Computernetzwerke, Teil 2

Skriptum zur Vorlesung CNW 3
des Bachelorstudienganges
Software Engineering
Wintersemester 2022

FH-Prof. Dipl.-Ing. Dr. Gerhard Jahn

31. August 2022

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Sicherheit in Netzen

> „400.000 Apps im Store von Apple sind 400.000 Angriffsmöglichkeiten.“
>
> Michael Hayden, Ex-Direktor von NSA und CIA im Sommer 2013, vgl. http://www.spiegel.de/international/world/how-the-nsa-spies-on-smartphones-including-the-blackberry-a-921161.html

1.1 Einführung

Für die Vernetzung entfernter Standorte standen in der Vergangenheit ausschließlich Standleitungen zur Verfügung. Diese wurden entweder für die exklusive Nutzung von Telekommunikations-Unternehmen angemietet oder standen im Eigentum der Firma. Für die gelegentliche Nutzung mit geringer Bandbreite bzw. für mobile Benutzer wurden Remote Access Services (RAS) verwendet. Angriffe von außerhalb und damit verbundene Schäden waren selten.

Heute sind viele Firmen direkt an das Internet angebunden. Auch private Nutzer sind durch Kabel-Modems, ADSL oder Flat-Free-Verträge basierend auf ISDN permanent online. Ohne entsprechende Sicherheitsmaßnahmen liegen diese Netze offen am Internet – sie sind praktisch eine Einladung für jeden Hacker.

Dieses Kapitel gibt einen Überblick über die Sicherheitsprobleme die u.a. durch eine Internet-Anbindung entstehen. Nachfolgende Kapitel zeigen Gegenmaßnahmen auf.

1.2 Was soll geschützt werden?

Strategien zur Sicherheit ergeben sich aus den zu schützenden Objekten [CZ95, S. 4]:

Daten: Die eigenen Daten haben schützenswerte Eigenschaften:

- **Vertraulichkeit:** Nicht Jeder soll die Daten einsehen können.
- **Integrität:** Nur dazu Berechtigte können die Daten ändern
- **Verfügbarkeit:** Die Berechtigten sollen Zugang zu den Daten haben.

Computer: Auch wenn die Daten bei einem Angriff unversehrt bleiben, ist die unerlaubte Nutzung der eigenen Ressourcen unerwünscht. Zumindest gehen Rechenzeit und Speicherkapazität verloren.

Reputation: Eindringlinge treten im Internet unter der ID des *geknackten* Netzes oder Benutzers auf. Angriffe auf weitere Systeme erfolgen in der Regel von gestohlenen Accounts aus.

1.3 Die Feinde – vor wem soll geschützt werden?

Angreifer haben unterschiedliche Motive und richten daher auch verschiedene Schäden an:

Interne Benutzer richten oft auch ungewollt durch Viren oder die Installation von *Trojanern* Schaden an.

Vandalen (Hacker) dringen oft nur aus Interesse in ein System ein. Die Bandbreite der Auswirkungen reicht von der unerlaubten Nutzung der Computer ohne weitere Schäden über das Lahmlegen der Computer durch *Denial of Service Angriffe* bis zur völligen Zerstörung der Daten.

Spione sind an sensiblen Daten interessiert. Oft werden Angriffe von Spionen der Konkurrenz spät oder gar nicht bemerkt.

1.4 Attacken

Dieser Abschnitt behandelt Attacken und Angriffe die in TCP/IP-Netzen wie dem Internet häufig vorkommen.

1.4.1 Spoofing – Vortäuschen einer falschen Identität

Der Angreifer täuscht eine falsche Identität vor.

IP-Spoofing: Services sind oft bestimmten Clients vorbehalten. Die Prüfung erfolgt dabei z. B. durch die IP-Adresse des Client-Host (vgl. die Ausgangsparameter bei `accept ()`[1]). Fälscht der Sender einer Nachricht seine IP-Adresse, kann er solche Services illegal nutzen. Dazu sendet der Angreifer gefälschte ARP-Responses mit seiner eigenen MAC-Adresse, womit durch den falschen Eintrag im ARP-Cache die Antworten des Servers zu ihm gelangen.

DNS-Spoofing: Eintragungen in DNS-Servern werden manipuliert. Durch die hierarchische Aufteilung der DNS-Datenbestände übernehmen auch andere Name-Server die manipulierten Daten.

1.4.2 Sniffing

Datenströme werden mit Sniffer-SW abgehört und analysiert. Besonders ergiebig sind solche Attacken, wenn Dienste wie Telnet oder FTP verwendet werden, welche die User-Daten unverschlüsselt übertragen.

1.4.3 Denial of Service

Hier handelt es sich um einen Missbrauch von Diensten zur Herabsetzung der Verfügbarkeit. Dies kann durch *Flooding* d.h. durch eine große Anzahl von Zugriffen oder durch Senden von bestimmten ungültigen Paketen erfolgen.

[1] z. B. unter `https://man7.org/linux/man-pages/man2/accept.2.html` oder im Kapitel 6.2.4.

SYN Flooding: Der Angreifer sendet TCP-Connection-requests mit einer falschen IP-Adresse (eine Adresse, die nicht existiert). Gemäß dem 3-Way-Handshake versucht der angegriffene Host die Verbindung zu bestätigen und wartet seinerseits bis zum Time-Out auf eine Bestätigung, die jedoch ausbleibt. In dieser Phase wird jedoch Speicher für die Verbindung gehalten, der bei einer hohen Anzahl von solchen halboffenen Verbindungen das System belastet.

Ping of Death: Hier wird ein Ping-Request mit einer zusätzlichen Payload von zumindest 65.510 Byte fragmentiert versendet. Der Empfänger erhält beim Defragmentieren ein Gesamtpaket von mehr als 65.536 Byte Länge (inkl. ICMP-Header). Dies brachte in der Vergangenheit manche Unix-Implementierungen zum Absturz.

UDP Flooding: UDP hat keine Flusskontrolle. Somit kann ein Angreifer Router und Hosts leicht mit UDP-Paketen überfluten.

RIP-Attack: In RIP senden Router ihre Routing-Tabellen alle 30 s an ihre Nachbarn. Durch Fälschen solcher Nachrichten können die Routing-Tabellen manipuliert werden.

Distributed Denial of Service (DDoS): Einer oder auch eine Kombination der obigen DoS-Angriffe wird in sehr hoher Anzahl ausgeführt. Dies würde natürlich auch die Netzwerkanbindung des Angreifers stark belasten. Der Angriff funktioniert nur, wenn der Angreifer schneller ist als der Angegriffene, was bei einer Attacke auf Server-Farmen großer Anbieter schwierig ist. Bei DDoS-Angriffen werden fremde Hosts benützt, in die vorher eingebrochen und entsprechende Daemons installiert wurden. Sind ausreichend viele Daemons verfügbar, wird die Attacke koordiniert gestartet. [Mar00; Str00]

1.4.4 Angriffe durch Fragmentierung

Hier wird versucht – durch geeignet manipulierte Fragmente – die Paket-Filter einer Firewall zu überwinden. Manche Paket-Filter untersuchen nur das erste Paket einer TCP-Verbindung:

Tiny Fragment Attack: Kleine Fragmente mit einem langen IP-Header (durch Optionen) bewirken, dass Teile des TCP-Headers im zweiten Fragment liegen. Paket-Filter untersuchen in der Regel die IP-Adressen und Ports. Dies ist bei solch kleinen Paketen nicht durch alleinige Untersuchung des ersten Fragmentes möglich.

Overlapping Fragment Attack: Ein Teil eines bereits versendeten und vom Paket-Filter akzeptierten Fragmentes wird mit geändertem Inhalt (der das Paket-Filter nicht passieren würde) wiederholt. Im Ziel-Host überschreibt das zweite Fragment das erste.

1.4.5 ICMP-Angriffe

redirect: Die Routing-Tabelle wird durch gefälschte *Redirect*-Nachrichten manipuliert.

destination unreachable, time exeeded: Gefälschte Pakete mit diesen Inhalten können den Empfänger dazu veranlassen, dass er zu den irrtümlich als unerreichbar gemeldeten Hosts keine Verbindung mehr aufbaut.

echo request an broadcast-Adresse: Alle angesprochenen Hosts senden ein *echo reply* an die im request angegebene und hier gefälschte Senderadresse.

1.4.6 TCP hijacking

Der Angreifer versucht eine bereits bestehende TCP-Verbindung zu übernehmen. Dabei verfolgt er zunächst den Zustand der Verbindung (Sequence Number und Acknowledgement Number). Dann übernimmt er stellvertretend für den Client die Verbindung zum Server in dem er die Pakete vom Server zu sich umleitet und dem Client den Abbruch der Verbindung vortäuscht. Alternativ übernimmt er auch noch die Verbindung zum Client und gibt sich ihm gegenüber als Server aus (*Man-in-the-Middle-Angriff*).

1.4.7 Replay-Angriff

Eine zuvor mittels Sniffing aufgezeichnete Sequenz wird zu einem späteren Zeitpunkt eingespielt. Der Angreifer fälscht dabei seine Adresse duch IP-Spoofing. Der Angreifer muss den Inhalt der Sequenz dabei gar nicht verstehen.

1.5 Stufen der Sicherheit

Keine Sicherheit: Die einfachste Methode ist es, sich auf die Sicherheitsmaßnahmen des Providers bzw. des Betriebssystems zu verlassen und keine weiteren Aktivitäten zu setzen.

Security Through Obscurity: Dies ist der obigen Stufe ähnlich. Die Sicherheit eines Systems besteht darin, dass potenzielle Angreifer nicht wissen, dass dieses System vorhanden und erreichbar ist. Mit zunehmender Lebensdauer verschwindet auch dieser Schutz. Oft reicht schon die Restrierung der Domain bzw. der IP-Adressen als Hinweis für Angreifer.

Host Security: Jeder Host wird individuell abgesichert. Diese Methode funktioniert, ist aber bei einer hohen Stückzahl bzw. inhomogener Ausstattung oder Konfiguration nicht praktikabel. Host Security wird in Hochsicherheitsbereichen zusätzlich zur Network Security angewendet.

Network Security: Der Zugang zu einem Netzwerk und die über diesen Zugang nutzbaren Services werden kontrolliert (im Sinne von eingeschränkt und überwacht). Maßnahmen sind z. B. Firewalls und Verschlüsselung beim Zugriff von außerhalb.

1.6 Weiterführende Literatur

SecurityFocus [Sec] galt lange als eine der wichtigsten Sites für Sicherheit im Internet. Aktuell wird dort ein Archiv von BugTraq gehalten. BugTraq ist eine Mailingliste, die sich mit Sicherheitslücken in Computersystemen befasst.

Das Computer Emergency Response Team [Sof] gibt – zumindest derzeit – sehr aktuelle Auskünfte über Sicherheitslücken. In CERT Coordination Center [CER] wird neben

einer allgemeinen Einführung speziell auf Heimanwender und deren *Always-on*-Zugänge eingegangen.

2 Firewalls

2.1 Grundlagen

2.1.1 Was ist eine Firewall?

Eine Firewall dient als Zugangskontrolle zwischen Netzwerken, genauer zwischen Zonen unterschiedlichen Vertrauens. Genau wie bei allen Arten von Zugangskontrollen erfolgt auch hier eine Überprüfung beim Wechsel zwischen Vertrauenszonen.

Eine Firewall implementiert zuvor definierte Zugangsregeln.

Dabei agiert sie ähnlich wie ein Wachposten, der eine ihm auferlegte Sicherheitspolitik exekutiert. Kontrolliert wird hier der Datenverkehr zwischen den Zonen. Als Ergebnis dieser Kontrolle wird die Kommunikation gestattet oder abgelehnt. Firewalls treffen diese Entscheidungen meist für einzelne Pakete, wobei je nach Intelligenz der Firewall – und natürlich auch der dahinter stehenden Sicherheitspolitik – mehr oder weniger Informationen herangezogen werden. Firewalls werden an neuralgischen Punkten platziert und sind in ihrer Funktionsweise Routern ähnlich bzw. haben in Bezug auf diese eine erweiterte Funktionalität. Tatsächlich entstanden einfache Firewalls durch Hinzufügen von Features zu Routern.

2.1.2 Warum eine Firewall?

Primär soll eine Firewall Schutz vor *dem Bösen* außerhalb des eigenen Netzes bieten. Dies gilt für die Anbindung zum Internet aber auch innerhalb eines Betriebes oder Standortes sind Zonen unterschiedlichen Vertrauens möglich. Schützenswert sind dabei die eigenen Daten und Resourcen. Eine Firewall kann auch als politisches Argument für die Einführung bzw. Aufrechterhaltung eines Internet-Zuganges gegenüber dem Management verwendet werden.

