Hochschule Kempten

Diplomstudiengang Informatik (FH)

Diplomarbeit

zur Erlangung des Grades
Diplom-Informatiker (FH)

Steffen Wendzel
Matrikel-Nr. 184774

Protokollwechsel zur Realisierung von Covert Channels und Header-Strukturveränderungen zur Vermeidung von Covert Channels

Aufgabensteller	Prof. Dr. rer. nat. Arnulf Deinzer
Arbeit begonnen am	06.10.2008
Arbeit vorgelegt am	13.05.2009
durchgeführt in der	Fakultät Elektrotechnik und Informatik
Mailadresse des Verfassers	SteffenWendzel@gmx.de

Zusammenfassung

Aufgabenstellung

Diese Diplomarbeit befasst sich mit mehreren Unterthemen der verdeckten Kommunikationskanäle (Covert Channels) und möchte vor allen Dingen neue Themen vorstellen und diskutieren:

Erstmalige und detaillierte Behandlung von Protocol Hopping Covert Channels

Protocol Hopping Covert Channels sind Storage Channels die, während sie existieren, das Netzwerkprotokoll, in dem die versteckten Informationen untergebracht werden, wechseln.

Vorstellung der Idee der Protocol Channels

Im Gegensatz zu Protocol Hopping Covert Channels sind Protocol Channels schwerer zu detektieren, da sie ausschließlich durch den Wechsel eines Protokolls (ohne zusätzliche Informationen zu verstecken), versteckte Daten übertragen.

Sowohl für Protocol Hopping Covert Channels als auch für Protocol Channels beschreibt diese Arbeit deren jeweilige Technik und untersucht deren Detektionsmöglichkeiten.

Vorstellung der Idee der Header-Strukturveränderung

Ziel der Header-Strukturveränderung ist es, die Möglichkeiten, die Angreifer bei der Erstellung von Storage Channels innerhalb von Paket-Headern haben, einzugrenzen. Bei der Header-Strukturveränderung wird der Aufbau von Paket-Headern für jedes neu verschickte Paket verändert. Eine entsprechende Strukturinformation, die den Headeraufbau bestimmt, ist nur vertrauenswürdigen Komponenten beim Empfänger bzw. Sender zugänglich. Diese Arbeit stellt sowohl ein theoretisches Modell der Header-Strukturveränderung als auch eine praktische Umsetzung vor.

Ergebniszusammenfassung

Protocol Hopping Covert Channels sind im Vergleich zu normalen Storage Channels nur geringfügig schwerer zu detektieren, da sie letztlich auf die gleiche Weise Informationen übertragen. Durch den Wechsel des Übertragungsprotokolls sind die im Channel versteckt übertragenen Informationen jedoch schwerer mit forensischen Mitteln zu extrahieren, da nicht ein einziges, sondern alle verwendeten Protokolle detektiert werden müssen und der Forensiker zudem herausfinden muss, wie die versteckten Informationen in den verschiedenen Netzwerkprotokollen untergebracht wurden.

Protocol Channels hingegen können praktisch kaum detektiert werden. Die übertragenen Daten können auch mit forensischen Mitteln derzeit noch nicht rekonstruiert werden, wenn nicht explizit bekannt ist, wie diese Daten kodiert wurden. Diese Tatsache macht Protocol Channels zu einer sehr nützlichen neuen, verdeckten Kommunikationstechnik.

Die Header-Strukturveränderung ist für den Einsatz in der Praxis nur als bedingt tauglich zu erklären. Dies hängt damit zusammen, dass Covert Channels mit dieser Technik auch nur bedingt vermieden werden können, durch die Verwendung der Technik zusätzliche Rechenlast erzeugt wird und ein zusätzliches Synchronisationsproblem entsteht. Die Vorteile der Technik sind wiederum die leichte Erweiterungsmöglichkeit bestehender Netzwerkdienste und der mögliche Betrieb mit eingeschränkten Zugriffsrechten (gegenüber alternativen Techniken, die erweiterte Zugriffsrechte benötigen).

Weitere Hinweise

Für alle hier neu vorgestellten Techniken wurden Proof-of-Concept-Codes entwickelt, die dieser Diplomarbeit beiliegen. Die Quellcodes finden sich sowohl im Anhang als auch auf der beiliegenden CD. Entwickelt wurden alle Programme unter Linux mit C bzw. Perl. Die Wahl fiel auf diese beiden Programmiersprachen, weil sie ausgefeilte Zugriffsmöglichkeiten auf die (Low-Level-)Netzwerk-API (und damit direkte Manipulationsmöglichkeiten) bieten.

