHER);
200
201 // LinkLayer Pakettyp (nur sinnvoll beim Empfang)
202 // Sock_Addr.sll_pkttype = PACKET_OTHERHOST;
203
204 // Laenge der Hardwareadresse
205 // Sock_Addr.sll_halen = ETHER_ADDR_LEN;
206
207 // Absender MAC Adresse:
208 // memcpy(Sock_Addr.sll_addr, (u_char *)ether_aton(eth_s_mac),
209 //         ETHER_ADDR_LEN);
210
211
212 if (bind(sock, (struct sockaddr*)&Sock_Addr),
213      sizeof(struct sockaddr_ll) < 0 )
214 {
215     perror("bind() failed");
216     exit(1);
217 }
218
219
220 // Paket kann einfach per send() gesendet werden, da
221 // Socket Adresse bereits mit bind() an der Socket
222 // gebunden wurde!
223
224 if ((bytes = send(sock, packet, SIZE_PACKET, 0)) == -1)
225 {
226     perror("send() failed");
227     exit(1);
228 }
229
230 printf ("\nPaket gesendet.\n");
231
232 return 0;
233 }
234
235 /** Ende von icmp_packet_gen.c */

```

Listing 28: Beispielprogramm: icmp_packet_gen.c

Literatur

- [1] Bauer, Jochen: *Re: Linux Blind TCP Spoofing*.
Bugtraq Mailing List, 10.03.1999.
<http://seclists.org/bugtraq/1999/Mar/0057.html>
- [2] Blatter, Markus: *Intrusion Detection Systeme*.
Seminararbeit: IT Sicherheit.
Universität Zürich, Zürich, 07.01.2002
http://www.ifi.unizh.ch/ikm/Vorlesungen/Sem_Sich01/Blatter.pdf
- [3] Döring, Ralf: *Kurzeinführung zum Berkeley Socket API*.
TU Ilmenau, Ilmenau, 2006.
<http://www.tu-ilmenau.de/fakia/fileadmin/template/startIA/...telematik/lehre/praktikum/socket-api.pdf>
- [4] Howard, Michael: *A little more info on raw sockets and Windows XP SP2*.
http://blogs.msdn.com/michael_howard/archive/2004/...08/12/213611.aspx
- [5] Opatz, Felix: *Netzwerkprogrammierung mit BSD Sockets*. (Rev.1.4)
<http://www.zotteljedi.de/permalinks/socket-buch>
- [6] Stevens, W. Richard: *Programmieren von UNIX-Netzwerken* (2.Auflage).
Hanser, München, 2000.