2.1.3 Connectivity vs. Security

Tatsächlich konkurrieren eine möglichst universelle Anbindung an die Außenwelt und die Sicherheit gegenüber Angriffen aus dieser miteinander. Genau das ist die Anforderung an eine passable Sicherheitspolitik. Sie soll die Vorteile einer guten Connectivity gegenüber den Nachteilen der dadurch eingeschränkten Sicherheit abwägen und einen guten d. h. individuellen Kompromiss finden.

2.1.4 Wogegen schützt eine Firewall?

Eine Firewall soll gegen unerlaubte externe Zugriffe auf interne Resourcen schützen. Trotzdem soll es internen Benutzern meist erlaubt sein, Services des externen Netzes möglichst ohne Beschränkung durch die Firewall zu nutzen. Bei richtiger Verwendung muss der ganze Verkehr mit der Außenwelt über die Firewall laufen. Somit ist sie auch ein guter Platz für die Protokollierung sowohl der akzeptierten als auch der abgewiesenen Zugriffe.

2.1.5 Wogegen schützt eine Firewall nicht?

Firewalls sind nicht perfekt. Im Detail versagen sie bei:

- allen Arten von Angriffen, die nicht über die Firewall laufen
  - z. B. über ein intern installiertes Modem
  - Social Engineering: Passwörter werden aus Benutzern *herausgelockt*
- Tunneling: d. h. eigentlich abzuweisende Zugriffe werden in ein genehmigtes Protokoll eingepackt
- Viren: einer Firewall ist es meist unmöglich, die volle Funktionalität eines Virenscanners zu integrieren. Problematisch ist auch die Vielfalt der Möglichkeiten zur komprimierten Übertragung. Einzelne Produkte unterstützen jedoch die Kooperation mit Virenscannern.

2.2 Entwurf von Firewalls

Unabhängig von der verwendeten Technologie sind beim Entwurf von Firewalls die folgenden Schritte durchzuführen:

1. **Sicherheitspolitik festlegen:** Zunächst muss festgelegt werden, was zu schützen ist und welche prinzipielle Strategie verfolgt wird. Die Bandbreite reicht hier von der *Beschränkung auf das unbedingt Notwendige* über diverse Abstufungen dazu, nur *wenige gefährliche Dienste zu verbieten.*
2. **Grad der Überwachung festlegen** – aus der zunächst eher allgemeinen Sicherheitspolitik werden konkrete Maßnahmen abgeleitet:
   - Was wird protokolliert?
   - Was wird gesperrt?
   - Was wird erlaubt?
3. **Finanzen:** Hier gibt es große Unterschiede. Kommerzielle Lösungen liegen bei den Kosten in der Größenordnung von bis zu USD 100.000, es existieren auch völlig freie Produkte, hier fällt nur die – meist höhere – Arbeitszeit an. Generell ist zu beachten: Eine Firewall verursacht auch im Betrieb einen erheblichen Aufwand. Die Regeln sind zu warten und Logs müssen ausgewertet werden.
4. **Implementierung:** Die Sicherheitspolitik wird durch die Implementierung exekutiert.

2.3 Technologien

2.3.1 Packet Filtering

Firewalls stehen meist anstelle von Routern an der Ankopplung zu einem größeren Netz. Sie erfüllen also auch zusätzlich die Aufgaben eines Routers. Ein *Packet Filter* ist ein Router, der zusätzlich bestimmte Pakete NICHT weiterleitet. Basis für diese Entscheidung sind die Header der Schichten 3 und 4, also z. B. Quell- und Zieladresse, Protokoll, Quell- und Zielport bei UDP und TCP. Zusätzlich werden meist auch noch das ein- und das abgehende Interface mit einbezogen.

Damit sind sie schneller als die anderen Varianten. Die Konfiguration ist meist aufwändig. Sie besteht aus einem – oft großen – Regelwerk, welches dann schwer durchschaubar wird. Für das Erstellen des Regelwerkes sind auch fundierte Kenntnisse über die verwendeten Protokolle notwendig.

2.3.2 Application Proxies

Proxies folgen einem völlig anderen Ansatz: Sie arbeiten auf der Applikationsschicht und koppeln über diese die beiden Vertrauenszonen. Ein Host auf dem ein Proxy installiert ist, agiert nicht unbedingt als Router. Er ist oft zwischen zwei Routern oder einem Router und einem Packet Filter platziert. Clients bauen die Verbindung *durch den Proxy* zum Server auf: Der Client baut zunächst die Verbindung zum Proxy auf, dieser baut dann – stellvertretend – selber eine Verbindung mit dem Server auf. Es entstehen somit zwei Verbindungen. Der Proxy agiert als Mittler zwischen Client und Server und steht damit eigentlich im Widerspruch zum Client/Server-Prinzip.

Application Proxies sind naturgemäß langsamer als Packet Filter. Wegen der Behandlung auf der Applikationsschicht können sie gezielt auf die Applikationsprotokolle eingehen, sie arbeiten daher sicherer. Natürlich muss für jedes Applikationsprotokoll ein eigener Proxy-Server eingerichtet werden, was aufwändig ist. Sie stellen auch hohe Ansprüche an die Performanz der Hardware.

2.3.3 Stateful Inspection – Dynamic Packet Filtering

Hier handelt es sich um eine Erweiterung der klassischen Packet Filter. Während dort nur die Header-Information des aktuellen Paketes verwendet wird, basiert bei *Stateful Inspection* die Entscheidung ob ausgefiltert werden soll auch auf der Historie der Verbindung und der Payload der Pakete. Hier wird wieder auf dem performanten Routing aufgesetzt und ein Kompromiss zwischen den schnellen aber unsicheren Packet Filtern und den langsamen aber sicheren Application Proxies gesucht. Mit fortschreitender Entwicklung wird diese Art der Firewalls durch Einbeziehung von Information höherer Schichten immer leistungsfähiger, sodass die Abgrenzung zum Application Proxy immer mehr verschwindet. Wir verweisen hier auf aktuelle Informationen diverser Hersteller von Firewalls.

Prinzipiell können Firewalls nach Stateful Inspection überall dort eingesetzt werden, wo bisher Packet Filter verwendet wurden. Durch ihre höhere Sicherheit machen sie zusätzliche Application Proxies in vielen Situationen (vgl. unten) überflüssig.

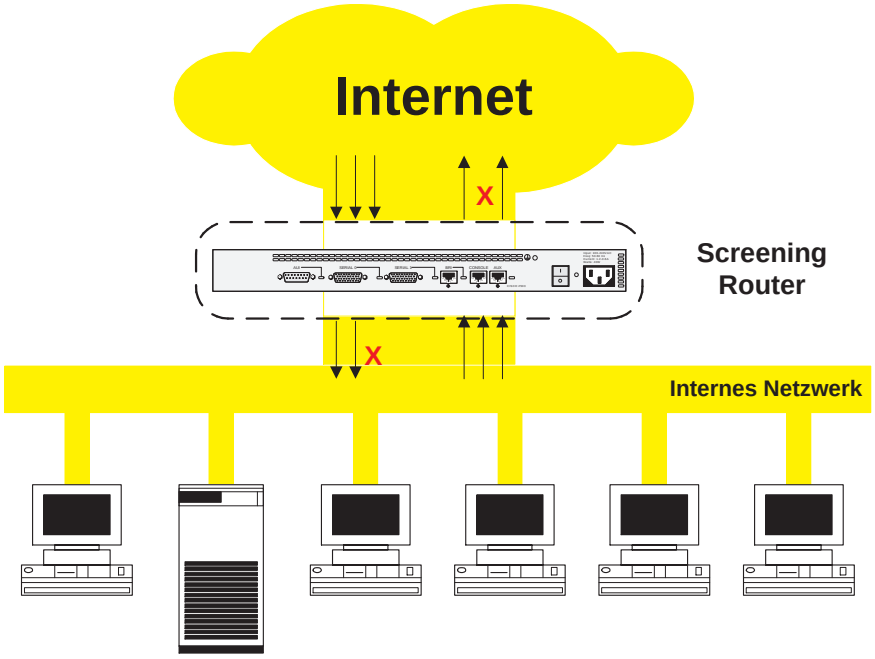

Abbildung 2.1: Packet Filter

2.4 Varianten

2.4.1 Packet Filter

In der einfachsten Variante wird der für die externer Anbindung – bzw. den Übergang zwischen Vertrauenszonen – zuständige Router durch ein Packet Filter ersetzt (siehe Abb. 3.1).

2.4.2 Screened Host

Ausgangspunkt ist das erste Szenario. Zusätzlich wird ein Proxy Server innerhalb der inneren Vertrauenszone eingerichtet. Damit aller Verkehr zwangsweise über ihn laufen muss, akzeptiert der Packet Filter nur Pakete die vom Proxy Server stammen bzw. an ihn gerichtet sind (Abb. 3.2 auf der nächsten Seite).

2.4.3 Screened Subnet

Zusätzlich werden Server für die Repräsentanz nach außen außerhalb des Packet Filters platziert. Damit belastet dieser Verkehr den Packet Filter nicht. In der Regel stellen Service Provider für die Ankopplung reine *Transportnetze* zur Verfügung, die dem Kunden nur eine einzige IP-Adresse bereitstellen. Deshalb sind die Server für die Internetpräsenz in einem eigenen Segment organisiert, der *Demilitarisierten Zone* (Abb. 3.3 auf der nächsten Seite).

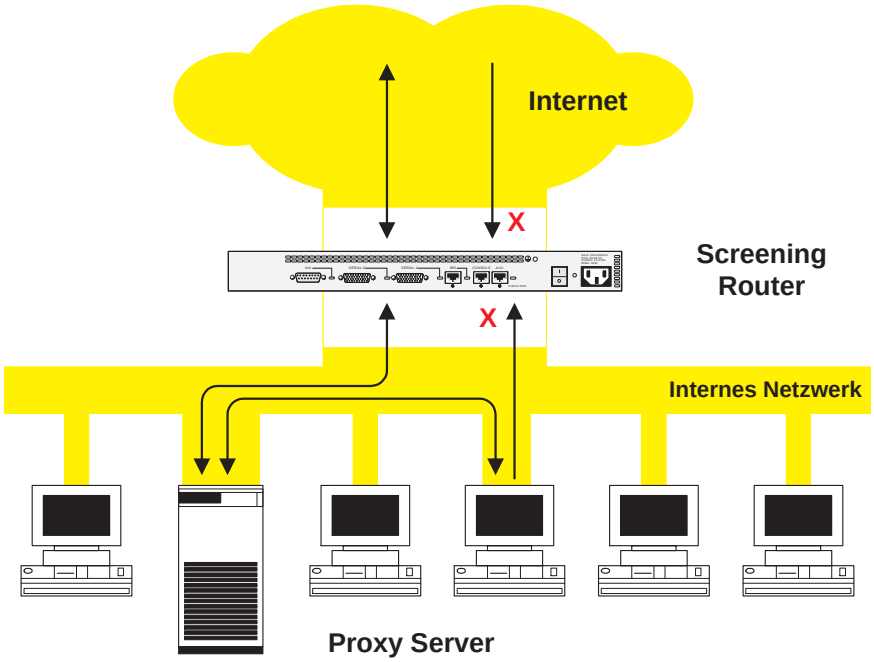

Abbildung 2.2: Screened Host

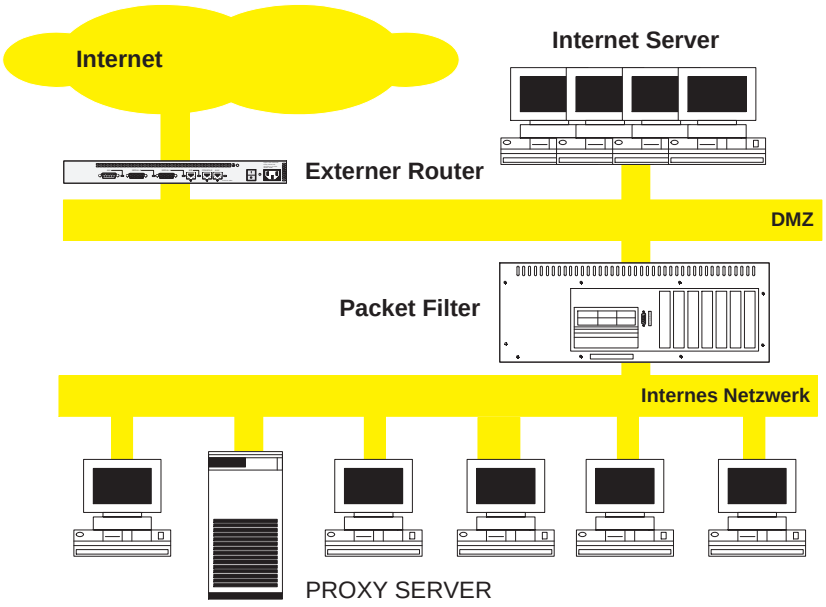

Abbildung 2.3: Screened Subnet

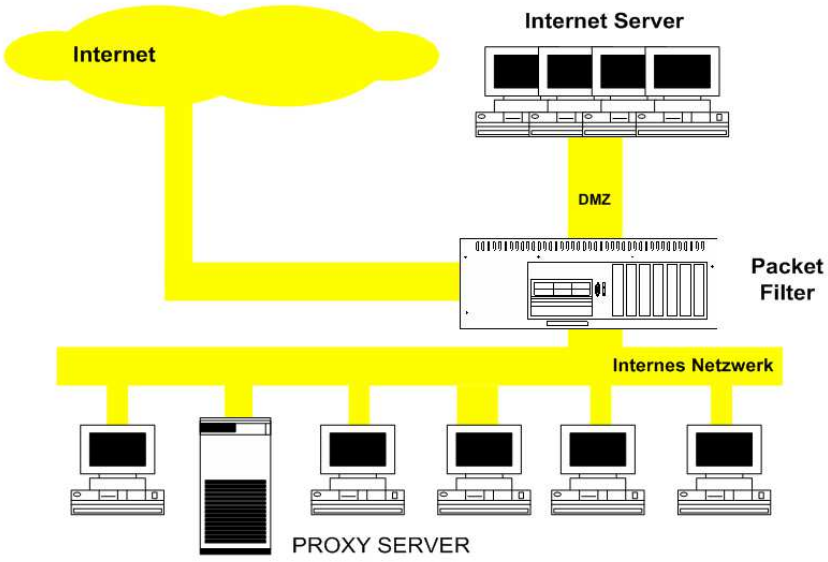

Abbildung 2.4: Eigene demilitarisierte Zone

2.4.4 Eigene demilitarisierte Zone

In der obigen Variante geht auch der Verkehr der internen Zone nach außen über die DMZ, was nicht unbedingt notwendig ist. Besitzt der Packet Filter mehrere Interfaces, dann kann die DMZ als eigenes Segment ausgeführt werden (Abb. 3.4).