Auf der CD sind zudem Kopien aller aus dem Internet bezogenen Quellen zu finden.

Diese Diplomarbeit wurde mit LaTeX gesetzt, die Grafiken wurden mit OpenOffice.org Draw erstellt.

Vorwort

Verdeckte Kommunikationskanäle sind in Zeiten zunehmender Überwachung von Netzwerken ein Thema mit wachsender Bedeutung, dass immer wieder Raum für neue Ideen bietet. Die vorliegende Arbeit stellt mit den Protocol Channels und der Header-Strukturveränderung zwei neue Ideen vor.

Ich bedanke mich bei Prof. Dr. rer. nat. Arnulf Deinzer für die Betreuung der Diplomarbeit und die Möglichkeit, die Diplomarbeit hochschulintern verfassen zu können.

Inhaltsverzeichnis

Zusammenfassung II

Vorwort IV

Inhaltsverzeichnis V

Abkürzungsverzeichnis VIII

Symbolverzeichnis IX

Abbildungsverzeichnis X

Listings XI

Tabellenverzeichnis XII

1 Einleitung 1

1.1 Definition verdeckter Kommunikation 1
1.2 Abgrenzung zum Begriff "Tunnel" 2
1.3 Verwendung verdeckter Kommunikation 2
1.4 Arten verdeckter Kommunikationskanäle 2
1.4.1 (Nicht-)Lokale Covert Channels 3
1.4.2 Storage Channels und Timing Channels 4
1.4.3 Passive Covert Channels 5
1.4.4 Rauschfreie Covert Channels 5
1.5 Zusammenfassung 5

2 Protokollwechsel-basierte Covert Channels 7

2.1 Protocol Hopping Covert Channels 7
2.1.1 Erste Implementierung mit manueller Steuerung 8
2.1.2 phcct 10
2.1.3 Protocol Hopping Covert Channels mit Mikroprotokollen 10

2.1.4 Anpassungsfähige Covert Channels 14
2.1.5 Detektionsmöglichkeiten 15
2.2 Protocol Channels 21
2.2.1 Weitere Eigenschaften der Protocol Channels 22
2.2.2 Nutzungsprobleme 25
2.2.3 Proof-of-Concept-Implementierung 27
2.2.4 Vergleichbare Techniken 30
2.2.5 Detektion von Protocol Channels 31
2.3 Zusammenfassung 31

3 Header-Strukturveränderungen 33
3.1 Einleitung 33
3.2 Das theoretische Grundmodell 33
3.2.1 Gemeinsames Geheimnis 35
3.2.2 Sende- und Empfangsfunktion 36
3.2.3 Permutation der Header-Bestandteile 36
3.2.4 Zusammenfassung des grundlegenden Modells 36
3.3 Erweitertes theoretisches Modell 37
3.3.1 Kommunikation mit mehreren Systemen 37
3.3.2 Erweiterte Sende-/Empfangsfunktion 38
3.3.3 Funktionsweise der bidirektionalen Kommunikation 38
3.3.4 Beispiel: Ablauf einer simplen Kommunikation 40
3.3.5 Zusammenfassung des erweiterten Modells 40
3.4 Einführung eines Angreifers 41
3.4.1 Interner Angreiferprozess 41
3.4.2 Externes Angriffssystem 42
3.5 Praktische Umsetzung 44
3.5.1 Grundlegendes zur praktischen Umsetzung 44
3.5.2 Schutz der Verbindung vor Storage Channels 45
3.5.3 Proof-of-Concept-Implementierung 46
3.6 Reine Traffic-Normalisierung als Alternative 51
3.7 Probleme der Header-Strukturveränderung 52
3.7.1 Variable Headerlänge 52
3.7.2 False Positives 52
3.7.3 Zusätzliche Rechenlast 53
3.7.4 Keine vollständige Vermeidung von Covert Channels 53
3.7.5 Möglichkeit der Desynchronisation 53

3.7.6 Eingeschränktes Routing 53
3.8 Vorteile der Header-Strukturveränderung 54
3.8.1 Erweiterung bestehender Dienste 54
3.8.2 Betrieb bei eingeschränkten Zugriffsrechten 54
3.9 Zusammenfassung 55