2.4.5 Mehrere interne Zonen

Auch mehrere interne Zonen können über eigene Interfaces des Packet Filters gekoppelt werden. Damit bilden die internen Zonen auch verschiedene Vertrauensstufen, was besonders für verschiedene Abteilungen interessant ist. Im Extremfall entsteht hinter dem Packet Filter ein Backbone, an das Abteilungen je nach Sicherheitsbedürfnis über einen Router oder auch über weitere Firewalls angebunden sind (Abb. 3.5 auf der nächsten Seite und 3.6 auf Seite 26).

2.4.6 Virtual Private Network

Ein VPN hat die Aufgabe, Partner durch Einrichtung eines gesicherten d. h. verschlüsselten Kanals über ein an sich unsicheres Medium wie z. B. das Internet zu verbinden. Das unsichere Medium beginnt meist gleich hinter der Firewall. Daher liegt es nahe, Features für VPN gleich in die Firewall-Implementierung einzubauen. Auch hier verweisen wir auf die Informationen diverser Hersteller (Abb. 3.7 auf Seite 26).

2.4.7 Virus-Erkennung

Umfassende Virus-Erkennung übersteigt die Leistungsfähigkeit derzeitiger Firewalls. Allerdings kann eine Firewall mit einer auf einem gesonderten Host installierten Erkennungs-

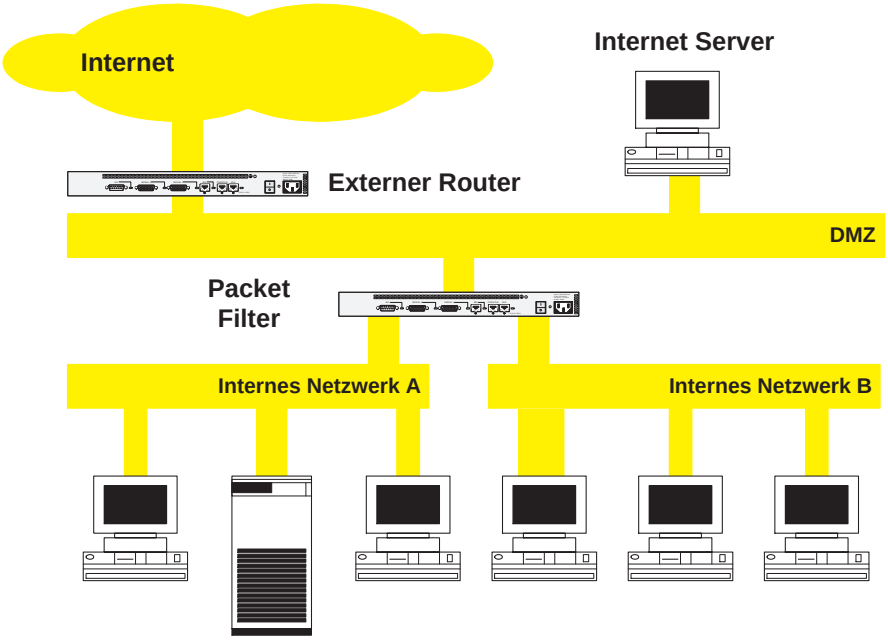

Abbildung 2.5: Kopplung interner Zonen über ein Packet Filter

Software für Viren kooperieren. Geeignete Software kennt neben gängigen Virenmustern auch vielfältige Arten der Komprimierung und applikationsabhängige Spezifika wie z. B. die Kodierung von Word-Macros. Die Firewall delegiert dann die Prüfung auf Viren an den externen Viren-Checker und berücksichtigt dessen Aussage (vgl. Abb. 3.8 auf Seite 27).

2.4.8 Falsche Lösungen

Einige Kardinalfehler beim Entwurf von Firewall-Lösungen bewirken ein Versagen der Schutzmechanismen. Dazu gehören der *Bypass* durch intern installierte Modems und auch das Einrichten eines internen RAS-Servers für gute Kunden (Abb. 3.9 auf Seite 27 und 3.10 auf Seite 28).

2.5 Personal Firewalls

Dieser *kleine Bruder* ausgewachsener Firewalls zielt auf den typischen Heimanwender ab, der einen einzigen Computer über eine Wählleitung, Kabelmodem o.ä. an das Internet anbindet. Ein eigener Host für die Firewall ist in solchen Fällen oft übertrieben. Eine Personal Firewall wird direkt auf dem angebundenen Host installiert und überwacht dort am Netzwerkinterface ein- und ausgehende Pakete. Sie fällt damit unter Host Security. Gängige Personal Firewalls arbeiten mit *Packet Filter* und *Stateful Inspection*.