A Quellcodelistings **56**
A.1 "phcc_detect"-Quellcode 56
A.1.1 phcc_detect.c 56
A.1.2 microproto_sim.pl 61
A.2 "pct"-Quellcode 63
A.2.1 pct_sender.pl 63
A.2.2 pct_receiver.c 67
A.3 "hsc"-Quellcode 72
A.3.1 secret.h 72
A.3.2 hsc_recv.c 73
A.3.3 hsc_send.c 79

Literaturverzeichnis **88**

Index **92**

Abkürzungsverzeichnis

AH	Authentication Header
DNS	Domain Name System
ESP	Encapsulated Security Payload
GRE	General Routing Encapsulation
HIDS	Host-basiertes Intrusion Detection System
HSV	Header-Strukturveränderung
HTTP	Hyper Text Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IP	Hiermit ist immer "*IPv4*" gemeint (→ IPv4)
IPSec	Internet Protocol Security
IPv4	Internet Protocol (Version 4)
IPv6	Internet Protocol (Version 6)
ISN	Initial Sequence Number
L2TP	Layer 2 Tunneling Protocol
NAPT	Network Address Port Translation
NAT	Network Address Translation
NIDS	Netzbasiertes Intrusion Detection System
PCC	Passive Covert Channel
PPTP	Point-to-Point Tunneling Protocol
TCP	Transmission Control Protocol
TTL	Time To Live
UDP	User Datagram Protocol

Symbolverzeichnis

A_n	Verfügbarer Platz für versteckte Daten in Protokoll n ($\rightarrow P_n$)
FE_x	Empfangsfunktion für Daten von System x bei Header-Strukturveränderung
FS_x	Sendefunktion für Daten an System x bei Header-Strukturveränderung
$G(x, y)$	Geheimnis der Header-Strukturveränderung zwischen System x und y
P_n	Protokoll n
PE	Empfangszeiger für Header-Strukturveränderung
PS	Sendezeiger für Header-Strukturveränderung
ψ_{Gesamt}	Insgesamt verfügbarer Platz für versteckte Daten
ψ_{Header}	Maximale Größe des Mikroprotokoll-Headers
$\psi_{Payload}$	Minimaler Platzbedarf für Payload

Abbildungsverzeichnis

2.1 Funktionsweise eines Protocol Hopping Covert Channels 8
2.2 Insgesamt für Storage Channels verwendbare Header-Bereiche 12
2.3 Beispielhafte Aufteilung von A_{ICMP}/A_{DNS} für Mikroprotokoll-Header und Payload 14
2.4 Beispiel eines Protocol Channels 22

3.1 Grundlage des theoretischen Modells der Header-Strukturveränderung . . . 34
3.2 Header-Strukturveränderung mit Geheimnissen und Geheimniszeigern. . . 35
3.3 Permutation eines Headers mit drei Bereichen 37
3.4 Erweitertes theoretisches Modell für Header-Strukturveränderungen 38
3.5 Zeigerbewegung im erweiterten theoretischen Modell 39
3.6 Interner Angreiferprozess bei Header-Strukturveränderung 42
3.7 Externes Angreifersystem bei Header-Strukturveränderung 43
3.8 Proof-of-Concept-Implementierung der Header-Strukturveränderung 47

Listings

2.1 Protocol Swapping im LOKI2-Code (Auszug) 8
2.2 Detektionsalgorithmus für Mikroprotokolle in Protocol Hopping Covert Channels (Pseudocode) 16
2.3 Aufruf von “phcc_detect” 20
2.4 Aufruf von “microproto_sim.pl” 20
2.5 Warnmeldungen von “phcc_detect” 20
2.6 Aufruf von pct_receiver 27
2.7 Aufruf von pct_sender.pl 28
2.8 Ausgabe von pct_receiver 29
3.1 Aufruf von hsc_send 49
3.2 Aufruf von hsc_recv 50
3.3 Telnet als POP3-Client für den HSV-Modifier “pct_send” 50
3.4 Versuch einen Storage Channel zu erzeugen (Client-seitige Ausgabe) 50
3.5 Versuch einen Storage Channel zu erzeugen (Server-seitige Ausgabe) . . . 51
A.1 Das Programm phcc_detect.c 56
A.2 Das Skript microproto_sim.pl 61
A.3 Das Skript pct_sender.pl 63
A.4 Das Programm pct_receiver.c 67
A.5 Das Programm secret.h 72
A.6 Das Programm hsc_recv.c 73
A.7 Das Programm hsc_send.c 79