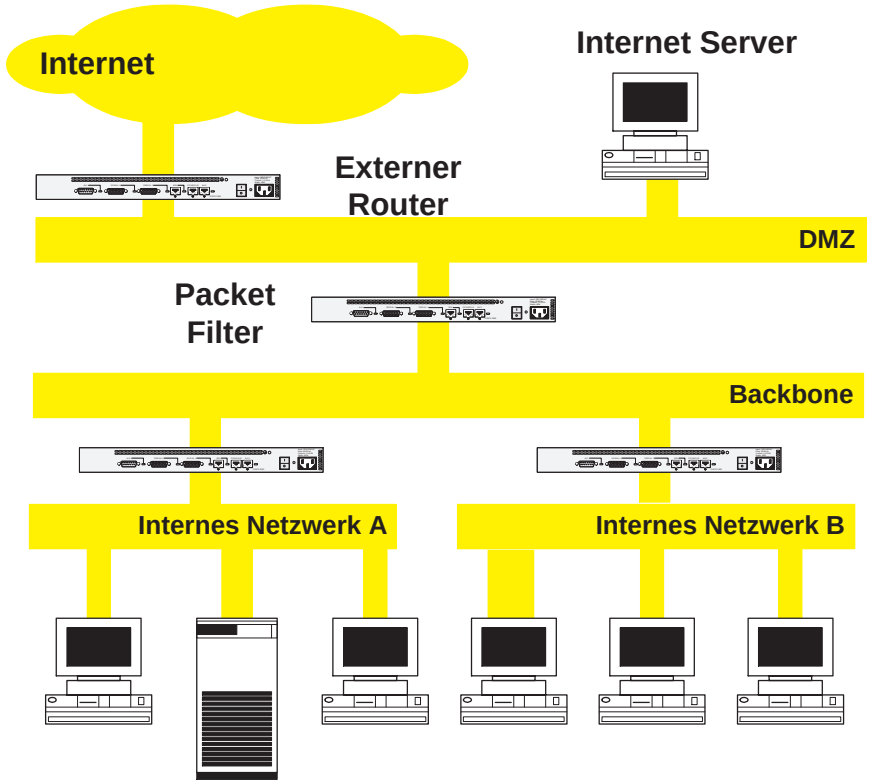

Abbildung 2.6: Kopplung interner Zonen über Backbone

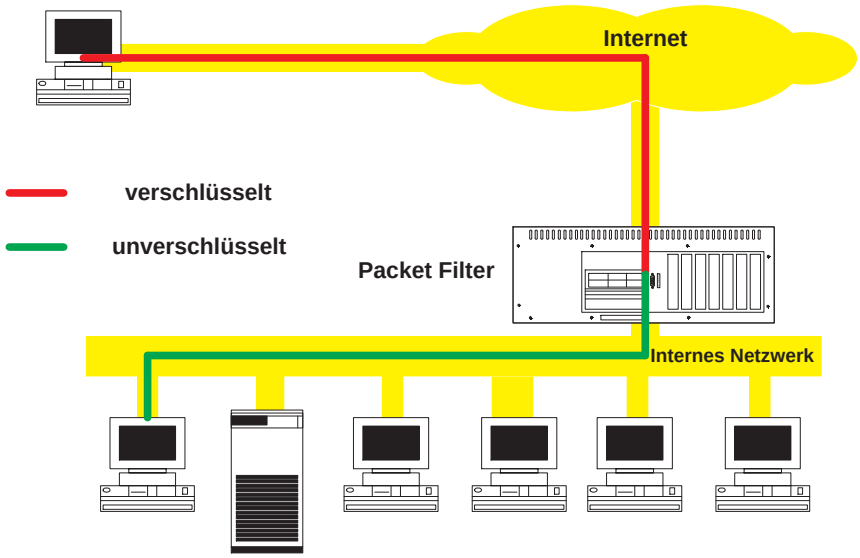

Abbildung 2.7: Firewall und Virtual Private Network

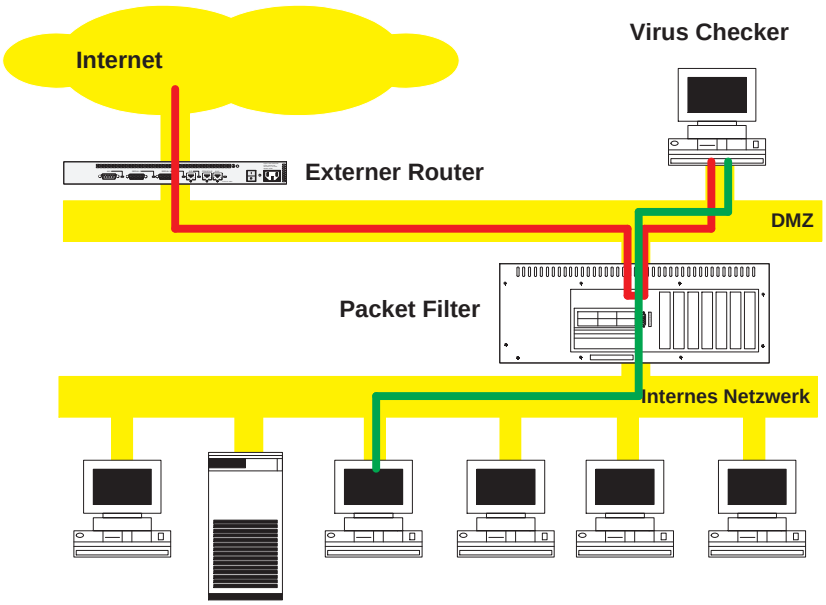

Abbildung 2.8: Firewall und Viren-Scanner

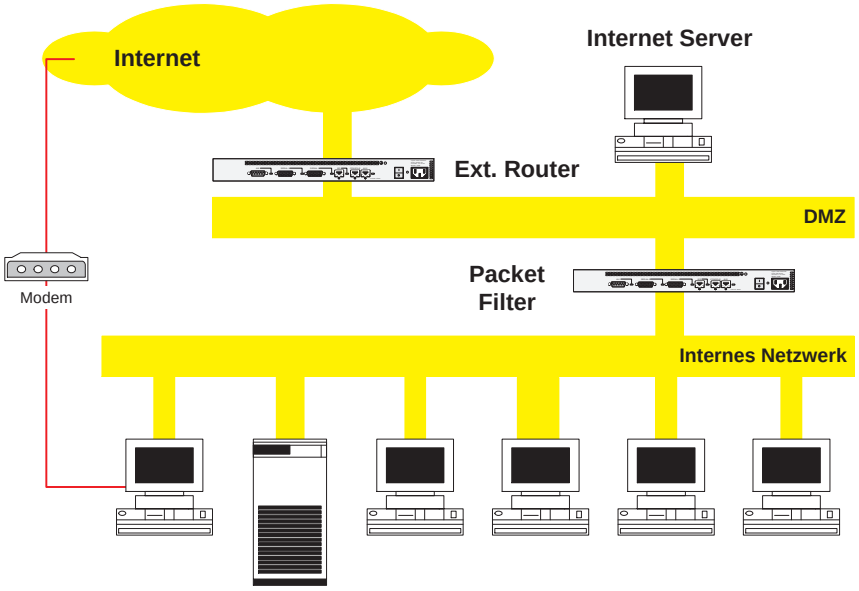

Abbildung 2.9: FALSCH: Firewall mit internem Modem

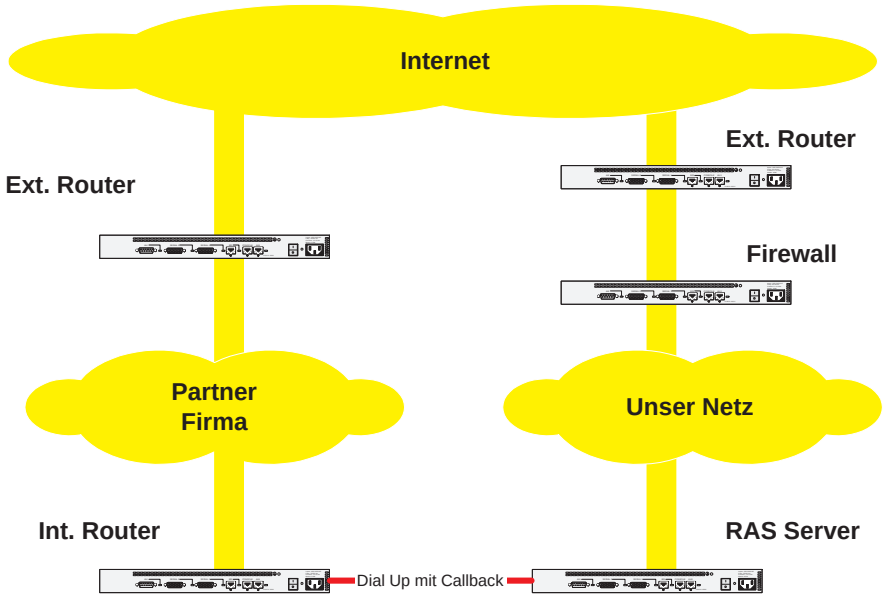

Abbildung 2.10: FALSCH: Firewall mit Hintertür durch RAS-Verbindung

2.6 Weiterführende Literatur

Die beiden Klassiker zum Thema Firewalls sind Cheswick und Bellovin [CB94] mit einer guten und umfassenden Einführung in die Thematik bzw. Chapman und Zwicky [CZ95] mit mehr technischen Details für die Implementierung.

Informationen über aktuelle Trends findet man in diversen Quellen am WEB, besonders bei den Seiten namhafter Hersteller von Firewall-Lösungen (vgl. dazu die Links im Foliensatz).

3 Kryptografie

> Die Inhalte zu diesem Kapitel entnehmen Sie bitte dem Foliensatz zur Kryptografie, im Skriptum finden Sie lediglich das Grundgerüst der Inhalte zur besseren Übersicht.

Dieses Kapitel dient der Vorbereitung auf das nachfolgende Kaptitel *Virtual Private Networks*, dort werden Grundlagen der Kryptografie benötigt. Die Sichtweise hier ist die des Anwenders und nicht des Mathematikers.

3.1 Begriffe

Welche Bereiche sind relevant?

Kryptografie: Wissenschaft vom Geheimhalten von Nachrichten

Kryptanalyse: Kunst vom *Brechen* kryptografischer Methoden

Kryptologie: Teilbereich der Mathematik, liefert den mathematischen Hintergrund zu kryptografischen Methoden

Steganografie: *Verstecken* wichtiger Nachrichten in harmlosen Nachrichten

Wozu dient die Kryptografie?

Integrität: Können Dritte die Nachricht verändern?

Geheimhalten von Nachrichten: Können Dritte die Nachricht lesen?

Authentifizierung: Ist der Sender eindeutig bestimmbar?

Verbindlichkeit: Kann der Sender später die Nachricht verleugnen?

3.1.1 Klassifizierung von Verschlüsselungsalgorithmen

Bei *alten* Algorithmen zur Verschlüsselung basierte die Sicherheit auf der Geheimhaltung des Algorithmus. Man spricht dann von einem *eingeschränkten Algorithmus*. Nachteilig ist, dass hier die Güte des Verfahrens nicht objektiv überprüft werden kann.

Moderne Algorithmen sind offen gelegt, geheim gehalten wird lediglich eine individuelle Parametrierung – der Schlüssel.

Symmetrische Algorithmen

Bei symmetrischen Algorithmen erfolgt die Chiffrierung und auch die Dechiffrierung mit dem gleichen Schlüssel. Sie sind performant, problematisch ist besonders der Transport (Austausch) der Schlüssel.

Public-Key-Algorithmen

Bei Public-Key-Algorithmen besteht der Schlüssel aus einem Schlüsselpaar: Der *öffentliche Schlüssel (Public Key)* wird veröffentlicht, der *private Schlüssel (Private Key)* wird geheim gehalten und muss nicht transportiert werden. Nachrichten werden mit dem Public Key des Empfängers chiffriert, nur der Empfänger selber kann die Nachricht mit seinem Private Key entschlüsseln. Einige konkrete Public-Key-Algorithmen erlauben auch die umgekehrter Verwendung, was bei digitalen Signaturen verwendet wird.

Nachteilig ist hier, dass das Verfahren um ca. den Faktor 10 langsamer ist als symmetrische Algorithmen.

3.1.2 Hybride Kryptosysteme

Hybride Kryptosysteme kombinieren die Vorteile beider Algorithmen-Klassen – nämlich die Geschwindigkeit und die Sicherheit: Daten werden performant mit einem symmetrischen Algorithmus verschlüsselt, wobei der Schlüssel zufällig gewählt wird (Sitzungsschlüssel). Der Sitzungsschlüssel wird dann mit dem Public Key des Empfängers verschlüsselt und das Ergebnis mit der verschlüsselten Nachricht an den Empfänger übermittelt. Symmetrische Schlüssel gelten derzeit ab einer Länge von ca. 128 bit als sicher, bei dieser Länge ist die schlechte Performanz von Public-Key-Algorithmen irrelevant.

Auch bei der digitalen Signatur wird meist nicht die eigentliche Nachricht mit dem eigenen Private Key verschlüsselt, sondern ein *repräsentativer Fingerabdruck* fixer Länge (vgl. Abschnitt 4.2 auf der nächsten Seite).

3.1.3 Kryptanalyse

Das Ziel der Kryptanalyse ist das Herstellen des Klartextes ohne Kenntnis des Schlüssels bzw. das Erfahren des Schlüssels. Sie wird einerseits bei bösartigen Angriffen aber auch andererseits bei der Bestimmung der Güte eines Kryptosystems verwendet. Je nach Art des Angriffs unterscheidet man:

Ciphertext-Only-Angriff: Hier steht dem Angreifer lediglich der verschlüsselte Text zur Verfügung.

Known-Plaintext-Angriff: Der Angreifer kennt auch den Plaintext und versucht den Schlüssel herauszufinden.

Chosen-Plaintext-Angriff: Hier kann der Angreifer den Plaintext und den dazugehörigen Ciphertext selbst wählen. Diese Art kommt besonders bei Public-Key-Systemen vor.

Kryptanalyse mit Gewalt: Der Angreifer wählt einen nicht-technischen Ansatz für die Kryptanalyse.

3.1.4 Klassifizierung in Block- und Stromchiffren

Kryptografische Verfahren werden auch nach ihrer Arbeitsweise klassifiziert.

Blockchiffren bearbeiten im Zuge der Ver- bzw. Entschlüsselung die Nachricht in Portionen fixer Länge. Wenn keine besonderen Vorkehrungen getroffen werden, ergibt ein

bestimmter Inhalt eines Blocks Plaintext bei konstantem Schlüssel immer einen bestimmten, gleichen Inhalt im Ciphertext. Typisch sind Blocklängen von 64 bit.

Stromchiffren bearbeiten die Nachricht in sehr kleinen Einheiten - meist ein Bit oder ein Byte. Dabei wird jede Einheit mit einem anderen Teil des Schlüssels bearbeitet. Bekannt ist hier besonders das *One Time Pad*.

3.1.5 Basisverfahren von Blockchiffren: Substitution und Transposition

Bereits in den Anfängen der Kryptografie wurden die auch heute noch verwendeten Basisverfahren angewendet.

Bei der *Substitution* werden Symbole in der Nachricht durch andere Symbole ersetzt. Diese Substitution ist meist statisch (z. B. Cäsars Methode), es sind aber auch adaptive Substitutionen bekannt (z.B. die Enigma).

Die *Transposition* ändert die Reihenfolge der Symbole in der Nachricht. Historisch bekannt ist besonders die Spaltentransposition, optional auch mit Code-Wörtern.

Moderne symmetrische Blockalgorithmen wenden diese Basisverfahren wiederholt in Runden an. Die einzelnen Runden unterscheiden sich dadurch, dass sie durch verschiedene Teile des Schlüssels parametriert werden.

3.2 Gängige Algorithmen

Kryptografische Verfahren – wie wiederholte Substitution und/oder Transposition – werden in konkreten kryptografischen Algorithmen umgesetzt. Einen guten Überblick über solche Algorithmen und deren Anwendung gibt z. B. `http://hp.kairaven.de/bigb/big2.html`. An dieser Stelle seien lediglich die Namen gängiger Algorithmen klassifiziert angeführt.

Symmetrische Algorithmen: DES, IDEA , AES

Public-Key-Algorithmen: RSA

Einweg-Hash-Funktionen: MD5 (Anwendung in digitaler Unterschrift)

3.3 Steganografie

Ein etwas exotischer Zweig der Kryptografie ist die *Steganografie*. Hier geht es um das Verstecken von Nachrichten in anderen, harmlos erscheinenden Nachrichten. Der Zweck ist es, potenzielle Angreifer gar nicht anzulocken. Meist wird die geheime Nachricht zusätzlich chiffriert. Ein einfaches Werkzeug zum Verstecken von Nachrichten in Bildern ist z. B. S-Tools.

4 Virtual Private Networks

4.1 Idee – Aufgabenstellung

Virtual Private Networks erlauben vertrauliche Kommunikation über unsichere Netze. Mit *vertraulich* ist hier die Sicherheitsstufe eines rein privaten Netzes gemeint, das *unsichere* Netz ist meist das Internet. Erreicht wird dies meist durch ein *Einpacken* an sich unsicherer Protokolle in Protokolle, welche die Vertraulichkeit realisieren. Das Einpacken wird auch als *Tunneling* bezeichnet. Vertraulichkeit wird in der Regel durch Verschlüsselung der Daten realisiert.

Je nach Anwendung endet ein solcher Tunnel nicht beim Host sondern schon bei einem Router, der den Host gemeinsam mit anderen über den Tunnel zu den Kommunikationspartnern verbindet. Entscheidend dafür ist der Anwendungsfall (Remote Access, Filiale, Teilbereich eines Partners, ...).

Nun folgen konkrete Protokolle/Implementierungen für ein VPN. Sie unterscheiden sich (auch) in der Schicht, in der sie ansetzen.

4.2 Upper Layers: SSL, S-HTTP, SSH

4.2.1 Secure Sockets Layer (SSL)

SSL wurde von Netscape mit dem Ziel private Dokumente über das Internet zu übertragen entwickelt. Es ist eine Erweiterung des Socket-Interface um verschlüsselte Übertragung. Der häufigste Anwendungsfall ist der sichere Datenaustausch im WWW über das Protokoll https. Mittlerweile unterstützen praktisch alle Web-Browser SSL. Server werden gegenüber den Clients mittels Public-Key-Verfahren authentifiziert, wobei der Server ein von einer (autorisierten) Zertifizierungsstelle ausgestelltes Zertifikat verwendet. Die Authentifizierung des Client gegenüber dem Server ist optional und wird kaum verwendet.

4.2.2 Secure HTTP (S-HTTP)

S-HTTP ist eine Erweiterung von HTTP um verschlüsselte Übertragung. Es ist auf der Applikationsschicht platziert und damit nicht für die Übertragung anderer Protokolle als http geeignet. Nur wenige Browser unterstützen S-HTTP, es ist auch wenig verbreitet.

4.2.3 Secure Shell (SSH)

Einführung

SSH ist weit verbreitet und hat eine gänzlich andere Zielsetzung als die zuvor beschriebenen Protokolle. Es dient als Ersatz für Telnet und FTP. Bei diesen Protokollen werden die

eigentlichen Daten und auch die Account-Daten (User-ID und Passwort) im Klartext übertragen. SSH beherrscht verschiedene Algorithmen zum Verschlüsseln, wobei sich Client und Server beim Aufbau einer Verbindung auf ein (Bündel von) Verfahren einigen. SSH

- verschlüsselt die Daten (symmetrisch)
- identifiziert den Server-Host vor dem Client-Anwender durch einen Fingerprint
- identifiziert den Client-Anwender vor dem Server durch ein login

Die Identifikation des Client vor dem Server kann mittels User-ID/Passwort oder auch über ein Public-Private-Key-Paar erfolgen. Der Server besitzt dann den Public-Key, nur Clients, die den dazupassenden Private-Key besitzen, können eine Verbindung aufbauen. Der Private-Key ist meist mit einer Pass-Phrase geschützt. Von SSH existieren mehrere Implementierungen. Eine bekannte ist OpenSSH, von der auch mehrere Portierungen für Windows verfügbar sind. Als Client unter Windows ist Putty weit verbreitet.

SSH verwendet den Well Known Port 22, er wird vom Client `ssh` für den Telnet-Ersatz und von `sftp` für den Ersatz von `ftp` verwendet.

Tunneling mit SSH

Eine interessante Eigenschaft von SSH ist seine Möglichkeit, andere Applikationsprotokolle – die auf TCP aufbauen – zu tunneln. Das ist ziemlich einfach, es sind nur wenige Einstellungen am SSH-Client notwendig. Es gibt prinzipell zwei Arten von Tunneln: *Local Forwards* und *Remote Forwards*. Beide Forwards werden lokal vom SSH-Client initiiert.

Ein *Local Forward* ist eine Verbindung vom Client-Host durch den SSH-Server. Das Argument

```
-L <local_port>:<destination_host>:<destination_port>
```

angegeben beim Start des SSH-Client bereitet eine solche Verbindung vor.

`-L` richtet einen Local Forward von einem lokalen Port durch die SSH-Verbindung d.h. über den SSH-Host zu einem Ziel-Host ein.

`<local_port>` ist der Port am SSH-Client hinter dem der Tunnel beginnt. Der lokale SSH-Client wartet an diesem Port auf Client-Applikationen, die sich durch den Tunnel zum Ziel-Service verbinden.

`<destination_host>` ist der Ziel-Host auf dem sich der zu erreichende Service befindet. Dieser Host muss nicht direkt durch den Client-Host erreichbar sein, sein Name wird auch im Kontext des SSH-Host aufgelöst. So können z. B. auch Hosts hinter einer Firewall erreicht werden, wenn ein SSH-Host im internen Bereich extern zugänglich ist. Wenn der Ziel-Service ohnehin am SSH-Host läuft, reicht hier z. B. die Angabe `localhost`.

`<destination_port>` ist der Port des Ziel-Service am `<destination_host>`.

Vom Client zum SSH-Server läuft diese Verbindung über den gesicherten SSH-Tunnel. Genau genommen wird beim Anmelden des SSH-Client am SSH-Server mit dieser Option zunächst lediglich das Port Forwarding am SSH-Server eingerichtet, die Verbindung zu `<destination_host>` erfolgt erst, wenn eine Applikation mit dem Port `<local_port>`

des Client-Host kommuniziert. Oft soll das Nutzen des Port Forwarding lediglich lokal am Client-Host angeboten werden, dies ist die Voreinstellung von SSH. Der Port `<local_port>` wird dann nur an das Interface `localhost` gebunden. Dies kann mit der Option `-g` (global) geändert werden.

`putty` zeigt das erfolgreiche Einrichten des Port Forwarding im Event-Log an:

```
2013-09-12 10:17:45 Local port 8010 forwarding to 10.41.0.10:80
```

Auch jeder Verbindungsaufbau über den Tunnel wird im Event-Log protokolliert.

Das *Remote Forward* ist das Gegenteil: Der Tunnel geht vom SSH-Server-Host durch den SSH-Client-Host zum Ziel-Service. Auch hier muss der Ziel-Service nicht zwangsweise am Client-Host liegen sondern lediglich von diesem erreichbar sein. Eingerichtet wird ein Remote Forward mit:

```
-R <remote_port>:<destination_host>:<destination_port>
```

`-R` erzeugt einen Remote Forward d.h. bindet einen Port am SSH-Server. Der Forward erfolgt über den SSH-Tunnel zurück zum SSH-Client.

`<remote_port>` ist der Port am SSH-Host hinter dem der SSH-Server auf Client-Applikationen wartet. Die Applikationen werden dann durch den Tunnel zum Ziel-Service verbunden.

`<destination_host>` ist der Host mit dem Ziel-Service. Er muss lediglich für den SSH-Client sichtbar sein, nicht aber für den SSH-Server.

`<destination_port>` spezifiziert den Ziel-Service am Ziel-Host.

Mit Remote Forwarding können Sicherheitszonen durch Firewalls und auch NAT-Komponenten umgangen werden. Aus diesem Grund erlauben die meisten SSH-Server den Zugriff auf den `<remote_port>` nur lokal.

Alle getunnelten Verbindungen erfolgen über die eine Verbindung vom Client zum SSH-Server. Die meisten SSH-Clients zeigen durch das Commando `~#` eine Liste der aktuellen Verbindungen, die über die SSH-Verbindung getunnelt werden. Für X11-Forwarding existieren eigene Einstellungen, sie werden mit der Option `-X` aktiviert.

4.3 Layer 2 VPNs

Die Bezeichnung *Layer 2 VPN* ist irreführend, es wird nicht ein einzelner Hop auf Schicht 2 gesichert, sondern ein verschlüsselndes Protokoll, welches auf der Schicht 2 angesiedelt ist, durch ein höheres Protokoll getunnelt.

4.3.1 Point to Point Tunneling Protocol (PPTP)

Internet-Provider verwenden für den letzten Hop zum Kunden oft das *Point-to-Point-Protocol* (PPP). Es ist auf der Schicht 2 angesiedelt. Eine mögliche – und meist verwendete – Payload von PPP ist IP. Interessant ist PPP für VPNs, da es optional eine Verschlüsselung erlaubt. Diese Vertraulichkeit von PPP wird für eine Kommunikation über mehrere Hops durch das PPTP nutzbar, PPTP ist ein *Adapter*, welcher PPP als Payload aufnimmt und selber wieder eine mögliche Payload von IP ist.

Je nach Länge des Tunnels wird zwischen *compulsory mode* und *voluntary mode* unterschieden.

4.3.2 Layer 2 Tunneling Protocol (L2TP)

Das L2TP entstand aus dem PPTP und einem Protokoll von Cisco namens *Layer 2 Forwarding*. Das L2TP ist im RFC 2661 spezifiziert. Die Aufgabenstellung ist gleich wie bei PPTP, allerdings ist das L2TP eine mögliche Payload von UDP und es kann IP als Payload aufnehmen. L2TP verschlüsselt selber.

4.4 IPsec

4.4.1 Motivation und Ziele

Die Schicht 3 ist für die Realisierung von Vertraulichkeit und Integrität eine gute Wahl, weil Router zwar den L3-Header einsehen müssen, aber nicht die L3-Payload. Vertraulichkeit auf tieferen Schichten geht immer nur mit Tunneling. Erfolgen andererseits die Maßnahmen für ein VPN auf höheren Schichten, so müssen diese Maßnahmen in jedes Transport- beziehungsweise Applikations-Protokoll aufgenommen werden, was ungünstig ist.

4.4.2 Security Association

Eine *Security Association* ist eine Beziehung zwischen zwei IPsec-Partnern. Sie ist parametriert mit Attributen wie

- kryptografischer Algorithmus,
- Schlüssel
- Gültigkeitsdauer des Schlüssels.

Die beiden IPsec-Partner sind die End-Punkte einer VPN-Verbindung.

4.4.3 Protokolle und Modes

Das aus der Version 6 von IP bekannte Prinzip der Erweiterungs-Header findet sich auch hier: Die Header der beiden Protokolle von IPsec werden zwischen dem IP-Header und dem Header der IP-Payload eingeschoben. Für jedes der beiden Protokolle existieren auch noch zwei Modes, die sich strukturell – und damit auch in ihre Anwendung – voneinander unterscheiden.

Beide Protokolle setzen einen Mindestumfang von kryptografischen Algorithmen voraus, sind aber erweiterbar in Bezug auf die Integration neuer Verfahren.

Authentication Header

AH bietet Authentizität und damit auch Integrität, aber keine Vertraulichkeit. Seine Anwendung ist damit alleine wenig sinnvoll. Meist tritt es daher in Kombination mit ESP auf.

Encapsulating Security Payload

ESP bietet neben Authentizität und Integrität auch Vertraulichkeit.

Transport Mode

Im *Transport Mode* liegt der IPsec-Header (AH oder ESP) einfach zwischen dem IP-Header und der IP-Payload.

Tunneling Mode

Beim *Tunneling Mode* wird das ganze IP-Paket – also inklusive dem IP-Header – eingepackt und – nach dem Motto des Tunneling – mit einem neuen IP-Header versehen. Zwischen den beiden IP-Headern ist der IPsec-Header platziert. Das hat z. B. den Vorteil, dass der innere IP-Header private Adressen enthalten kann und diese bei ESP auch noch verschlüsselt werden können.

Kombinationen der Protokolle

Durch Kombinieren der beiden Protokolle, auch mit unterschiedlichen Modes und unterschiedlicher Reihenfolge kann IPsec individuell an konkrete Anwendungen angepasst werden.

4.4.4 Internet Key Exchange

Der Austausch der Parameter für das Einrichten der Security Associations ist nicht Gegenstand der IPsec-Protokolle. Hier werden zwei Varianten unterschieden:

- Bei *manual keying* werden diese Parameter statisch eingerichtet,
- alternativ wird mittels dem Protokoll *Internet Key Exchange* (IKE) zunächst eine eigene Security Association für den Schlüsselaustausch eingerichtet, anschließend werden über diese die Securtiy Assiciations für die Daten aufgebaut.

5 Programmierschnittstellen unter UNIX

5.1 Einführung

Das Betriebssystem Unix war von Beginn an eng mit Netzwerken verbunden. Aus diesem Grund sind Software-Schnittstellen für den Zugriff auf die Kommunikationsteile des Betriebssystems integraler Bestandteil jeder Unix-Implementierung.

Dem Programmierer bieten sich viele unterschiedliche Alternativen zum Realisieren seines Programmes an. Die *high level Schnittstellen* verbergen die Details der Kommunikation vor dem Applikationsprogrammierer. In den folgenden Kapiteln werden zwei typische Vertreter behandelt:

Sockets bauen auf Streams, also den Zugriffen auf Dateien, auf. Die Socket-Schnittstelle implementiert viele Routinen aus der Dateibehandlung auch für den Datenaustausch zwischen einzelnen Rechnern. Einmal als Stream geöffnete Socket-Kanäle können mit den konventionellen Stream-Befehlen bearbeitet werden.

Remote Procedure Calls (RPCs) gehen einen völlig anderen Weg: Funktionen, welche auf einem anderen Rechner (Server) implementiert sind, können wie lokal vorhandene Funktionen aufgerufen werden. Auch hier werden die Details der Kommunikation vor dem Anwendungsprogramm weitestgehend verborgen.

5.2 Socket-Interface

Hinweis: Dieser Abschnitt entstammt weitestgehend [Rag93, Kap. 7]. Diverse Web-Server bieten einführende Seiten zur Socket-Programmierung an (vgl. z. B. [Hal]). Comer und Stevens [CS96] behandeln das Thema umfassend, in [CS97] wird auf die Socket-Programmierung unter Windows eingegangen.

5.2.1 Einführung in Clients, Server, Daemons und Protokolle

Unter *Server* versteht man im Allgemeinen jemanden, der einen Dienst zur Verfügung stellt. Ein Server macht Ressourcen über das Netzwerk nutzbar. Ressource ist in diesem Zusammenhang ein weitläufiger Begriff. Eine Ressource kann sowohl Hardware (Drucker, Faxmodem, ...) als auch Software (Datenbank, Filesystem, Telnet, ...) sein. Meist erfolgt die Bereitstellung der Ressource durch ein Server-Programm, welches ein Server-Host als Server-Prozess ausführt. Der Server-Prozess läuft auf dem Host, an dem die Ressource lokal vorhanden ist, er wartet darauf, dass der zur Verfügung gestellte Dienst von einem Client angefordert wird. Server-Pozesse werden meist schon beim Hochlauf durch Start-up-Skripte gestartet.