Tabellenverzeichnis

1.1 Beispiel für Bitübertragung durch Vorhandensein einer Datei 3

2.1 Beispiel für Bitübertragung durch einen Protocol Channel 22
2.2 Besonders verbreitete Protokolle, die sich für Protocol Channels eignen und ihre Identifier für eingekapselte Protokolle. 23
2.3 Die von pct verwendete Kodierung. 28

3.1 Anzahl der Kombinationen pro Anzahl von Header-Bereichen 42

1 Einleitung

In dieser Einleitung sollen die Begriffe erläutert werden, auf die im weiteren Verlauf der Arbeit zurückgegriffen wird. Vorangestellt ist notwendiges Grundlagenwissen zu verdeckten Kommunikationskanälen.

1.1 Definition verdeckter Kommunikation

Verdeckte Kommunikation ist als Alternative zur Datenverschlüsselung zu betrachten. Dies ist genau dann der Fall, wenn es um die Übertragung von Informationen geht, die kein Dritter lesen können soll. Verdeckte Kommunikation *verschlüsselt* geheime Daten allerdings nicht, sondern *versteckt* diese. Eine Kombination aus verdeckter Kommunikation und Datenverschlüsselung ist jedoch möglich.[1]

Diese Arbeit setzt sich mit verdeckten Kommunikationskanälen (*Covert Channels*) in Netzwerken auseinander. In der Literatur findet man auch oft die Bezeichnung *Subliminal Channel* (wie durch [SIMMONS83] (S. 51) eingeführt). Die Unterscheidung zwischen Covert Channel und Subliminal Channel liegt im Detail, ist aber für diese Arbeit von Bedeutung: Während Subliminal Channels vom Entwickler eines Systems bzw. eines Schemas vorgesehen sind, ist dies bei Covert Channels nicht der Fall.[2]

[BISHOP02] definiert verdeckte Kommunikationskanäle explizit mit ähnlichen Worten, die auf die gleiche Bedeutung hinauslaufen: "*A covert channel is a path of communication that was not designed to be used for communication.*"[3]

In [SCHNEI06] heißt es: "*Die Steganographie hat den Zweck, Nachrichten in anderen Nachrichten zu verstecken, um die bloße Existenz einer geheimen Botschaft zu verbergen.*"[4] Gleicher Auffassung ist [LINGM02]: "*Bei der Steganographie werden Möglichkeiten gesucht und gefunden, wie geheime Daten in harmlos wirkenden Medien versteckt werden*

[1] Dies wird etwa am Beispiel der Proof-of-Concept-Implementierung "NUSHU" [RUTK04] demonstriert. Vgl. auch [ESSER05], S. 7.

[2] [BCKK05], S. 175: "*[...] a covert channel exists outside of the scheme/system design, while a subliminal channel is forseen by the scheme designer as hidden in a publicly known communication channel [...]*".

[3] [BISHOP02], S. 440.

[4] [SCHNEI06], S. 10.

können."[5] Dies beschreibt genau die Vorgehensweise verdeckter Kommunikationskanäle; sie sind somit eine Form der Steganographie.[6]

1.2 Abgrenzung zum Begriff "Tunnel"

Ein Netzwerktunnel ist nicht direkt mit einem Covert Channel gleichzusetzen. Tunneling bezeichnet die Einkapselung eines Protokolls in ein Transportprotokoll und ist nicht notwendigerweise zur versteckten Datenübertragung vorgesehen. Anwendung findet das Tunneling etwa beim AH- bzw. ESP-Protokoll von IPSec sowie anderen explizit auf Tunneling ausgelegten Protokollen wie PPTP, L2TP und GRE.[7]

1.3 Verwendung verdeckter Kommunikation

[SCHNEI06] nennt diverse Anwendungsmöglichkeiten verdeckter Kommunikation, etwa zum Informationsaustausch in einem Netzwerk von Spionen.[8] Verbreitet ist verdeckte Kommunikation heute auch in Botnetzen.[9]

Denkbar ist auch die Kommunikation zwischen unterdrückten Parteien innerhalb von bzw. zwischen Staaten mit Internetzensur. Während verschlüsselte Kommunikation ihre Existenz nicht verdeckt und somit von einer überwachenden Instanz gefunden werden kann (diese könnte dann die sendende und/oder empfangende Person mit Gewalt zur Kooperation zwingen), verhindert verdeckte Kommunikation, dass eine überwachende Instanz überhaupt von stattfindender Kommunikation erfährt. Somit kann verdeckte Kommunikation auch als Mittel zur eingeschränkten Wahrung von Grundrechten (etwa dem Austausch freier Meinungen) gesehen werden. Sofern eine weitere Zunahme an Zensur und Überwachung in Unternehmens-Intranets und auch im Internet stattfinden werden, ist daher besonders aus diesem Grund mit einer zunehmenden Bedeutung der verdeckten Kommunikationskanäle zu rechnen.