Ein *Client* nutzt eine solche Ressource. Dazu ist ein Programm notwendig – das Client-Programm – welches ein Client-Host als Prozess exekutiert. Ein wichtiger Punkt beim

Client-Server-Konzept ist die Nutzung von Ressourcen UNABHÄNGIG von deren physikalischem Standort. Ein Client muss lediglich über das Netzwerk eine Verbindung zum Server herstellen. Natürlich funktioniert dieses Prinzip auch, wenn ein einziger Host den Client-Prozess und auch den Server-Prozess exekutiert. Der Host fungiert dann als Client-Host und auch als Server-Host.

In der UNIX-Welt werden Server-Prozesse meist als *Daemon* bezeichnet, besonders wenn der Server-Prozess üblicherweise beim Boot-Vorgang des Server-Host gestartet wird. In der Windows-Welt spricht man dann von einem *Service*.

Daemons oder Services zeichnen sich gegenüber anderen Prozessen dadurch aus, dass sie die meiste Zeit im Zustand *suspended* im Hintergrund schlummern und plötzlich für meist kurze Zeit auftauchen. Auslöser ist bei unserer Art von Daemons die Dienstanforderung durch einen Client. Nach der Diensterfüllung – die Sitzung mit dem Client endet – taucht der Daemon wieder in den Hintergrund ab, er wartet passiv auf den nächsten Client. Unter UNIX erkennt man ein Daemon-Programm an der Endung des Dateinamens mit `d`. Beispiele: `inetd`, `telnetd`, ... (vgl. dazu den Output des Kommandos `ps -ef | grep telnetd`).

Unter *Protokoll* versteht man die Beschreibung der Interaktion zwischen Server und Client. Das Protokoll definiert das genaue Format der Daten, die zwischen den beiden Prozessen ausgetauscht werden und auch deren Semantik.

5.2.2 Adressierung von Diensten

Um einen Dienst im Netzwerk zu finden, muss der Client erst einmal wissen, mit welchem Host er Kontakt aufnehmen muss. Dies erfolgt meist über eine im Netzwerk gültige Adresse, das ist im heute üblichen Protokoll-Stack TCP/IP entweder eine IP-Adresse oder ein Host-Name (Bsp.: `www.fh-hagenberg.at`). Auf dem adressierten Host gibt es nun aber mehrere *Kommunikationsendpunkte*. Diese Kommunikationsendpunkte, die sogenannten Ports, werden mit einer 16-Bit-Portnummer angesprochen. Jeder Server-Prozess *hört* einen solchen Port auf Anfragen von Clients ab (Bsp.: ftpd überprüft Port 21).

Unter UNIX sind alle Portnummern, die kleiner als 1024 sind, reserviert. Das bedeutet, dass nur Prozesse, die unter der UID `root` laufen, diese Ports benutzen dürfen. Das stellt eine einfache Form von Sicherheitsmechanismus dar.[1] Die meisten Daemons arbeiten mit reservierten Ports. So hat der Client die Sicherheit, mit einem echten Daemon zu kommunizieren und nicht mit einem Programm, das ein gewöhnlicher Benutzer geschrieben hat um Passwörter abzufangen. Der Zusammenhang zwischen Diensten und Portnummern – also die *Well Known Ports* – ist in der Datei `/etc/services` definiert. Hier ein Auszug daraus:

```
daytime        13/tcp
telnet         23/tcp
time           37/udp       timeserver
time           37/tcp       timeserver
fax_modem    2055/tcp       faxmodem
```

[1] Nur der Administrator des Server-Host kann einen Service an einen solchen *privilegierten* Port binden. Damit kann ein Client dem Server genau so trauen wie er dem Server-Host traut. Services hinter unpriviligierten Ports konnen von jedem beliebigen Benutzer eines Host gestartet werden.

Die erste Spalte eines Eintrages bezeichnet den Namen des Dienstes, die zweite die Portnummer sowie das verwendete Transport-Protokoll und der dritte Eintrag ist ein alternativer Name für den Dienst. In größeren UNIX-Netzwerken wird diese Information über Dienste nicht auf jedem Host in einer Datei abgespeichert, sondern liegt in einer zentralen Datenbank (Network Information Service).

5.2.3 Allgemeines zum Socket Interface

Sockets

Die Transportprotokolle TCP und UDP der TCP/IP-Protokollfamilie sind die Basis der Sockets. Ein Socket ist ein Kommunikationsendpunkt, an dem sich Anwendungsprogramm und Transportschicht treffen. Die ursprüngliche Schnittstelle zu TCP und UDP stammt aus dem Release BSD 4.2 des Berkeley UNIX. Sie besteht aus acht Sytemaufrufen und wird mit dem allgemeinen Namen *Sockets* bezeichnet.

Adressfamilien von Sockets

Die einfachste Adressschema ist die sogenannte *UNIX-Adressfamilie*. Dabei wird ein Socket mit einem UNIX-Pfadnamen assoziiert. Unter BSD-UNIX werden diese Sockets in der Verzeichnisstruktur als Einträge mit dem Typ `s` gezeigt:

```
turing% ls -l /tmp/mysocket
srwxrwxrwx      1       chris    0   Jul   1   22:00    /tmp/mysocket
turing%
```

Unter SVR4 UNIX wird dieser Typ als *Named Pipe* implementiert und im Verzeichnis mit der Kennung `p` angezeigt. Diese Art von Adressierung ist zwar einfach, aber für eine Kommunikation über das Netzwerk unbrauchbar.

Ein anderes Konzept für die Adressierung von Sockets ist das *Internet Domain Addressing*. Dabei stehen hinter dem Socket zwei Zahlenwerte: Die IP-Adresse (32 Bit) des Host, auf dem sich der Socket befindet und die Portnummer (16 Bit).

Socket-Aufrufe (z. B. die hier exemplarisch gewählte Routine `bind()`) müssen nun flexibel genug sein, um mit den unterschiedlichen Adressfamilien umgehen zu können. Dies wird durch eine flexible Funktionsschnittstelle erreicht. Für beide Arten von Socket-Adressen werden Strukturen definiert.

UNIX-Bereichsadresse:

```
struct sockaddr_un {
  short sun_family;          /* Tag: AF_UNIX */
  char  sun_path[108];       /* path name    */
};
```

Internet-Bereichsadresse:

```
struct sockaddr_in {
  short      sin_family;          /* Tag: AF_INET  */
  u_short    sin_port;            /* Port Number   */
  struct     in_addr sin_addr;    /* IP-address    */
  char       sin_zero[8];         /* Padding       */
};
```

```
struct in_addr {
  u_long s_addr;
};
```

Beide Strukturen enthalten am Beginn ein *Tag*, das unbedingt gesetzt sein muss. Dieses Tag benutzen Funktionen wie `bind()`, die nur einen Zeiger auf die Struktur (Übergabeparameter) erhalten, um herauszufinden, ob es sich um eine Struktur vom Typ `sockaddr_in` oder um eine vom Typ `sockaddr_un` handelt. Weiters muss noch ein zusätzlicher Parameter – der die Länge der Adresse enthält – übergeben werden. Dies vereinfacht die Implementierung dieser Systemfunktionen. Ein Aufruf der Funktion `bind()` könnte wie folgt aussehen:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

/* prototype:
int bind(int sockfd, const struct sockaddr *addrp, socklen_t addrlen);
 */

bind (sockfd, (struct sockaddr *) &server, sizeof(server));
```

Hier ist `server` z. B. vom Typ `struct sockaddr_in`.

Typen von Sockets

Zusätzlich zu den Adressfamilien besitzen die Sockets der Familie `AF_INET` auch noch einen Typ, der sich auf die Art des zugrundeliegenden Protokolls bezieht:

```
SOCK_STREAM     /* verbindungsorientierter Transport (TCP) */
SOCK_DGRAM      /* verbindungsloser Transport (UDP) */
SOCK_RAW
```

Der Typ `SOCK_RAW` wird dazu eingesetzt, um direkt mit der IP-Schicht zu kommunizieren. Dazu sind root-Rechte nötig.

5.2.4 Verbindungsorientierte Server

Bei dieser Art der Kommunikation müssen Client und Server ein bestimmtes Prozedere einhalten. Die Abbildung 6.1 auf der nächsten Seite zeigt die einzelnen Etappen mit den zugehörigen Systemaufrufen.

Verbindungsaufbau

Der wesentliche Unterschied zwischen Client und Server ist der, dass der Server passiv auf Arbeit wartet, der Client hingegen aktiv mit einem Server Verbindung aufnimmt.

Die vom Server auszuführenden Schritte:

1. Ein Socket der benötigten Adressfamilie und des benötigten Typs muss angelegt werden:

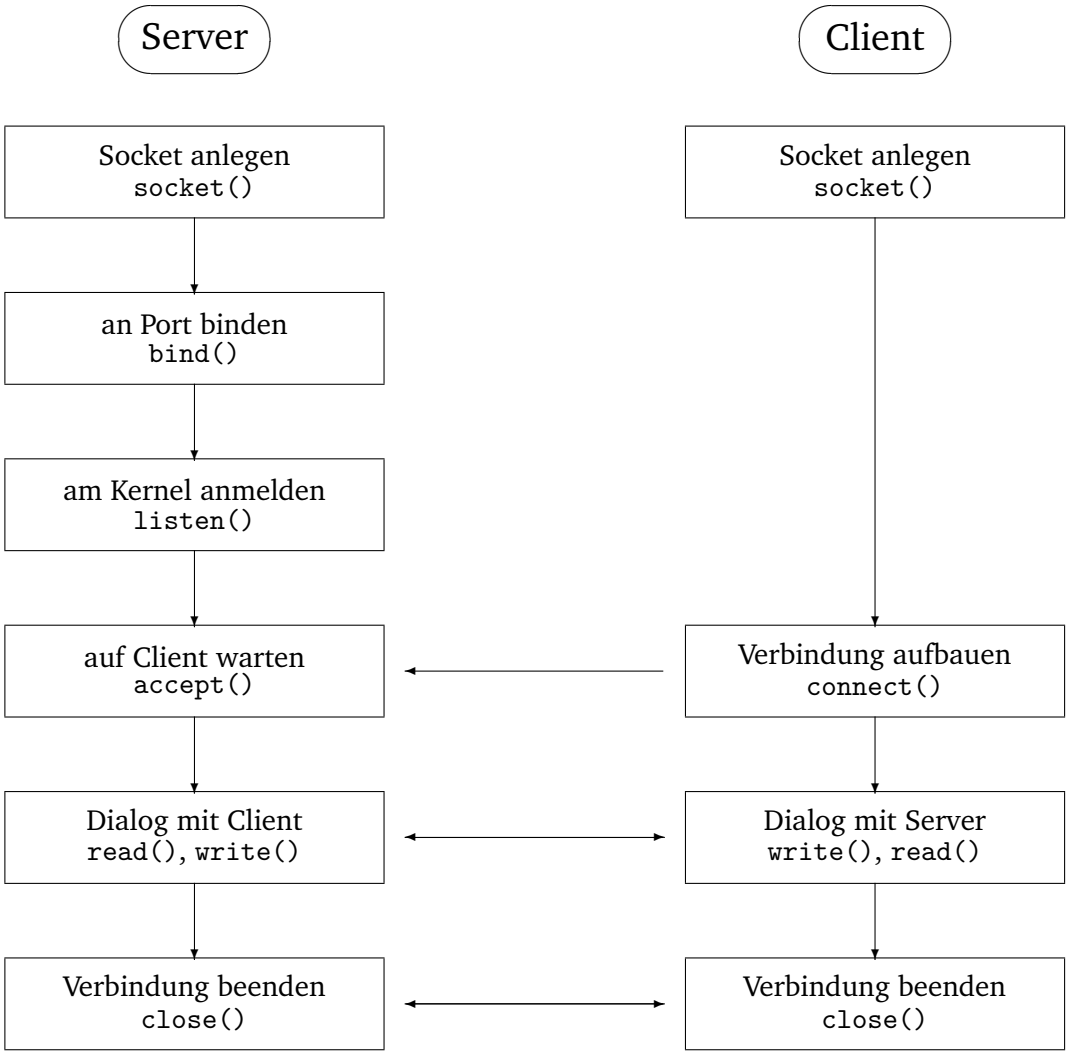

Abbildung 5.1: Verbindungsorientierte Client-/Serveroperationen

```
#include <sys/types.h>
#include <sys/socket.h>

/* prototype:
int socket(int family, int type, int protocol);
 */

int sockfd;
sockfd = socket (AF_INET, SOCK_STREAM, 0)
if (sockfd < 0) {
   perror ("cannot get socket");
   exit (1);
}
```

Das erste Argument gibt die Adressfamilie an, das zweite Argument spezifiziert den Socket-Typ und das dritte Argument gibt das Protokoll an, wird aber im Allgemeinen auf *0* gesetzt – d. h. *lass das System wählen*. Ist das Anlegen des Socket nicht erfolgreich, dann retourniert `socket()` mit dem Wert -1. In diesem Fall enthält die globale Variable `errno` den Fehlercode. Die Funktion `perror` gibt einen mit `errno` korrespondierenden Text auf `stderr` aus (vgl. `man perror` und `man strerror`). Viele Systemroutinen teilen so die Ursachen für einen Fehler mit. Zum Zweck der besseren Lesbarkeit wird das Testen der Return-Werte jedoch in den folgenden Code-Fragmenten weggelassen.

2. Eine Adresse wird an den Socket gebunden. Dabei handelt es sich je nach Adressfamilie entweder um einen Pfadnamen oder um eine IP-Adresse und eine Port-Nummer. Beispiel:

```
#define SERVER_PORT 2222
struct  sockaddr_in  server;

server.sin_family       = AF_INET;
server.sin_addr.s_addr  = INADDR_ANY;
server.sin_port         = htons (SERVER_PORT);

bind (
    sockfd,
    (struct sockaddr *) &server,
    sizeof(server)
);
```

Handelt sich ein einen experimentellen Server, so ist darauf zu achten, dass die Portnummer größer als 1024 ist. Die Portnummer muss natürlich mit dem Client abgestimmt sein, da dieser sonst den Server nicht findet. Das Macro `htons` konvertiert einen short-Wert von der internen Darstellung in eine (standardisierte) Netzwerk-Darstellung (*host to network short*). Damit werden die Unterschiede zwischen little und big endian ausgeglichen. Das Macro für die Gegenrichtung heißt `ntohs`. Weiters gibt es noch entsprechende Macros für long-Werte: `htonl` und `ntohl`. Diese sind für die IP-Adressen zu verwenden. Der der IP-Adresse des Socket zugewiesene Wert `INADDR_ANY` (vgl. include `netinet/in.h`) ist ein vordefinierter Wert und bedeutet, dass dieser Socket Verbindungen an jedem Netzwerk-Interface dieser Maschine akzeptiert. `INADDR_ANY` hat den Wert `FF...FF`. Daher wurde hier das an dieser Stelle

eigentlich notwendige Macro `htonl` weggelassen. Normalerweise hat ein Host nur eine einzige Netzwerkkarte. Ist ein Host mit mehreren Netzwerk-Interfaces ausgestattet (Gateway), so gehört zu jedem Interface eine eigene IP-Adresse. Sollen nur Verbindungen von einem bestimmten Interface akzeptiert werden, so kann die IP-Adresse des gewünschten Interface hier angegeben werden. Es ist jedoch zu beachten, dass diese Adresse überhaupt nichts damit zu tun hat, von welchen Clients Verbindungen akzeptiert werden.

3. Nun muss der Kernel informiert werden, dass von diesem Socket Verbindungen akzeptiert werden.

```
#include <sys/types.h>
#include <sys/socket.h>

/* prototype:
int listen(int sockfd, int backlog);
 */

listen (sockfd, 5);
```

   Das zweite Argument legt die Anzahl der wartenden Verbindungsanfragen fest. Eine Verbindungsanfrage liegt dann vor, wenn ein Client versucht mit einem Server Verbindung aufzunehmen, während ein anderer Client gerade mit dem Server kommuniziert. Die Anfragen werden der Reihe nach in eine Warteschlange eingereiht. Ist der angegebene Wert erreicht, wird die Verbindung zurückgewiesen.

4. Jetzt muss nur noch darauf gewartet werden, dass ein Client eine Verbindung aufbaut. Versucht ein Client nun eine Verbindung aufzubauen, so muss diese vom Server akzeptiert werden.

```
#include <sys/types.h>
#include <sys/socket.h>

/* prototype:
int accept(int sockfd, struct sockaddr *addrp, socklen_t *addrlen);
 */

struct sockaddr_in client;
int new_fd, client_len;

client_len = sizeof(client);
new_fd = accept (sockfd, &client, &client_len);
```

   Das zweite Argument liefert bei Akzeptieren der Verbindung die IP-Adresse und den Port des Client, der die Verbindung aufgebaut hat. Akzeptieren wir von jedem beliebigen Host Verbindungen, so können wir diesen Parameter ignorieren. Wollen wir eine Zugriffskontrolle durchführen, so können wir über diesen Parameter die Adresse des Client erfahren. Der Rückgabewert von `accept()` ist ein Deskriptor, der für die mit dem Client hergestellte Verbindung steht. Wir können nun mit den Funktionen `read()` und `write()` über den Socket `new_fd` auf diese Verbindung zugreifen.

Datentransport mittels Sockets

Nach dem Verbindungsaufbau verhält sich unser Deskriptor `new_fd` wie jeder andere Datei-Deskriptor. Man kann Operationen wie `read()`, `write()`, `dup()`, `close()`, ... darauf anwenden. Die Kommunikation zwischen Client und Server ist im Applikationsprotokoll definiert. Üblicherweise sieht ein typischer Dialog folgendermaßen aus: Ein Client stellt eine Anfrage an den Server und der Server antwortet. Die Verbindung wird durch ein Schließen des Deskriptors mit `close()` beendet. Der Partner erkennt dies durch einen Return-Wert von 0 beim nächsten `read()`. In der weiteren Folge ist der Server bereit für den nächsten Client – d. h. er sollte alle Aktionen ab einschließlich `accept()` in eine Endlosschleife einbetten.

5.2.5 Parallele Server

Server, die gleichzeitig nur einen einzigen Client bedienen, werden iterative Server genannt. Sie sind nur für Dienste geeignet, die eine äußerst kurze Verbindungsdauer aufweisen (z. B.: `timed`), da weitere Clients, die einen solchen Dienst anfordern, unbestimmte Zeit warten müssen. Das ist z. B. für einen SSH-Daemon untragbar. Aus diesem Grund wollen wir uns parallelen Servern zuwenden. Ein solcher Server erzeugt mit dem System Call `fork()` für jeden Client einen eigenen Kindprozess. Der Elternprozess übernimmt lediglich mit `accept()` die Verbindung und übergibt sie dem Kind. Da der Kindprozess die offenen Deskriptoren vom Elternteil erbt, ist der Server sehr einfach zu implementieren:

```
sockfd = socket ( ... );
bind (sockfd, ... );
listen (sockfd, ... );

while(1) {
  new_fd = accept (sockfd, ... ); /* await and accept connection */
  if (fork()==0) {              /* --- child: */
      close (sockfd);                 /* not needed in child */
      ServerProcess (new_fd);         /* do server application */
      close (new_fd);                 /* close connection */
      exit(0);                        /* end of child's part */
  } else {                      /* --- parent: */
      close(new_fd);                  /* not needed in parent */
  }                                   /* end of parent's part */
}
```

Diese Implementierung hat noch einen Nachteil: Terminierende Kind-Prozesse warten auf die Übernahme ihres Exit-Status durch den Vater-Prozess. Holt dieser den Exit-Status seiner Kinder nicht ab, so leben die Kind-Prozesse als „Zombies“ weiter.[2] In der nachfolgenden Programmskizze löst der Vater-Prozess dieses Problem mit Signalen:

```
/*
 * waiter accepts the exit code from a child created previously
 */
void waiter()
{
  int cpid, stat;
```

[2] Unter Unix scheinen solche Prozesse mit `ps` als `<defunct>` auf, Linux zeigt sie als `zombie` an.

```
  cpid = wait (&stat);        /* wait for a child to terminate */
  signal (SIGCHLD, waiter);   /* reinstall signal handler */
}


/*
 * main procedure of the fork based socket server
 */
void main(void)
{
  signal (SIGCHLD, waiter);   /* install signal handler */

  sockfd = socket ( ... );
  bind (sockfd, ... );
  listen (sockfd, ... );
    .
    .
} /* end of program */
```

5.2.6 Client zur verbindungsorientierten Kommunikation

Für den Client sind wesentlich weniger Schritte notwendig. Gemäß Abb. 6.1 auf Seite 43 legt er den Socket genau wie ein Server an. Der nächste Schritt ist bereits der Verbindungsaufbau mittels `connect()`.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

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

Vor dem Aufruf sind in `addr` die Adressdaten des gewünschten Server einzustellen, `addrlen` gibt – ähnlich wie bei `bind()` – die Länge der verwendeten Adress-Struktur an. `connect()` vergibt für die Client-Seite einen lokalen (dynamisch bestimmten) Port. Fehler werden durch den Return-Wert -1 angezeigt, die globale Variable `errno` spezifiziert dann die Ursache des Fehlers genauer. Zu beachten ist, dass – im Gegensatz zu `accept()` – hier lediglich ein Fehler-Code retourniert wird, der nachfolgende Datenaustausch erfolgt beim Client direkt mit dem Socket.

Der eigentliche Datentransfer geschieht wie beim Server z. B. mit den Funktionen `read()` und `write()`. Natürlich müssen Client und Server ein gemeinsames Applikationsprotokoll abwickeln. D. h. es muss klar sein, wer wann sendet bzw. empfängt.

Oft erhalten Clients die Adresse des Server durch die Kommandozeile oder durch ein GUI-Element vom Benuzter. Dann liegt diese Adresse intern als String vor. Meist wird statt der IP-Adresse der Domain-Name des Servers angegeben. Für die Konvertierung solcher Daten in eine von `connect()` akzeptierte Form ist die Funktion `gethostbyname()` sehr nutzbringend:

```
#include <netdb.h>

struct hostent *gethostbyname (char *Name);
```

Sie ermittelt zu einem Host-Namen (z. B. `jersey.fh-hagenberg.at`) unter anderem die IP-Adresse. `gethostbyname()` retourniert einen Zeiger auf eine Struktur:

```
struct hostent {
    char*   h_name;
    char**  h_aliases;
    int     h_addrtype;
    int     h_length;
    char**  h_addr_list;
};
#define h_addr h_addr_list[0]
```

Diese Felder haben die folgende Bedeutung:
`h_name` - offizieller Name des Host
`h_aliases` - ein NULL-terminiertes Feld von alternativen Namen
`h_addrtype` - der Typ der retournierten Adresse, in der Regel `AF_INET`
`h_length` - die Länge der retournierten Adresse
`h_addr_list` - ein NULL-terminiertes Feld von Netzwerkadressen des Host, bereits in der Darstellungsform des Netzwerkes
`h_addr` - die erste Adresse in `h_addr_list`

`gethostbyname()` retourniert einen Zeiger auf die vollständig ausgefüllte Struktur. Im Fehlerfall enthält der Zeiger den Wert NULL, dann ist die globale Variable `h_errno` entsprechend eingestellt. Zur Demonstration der Anwendung wird an dieser Stelle ein Programm von Hall [Hal] übernommen:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{
  struct hostent *h;

  if (argc != 2) {  /* error check the command line */
    fprintf(stderr,"usage: getip address\n");
    exit(1);
  }

  /*
   * get the host info
   */
  if ((h=gethostbyname(argv[1])) == NULL) {
    herror("gethostbyname");
    exit(1);
  }

  printf("Host name  : %s\n", h->h_name);
  printf("IP Address : %s\n",inet_ntoa(
    *((struct in_addr *)h->h_addr)
    ));

  return 0;
}
```

Ereignis	Beschreibung
POLLIN	Daten sind zu lesen
POLLPRI	hochpriore Daten sind zu lesen

Tabelle 5.1: Spezifizierbare Ereignisse für `poll()`

5.2.7 Client für mehrere Streams

Applikationen müssen manchmal gleichzeitig mehrere Streams überwachen und von diesen eingehende Daten lesen und verarbeiten. Eine Möglichkeit ist der *non blocking mode* von Streams. Ein `read()` retourniert dann sofort, auch wenn gerade keine Daten zu lesen sind. Nachteilig ist hier, dass dauernd zu lesen bzw. prüfen ist und dabei der Prozessor belastet d. h. nicht für andere Prozesse frei gegeben wird (*busy waiting*).

Deshalb existieren für das Überwachen mehrerer Streams Funktionen wie z. B. `poll()`:

```
#include <sys/types.h>
#include <poll.h>
#include <stropts.h>

int poll(struct pollfd *parray, ulong_t nfds, int timeout);
```

Der Rufer stellt die zu überwachenden Streams im Array `parray` zusammen, die Anzahl des Arrays steht in `nfds`, `timeout` gibt die maximale Wartezeit in Millisekunden an. Tritt binnen Ablauf der Timeout-Zeit kein Ereignis an einem der Streams auf, so retourniert `poll()` mit 0, sonst wird unmittelbar beim Auftreten mit der Anzahl der zu behandelnden Streams retourniert. -1 signalisiert wie üblich einen Fehler. Ein Timeout-Wert von -1 steht für ∞, der Wert 0 wartet gar nicht sondern testet lediglich die Streams. Die Struktur `pollfd` ist wie folgt definiert:

```
struct pollfd {
    int       fd;         /* file descriptor */
    short     events;     /* requested events */
    short     revents;    /* returned events */
};
```

Dabei ist vom Rufer in `fd` der File Deskriptor des Streams einzutragen, `events` ist ein Bit-Feld in dem die zu überwachenden Ereignisse einzutragen sind. In `revents` stehen nach dem Retournieren die davon tatsächlich aufgetretenen Ereignisse. Die Tabelle 6.1 zeigt einige sinnvolle Ereignisse, für eine vollständige Liste wird auf die man-Page von `poll()` verwiesen. Einige weitere Ereignisse können immer auftreten (vgl. Tab. 6.2 auf der nächsten Seite). Sie werden auch immer von `poll()` in `revents` gemeldet und sollten nicht in `events` angegeben werden.