1.4 Arten verdeckter Kommunikationskanäle

Im Folgenden werde ich die im Rahmen dieser Arbeit bedeutsamen Charakteristika verdeckter Kommunikationskanäle erläutern.

[5] [LINGM02], S. 39.
[6] Bestätigt wird dies auch in [SCHNEI06], S. 95.
[7] Vgl. [ECKERT08], S. 677, 699f., 704; Vgl. dazu auch [ALEX06], S. 333.
[8] Vgl. [SCHNEI06], S. 97.
[9] Vgl. [LIGOCH08], S. 25.

Zeitpunkt	Datei vorhanden?	Empfangenes Bit
1	Ja	1
2	Ja	1
3	Nein	0
4	Ja	1

Tabelle 1.1: Beispiel für Bitübertragung durch Vorhandensein einer Datei

1.4.1 (Nicht-)Lokale Covert Channels

Ein verdeckter Kommunikationskanal kann entweder innerhalb eines Systems (also lokal) Daten übermitteln, oder – wie im Fokus dieser Arbeit – Daten in einem Netzwerk übermitteln. Der Vollständigkeit halber werde ich auch die lokale Variante skizzieren.

Lokale Kommunikationskanäle sind sehr vielfältig, schließlich muss nur auf eine beliebige Weise ein Bitstrom übermittelt werden und Betriebssysteme stellen dazu eine Unmenge an Möglichkeiten bereit. Hier einige Beispiele, bei denen jeweils ein Sendeprogramm P_1 Daten an ein Empfangsprogramm P_2 übertragen soll.

- P_1 sorgt dafür, dass zu jeder vollen Sekunde eine bestimmte oder auch mehrere Dateien vorhanden bzw. nicht vorhanden sind. Das Vorhandensein dieser Dateien wird von P_2 als Bitkombination interpretiert. Tabelle 1.1 stellt dieses Prinzip mit einer Einzeldatei exemplarisch dar. Dabei wird das Vorhandensein der Datei als 1er-Bit interpretiert, was selbverständlich nicht als allgemeine Vorgabe zu betrachten ist. [BISHOP02] beschreibt ein ähnliches Verfahren zum Datenaustausch über ein Protokoll mit Start- und Enddateien, die signalisieren, wann die Kommunikation beginnt, und wann sie endet.[10]

- P_1 sorgt durch eine rechenintensive Schleife (eine Pause) für eine hohe (niedrige) Prozessorauslastung. P_2 führt in periodischer Abfolge Überprüfungen der beanspruchten Rechenzeit von P_1 aus und erhält somit die gesendete Bitkombination. Alternativ kann auch die Ausführungszeit eines Programmes wechseln.[11]

- Analog könnten Veränderungen an den Zugriffsrechten von unbenutzten Dateien, an der Größe von Dateien usf. durchgeführt und interpretiert werden.

Nichtlokale Kommunikationskanäle können ebenfalls auf eine größere Anzahl verschiedener Modifikationen zugreifen. Auch hierzu einige Beispiele, bei denen Host H_1 Daten an Host H_2 übertragen soll.[12]

[10] Vgl. [BISHOP02], S. 451.
[11] Vgl. [ECKERT08], S. 4.
[12] Ungeachtet ist hierbei die Detektionswahrscheinlichkeit und Störanfälligkeit dieser Beispiele – sie dienen nur der Veranschaulichung der Möglichkeiten.