Ein kleines Beispiel zeigt die Anwendung von `poll()`. Es koppelt einfach zwei – zuvor geöffnete – Streams d. h. es liest von einem und schreibt diese Daten auf den anderen Stream. Wenn ein Stream für eine geöffnete Socket-Verbindung und der zweite für `STDIN/STDOUT` verwendet wird, lässt sich damit ein einfaches Terminal realisieren. Dieses Beispiel wurde aus Rago [Rag93, S. 125–127] übernommen.

```
#include <poll.h>
#include <unistd.h>
```

Ereignis	Beschreibung
POLLERR	Stream meldet Fehler
POLLHUP	Stream ging verloren (hang up)
POLLNVAL	ungültiger File Deskriptor

Tabelle 5.2: Nicht maskierbare Ereignisse für `poll()`

```
extern void error(const char *fmt, ...);

void comm(int tfd, int nfd)
{
  int n, i;
  struct pollfd pfd[2];
  char buf[256];

  pfd[0].fd = tfd;  /* terminal */
  pfd[0].events = POLLIN;
  pfd[1].fd = nfd;  /* network */
  pfd[1].events = POLLIN;
  for (;;) {

    /*
     * Wait for events to occur.
     */
    if (poll(pfd, 2, -1) < 0) {
      error("poll failed");
      break;
    }

    /*
     * Check each file descriptor.
     */
    for (i = 0; i < 2; i++) {
      /*
       * If an error occurred, just return.
       */
      if (pfd[i].revents&(POLLERR|POLLHUP|POLLNVAL))
        return;

      /*
       * If there are data present, read them from
       * one file descriptor and write them to the
       * other one.
       */
      if (pfd[i].revents&POLLIN) {
        n = read(pfd[i].fd, buf, sizeof(buf));
        if (n > 0) {
          write(pfd[1-i].fd, buf, n);
        } else {
          if (n < 0)
            error("read failed");
          return;
        }
```

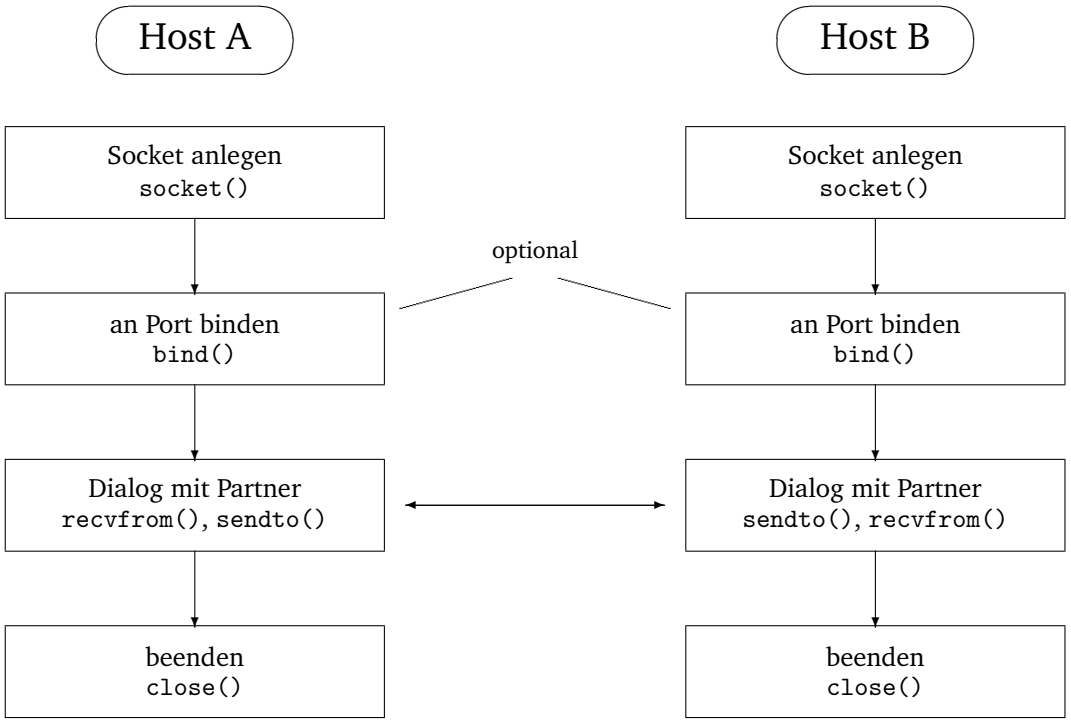

Abbildung 5.2: Verbindungslose Kommunikation

```
        }
      }
    }
}
```

Manchmal wird statt `poll()` auch die praktisch gleichwertige Funktion `select()` verwendet. Sie unterscheidet sich im Wesentlichen durch die Parameter, leistet aber prinzipiell das Gleiche. `select()` verwendet in der Parameterliste Mengen von Streams. Eine Menge ist intern als Bit-Array repräsentiert und hat damit eine konstante Länge, unabhängig von der Anzahl der Elemente. Damit eignet sich `select()` besser für Situationen in denen die zu überwachenden Streams fluktuieren. Das trifft besonders auf Server zu, die mit einem einzigen Prozess mehrere Clients servisieren. Details liefert die man-Page von `select()`, der Umgang ist z. B. in Hall [Hal, Kap. 6.2] beschrieben.

5.2.8 Verbindungslose Kommunikation

Die verbindungslose Kommunikation basiert auf dem Protokoll UDP. Damit ist die Übertragung nicht gesichert. Anwendungen, die auf UDP aufsetzen, sollten selber für die Sicherung und Quittierung sorgen. Die für die beteiligten Stationen notwendigen Aktionen sind in Abb. 6.2 dargestellt.

Zunächst wird ein Socket angelegt (`socket()`). Die Operation `bind()` ist eigentlich optional, sie dient lediglich dem Binden an einen BESTIMMTEN lokalen Port. Wird `bind()` nicht aufgerufen, dann wählt das Socket-Interface willkürlich einen freien Port. In der Regel übernimmt auch hier eine der beteiligten Stationen die Rolle des Servers. Dann ist

bei ihr das Verwenden von `bind()` sinnvoll, nur so kann der Client den Server-Prozess identifizieren.

Bei verbindungsloser Kommunikation entfallen die Aufrufe von `listen()`, `accept()` und `connect()`. Die Adressdaten der Partnerstation werden direkt beim Senden von Nachrichten angegeben:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto (
        int                     Socket,
        char*                   Message,
        int                     Length,
        int                     Flags,
        struct sockaddr*        To,
        int                     ToLength);
```

Die durch `Message` und `Length` spezifizierte Nachricht wird an den mit `To` und `ToLength` angegebenen Host gesendet. Broadcasts sind nur nach einem vorhergehenden Setzen der Option `SO_BROADCAST` (vgl. die Manual-Seite zu `setsockopt()`) möglich. Mit dem Parameter `Flags` können noch einige Attribute gesteuert werden (vgl. die Manual-Seite zu `sendto()`). Treten Fehler auf, dann retourniert `sendto()` mit dem Wert -1, `errno` ist dann wieder entsprechend eingestellt. Andernfalls retourniert diese Funktion die Anzahl der tatsächlich gesendeten Bytes. Wurden weniger Zeichen gesendet, als angegeben wurden, dann muss das Senden der verbleibenden Zeichen erneut durchgeführt werden.

Umgekehrt bekommt eine empfangende Station auch die Adressdaten des Senders übermittelt:

```
#include <sys/types.h>
#include <sys/socket.h>

int recvfrom (
        int                     Socket,
        char*                   Buffer,
        int                     Length,
        int                     Flags,
        struct sockaddr*        From,
        int*                    FromLength);
```

Hier werden maximal `Length` Bytes empfangen und in `Buffer` abgelegt. Die Adressdaten des Senders werden in `From` abgelegt. `From` muss beim Aufruf auf allokierten Speicher ausreichender Größe zeigen. Die Größe dieses Speicherbereiches ist in `FromLength` anzugeben. Ein Server erhält so die für eine ev. Antwort notwendigen Adressdaten. `recfrom()` retourniert die Anzahl der empfangenen Zeichen bzw. -1 im Fehlerfall. Auch `FromLength` wird auf die tatsächliche Größe der Adressstruktur eingestellt.

5.3 Socket-Programmierung in C unter Windows

Microsoft hat das Socket-Interface von Unix unter Windows in einer abgemagerten Version als *WinSocks* nachgebaut. Die wesentlichen Unterschiede sind:

1. Anstelle der üblichen Include-Dateien aus der Socket-Library wird lediglich das System-Include `winsock.h` verwendet.

2. Vor dem ersten Aufruf einer Socket-Funktion muss die richtige DLL geladen werden:

```
#include <winsock.h>

{
  WSADATA wsaData;  // if this does'nt work
  //WSAData wsaData;    // then try this instead

  if (WSAStartup(MAKEWORD(1, 1), &wsaData) != 0) {
    fprintf(stderr, "WSAStartup failed\n");
    exit(1);
  }
```

3. Dem Linker muss die richtige Library angegeben werden (`wsock32.lib` bzw. `winsock32.lib`).
4. Vor dem Terminieren einer Socket-Anwendung sollte `WSACleanup()` aufgerufen werden.
5. Die Rolle von `errno` übernimmt `GetLastError()`.
6. Windows Sockets sind nicht mit File-Deskriptoren kompatibel. Das hat die folgenden Konsequenzen:
   a) An Stelle von `close()` ist `closesocket()` zu verwenden.
   b) `select()` arbeitet nur mit Socket-Deskriptoren und nicht mit File-Deskriptoren.
   c) Die Funktionen `read()` und `write()` können für Windows-Sockets nicht verwendet werden. An ihrer Stelle sind `send()` und `recv()` zu benutzen. Diese beiden Funktionen existieren auch für das klassische Socket-Interface unter Unix, wegen ihrer Inkompatibilität zu den File-Deskriptoren werden sie dort aber eher selten verwendet.
7. `fork()` exitiert unter Windows nicht, an seiner Stelle ist `CreateProcess()` zu verwenden.

Detaillierte Informationen über die Socket-Programmierung unter Windows liefern z. B. Comer und Stevens [CS97].

Literatur

[CB94] William R. Cheswick und Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison–Wesely, 1994.

[CER] CERT Coordination Center. *Home Network Security*. zuletzt besucht im August 2022. URL: http://www.cert.org/tech_tips/home_networks.html.

[CS96] Douglas E. Comer und David L. Stevens. *Internetworking with TCP/IP, Vol. III, Client-Server Programming and Applications, BSD Socket Version*. 2. Aufl. Prentice-Hall, 1996.

[CS97] Douglas E. Comer und David L. Stevens. *Internetworking with TCP/IP, Vol. III, Client-Server Programming and Applications, Windows Sockets Version*. Prentice-Hall, 1997.

[CZ95] D. Brent Chapman und Elizabeth D. Zwicky. *Building Internet Firewalls*. O'Reilly, 1995.

[DH99] Naganand Doraswamy und Dan Harkins. *IPSec – The New Security Standard for the Internet, Intranets, and Virtual Private Networks*. Prentice-Hall, 1999.

[Hal] Brian Hall. *Beej's Guide to Network Programming – Using Internet Sockets*. zuletzt besucht im August 2022. URL: http://beej.us/guide/bgnet/.

[Mar00] Kai Martius. "DDoS Attacken auf internationale Websites". In: *iX* 4 (2000).

[Rag93] Stephen A. Rago. *UNIX System V Network Programming*. Reading Massachusetts: Addison Wesley, 1993.

[Sch96] Bruce Schneier. *Angewandte Kryptographie*. Addison-Wesley, 1996.

[Sec] SecurityFocus. zuletzt besucht im August 2022. URL: http://www.securityfocus.com/.

[Sof] Software Engineering Institute, Carnegie Mellon University. *Computer Emergency Response Team*. URL: https://www.sei.cmu.edu/about/divisions/cert/index.cfm (besucht am 31. 08. 2022).

[Str00] Stefan Strobel. "Distributed-Denial-of-Service-Angriffe". In: *iX* 8 (2000).

[SWE99] Charlie Scott, Paul Wolfe und Mike Erwin. *Virtual Private Networks*. 2. Aufl. O'Reilly, 1999.

[Til01] James S. Tiller. *A Technical Guide to IPSec Virtual Private Networks*. Auerbach, 2001.

[Zie02] Robert L. Ziegler. *Linux Firewalls*. 2. New Riders, 2002.