- Wie in [ZANDER06] beschrieben, würde H_1 an den nur wenige Hops entfernten H_2 ein IPv4-Paket senden, bei dem bestimmte Bits des "Time to Life"-Felds (TTL) im Header modifiziert werden, die auf einer kurzen Distanz (mit hoher Wahrscheinlichkeit) nicht verändert werden.[13]

- H_2 betreibt einen Webserver und H_1 sendet HTTP Request-Pakete mit gefälschtem Cookie-Wert an H_2.[14]

- H_1 sendet die Bitkombination durch eine künstliche Wahl der TCP Initial Sequence Number (ISN) an H_2.[15]

- H_1 schickt ICMP Echo-Response-Pakete an H_2, bei denen die versteckten Daten im Payload-Bereich untergebracht werden.[16]

1.4.2 Storage Channels und Timing Channels

Bishop definiert in [BISHOP02] Storage- und Timing Channels folgendermaßen: "*A covert storage channel uses an attribute of the shared resource. A covert timing channel uses a temporal or ordering relationship among accesses to a shared resource.*"[17] Der TCSEC-Standard enthält eine ähnliche Unterscheidung beider Channel: "*Covert storage channels include all vehicles that would allow the direct or indirect writing of a storage location by one process and the direct or indirect reading of it by another. Covert timing channels include all vehicles that would allow one process to signal information to another process by modulating its own use of system resources in such a way that the change in response time observed by the second process would provide information.*"[18]

Bezieht man diese Definition auf die Netzwerkkommunikation, so ist ein typischer Storage Channel demnach auch ein Covert Channel, der Header-Bestandteile eines Übertragungsprotokolls modifiziert, um darin versteckte Informationen unterzubringen. Ein Timing Channel würde hingegen die Sendereihenfolge von Netzwerkpaketen (Vgl. [AHSAN02], S. 53ff.) bzw. die Zeitdeltas zwischen Paketsendungen, wie in [ESSER05] beschrieben und implementiert, verändern.[19]

[13] Vgl. [ZANDER06], S. 3.
[14] Vgl. [CAST06], S. 50ff.
[15] Vgl. [RUTK04], S. 1ff.
[16] Vgl. [AHSAN02], S. 26. Im Text finden sich noch zahlreiche weitere Möglichkeiten, Covert Channels in Netzwerkpaketen unterzubringen.
[17] [BISHOP02], S. 446.
[18] [DOD85], S.80.
[19] Vgl. [ESSER05], S. 50.

1.4.3 Passive Covert Channels

Passive Covert Channels (PCC) in Netzwerken erzeugen im Gegensatz zu aktiven Covert Channels keinen eigenen Traffic, sondern modifizieren bestehenden Traffic.

2004 hat J. Rutkowska die Idee von passiven Covert Channels samt einer Beispielimplementierung vorgestellt. Sie definiert passive Covert Channels folgendermaßen: "*The idea is pretty simple – we do not generate our own traffic (i.e. packets) but only change some fields in the packets which are normally generated by the compromised computer.*" ([RUTK04], S. 1). Ihr NUSHU-Code[20] implementiert zu diesem Zweck eine zusätzliche TCP-Modifikationsschicht in den Linux-Kernel. Diese Modifikationsschicht ändert bei jedem Verbindungsaufbau die TCP Initial Sequence Number (ISN) und versteckt in Ihr die zu übertragenen Daten (die Daten werden zudem verschlüsselt).[21]

Dieses Verfahren konnte – unter Laborbedingungen – von S. J. Murdoch detektiert werden: "*Nushu encrypts data before including it in the ISN field. This will result in a distribution unlike that normally generated by Linux and so will be detected by the other TCP tests.*"[22].

1.4.4 Rauschfreie Covert Channels

[BISHOP02] definiert rauschfreie Covert Channels folgendermaßen: "*A noiseless covert channel is a covert channel that uses a resource available to the sender and receiver only. A noisy covert channel is a covert channel that uses a resource available to subjects other than the sender and receiver, as well as to the sender and receiver.*"[23] Bei den in dieser Arbeit behandelten Covert Channels handelt es sich *nicht* um rauschfreie Covert Channels, da sie ihr Übertragungsmedium gemeinsam mit anderen Teilnehmern nutzen. Die dadurch speziell bei *Protocol Channels* auftretenden Probleme sind auf Seite 25 entsprechend erläutert.

1.5 Zusammenfassung

Verdeckte Kommunikationskanäle (oder Covert Channels) werden mit dem Ziel errichtet, Informationen versteckt in öffentlichen Kanälen zu übertragen. Man unterscheidet dabei

[20] Der Code wurde übrigens nach der chinesischen Geheimschrift "Nü Shu" benannt, die von Frauen verwendet wurde.
[21] Als Alternative für die Erzeugung von TCP ISN-basierten Passive Covert Channels nennt Rutkowska zudem die Manipulation von HTTP Cookie-Werten.
[22] [MURD05], S. 256.
[23] [BISHOP02], S. 447.

zwei Kategorien von Covert Channels: Storage Channels, die Attribute übertragener Daten verändern und Timing Channels, die über die Veränderung zeitlicher Intervalle sowie die Ordnung von Daten versteckte Informationen übertragen.

Die Anwendungsgebiete verdeckter Kommunikationskanäle sind breit gefächert. Der geheime Austausch von Informationen in überwachten Netzwerken kann dabei sowohl von ungeduldeten Oppositionellen als auch von Personen, die über nicht geduldete Themen sprechen sowie von Botnetzen verwendet werden. Covert Channels sind somit ein Dual-Use-Gut.

2 Protokollwechsel-basierte Covert Channels

Dieses Kapitel beschäftigt sich mit der Realisierung von Covert Channels, die ihr Übertragungsprotokoll wechseln. Es gibt bisher zwei Arten von Covert Channels, die diese Technik einsetzen: **Protocol Hopping Covert Channels** und **Protocol Channels**.

Da über beide Techniken noch wenig bekannt ist (speziell über die hier erstmals vorgestellten **Protocol Channels** gibt es bisher keine anderen Arbeiten), beschreibt und analysiert diese Arbeit beide Techniken von Grund auf, wobei die Analyse der **Protocol Hopping Covert Channels** auf meinen bisherigen Analysen [WEND07] und [WEND08] aufbaut. Für die **Protocol Channels** ist das Ziel im Besonderen, die erstmalige und detaillierte Beschreibung sowie die Vorstellung einer Proof-of-Concept-Implementierung.

2.1 Protocol Hopping Covert Channels

Zunächst soll der Begriff des "Protocol Hopping Covert Channels" exakt definiert werden; anschließend werde ich diese Definition erläutern.

Definition 2.1.1. *(Protocol Hopping Covert Channel)* Protocol Hopping Covert Channels sind Storage Channels, die während der Übertragung von versteckten Daten das Übertragungsprotokoll wechseln. ■

Soll etwa die Bitfolge B über einen Protocol Hopping Covert Channel übertragen werden, und ist festgelegt, dass pro Paket des Protokolls P_n maximal A_n versteckte Bits übertragen werden können, dann läuft die Übertragung folgendermaßen ab:

1. Ein erstes Paket eines Protokolls P_1 wird verschickt. Es enthält von den zu übertragenden Daten maximal A_1 Bits. Falls die Bitfolge weniger Bits als A_1 beinhaltet, werden weniger Bits übertragen und die Übertragung ist beendet. Andernfalls wird Schritt 2 behandelt.

2. Ein weiteres Paket eines Protokolls P_2, wobei P_2 ein bereits verwendetes Protokoll sein kann, wird verschickt, dass die nächsten maximal A_2 Bits enthält.

3. Schritt 2 wird wiederholt, bis alle Bits der Bitfolge B übertragen wurden.

Ein konkretes Beispiel wird durch Abbildung 2.1 veranschaulicht. Hierbei wird ein Protocol Hopping Covert Channel mit drei Protokollen (HTTP, DNS und ICMP) verwendet und es können pro Paket jeweils vier Bits versteckt übertragen werden. Die zu übertragende Bitfolge "0100 1010 1010 1010" muss entsprechend auf vier Pakete aufgeteilt werden.

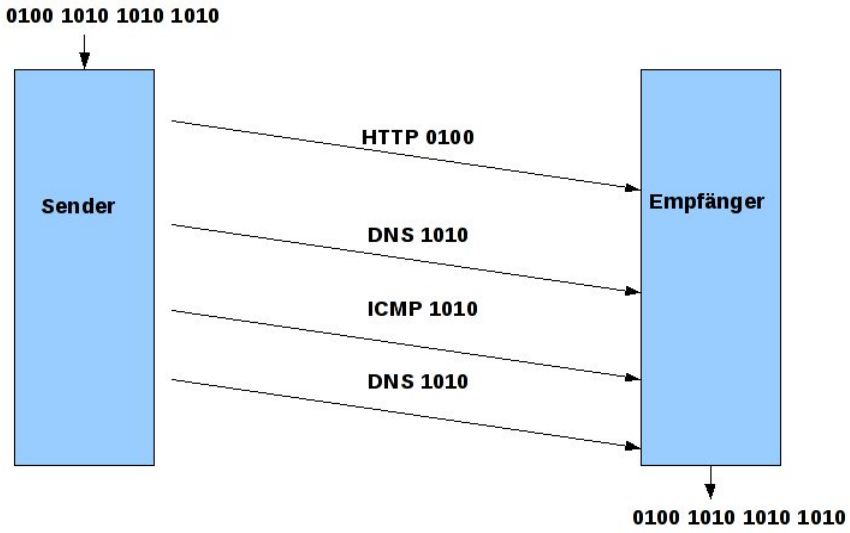

Abbildung 2.1: Funktionsweise eines Protocol Hopping Covert Channels

2.1.1 Erste Implementierung mit manueller Steuerung

Erstmalig (und bis zu meinem Proof-of-Concept-Code "phcct", auf den ich später eingehen werde, auch einzig) kam eine solche Technik im LOKI2-Code in primitiver Form zum Einsatz ([DAEM97]). Der Autor "daemon9" stellt darin einen Covert Channel Code vor, der sowohl über das ICMP- als auch über das UDP-Protokoll senden kann. Im Code wurde ein spezieller "/swapt"-Befehl implementiert. Wird dieser Befehl ausgeführt, dann wechselt das Programm das Übertragungsprotokoll. Das nächste Listing zeigt auszugsweise die entsprechenden Codezeilen des LOKI2-Codes; sobald im Empfangsbuffer der "/swapt"-String gefunden wird, sendet der Prozess ein eigenes Signal bei dessen Empfang die swap_t()-Funktion aufgerufen wird, die das Protokoll ändert.[1] Durch dieses Feature kann der LOKI2-Code als manuell steuerbarer Protocol Channel gezählt werden.

Listing 2.1: Protocol Swapping im LOKI2-Code (Auszug)

```
1 #define SWAP_T      "/swapt"        /* Swap protocols          */
```

[1] Die Zeilen im Listing wurden umgebrochen um eine bessere Darstellung auf dem Papier zu erzielen.

```
2 ...
3
4 int main(int argc, char *argv[])
5 {
6     ....
7     if (signal(SIGUSR1, swap_t) == SIG_ERR)
8         err_exit(1, 1, verbose, L_MSG_SIGUSR1);
9     ...
10 }
11
12 /*
13  *  Parse escaped commands (server-side version)
14  */
15
16 void d_parse(u_char *buf, pid_t pid, int ripsock)
17 {
18     ....
19     if (!strncmp(buf, SWAP_T, sizeof(SWAP_T) - 1))
20     {
21         if (kill(getppid(), SIGUSR1))
22             err_exit(1, 1, verbose,
23                 "[fatal] could not signal parent");
24         clean_exit(0);
25     }
26     ...
27 }
28
29 void swap_t(int signo)
30 {
31     ...
32     close(tsock);
33
34     prot = (prot == IPPROTO_UDP) ? IPPROTO_ICMP : IPPROTO_UDP;
35     if ((tsock = socket(AF_INET, SOCK_RAW, prot)) < 0)
36         err_exit(1, 1, verbose, L_MSG_SOCKET);
37     pprot = getprotobynumber(prot);
38     sprintf(buf, "lokid: transport protocol changed to %s\n",
39         pprot -> p_name);
40     fprintf(stderr, "\n%s", buf);
```

```
41        ...
42 }
```

Dass das Feature des Protokollwechsels so unbeachtet blieb, liegt meines Erachtens nach daran, dass der Autor selbst kaum etwas darüber schrieb und das Feature als unfertig bezeichnete: "*Swapping protocols is broken in everything but Linux. I think it has something to do with the Net/3 socket semantics. This is probably just a bug I need to iron out. Quite possibly something I did wrong. (...) Nevermind the fact that the server isn't doing any synchronous I/O multiplexing, consequently, swapping protocols requires a socket change on everone's part. This is why this feature is 'beta'.*" ([DAEM97]). Auch [GOLTZ03] geht bei seiner Untersuchung des LOKI2-Codes nicht auf das Feature des Protocol Swappings ein.[2]

2.1.2 phcct

Mit dem Proof-of-Concept-Code "phcct" hatte ich einen ersten automatischen Protocol Hopping Covert Channel implementiert. Gegenüber dem LOKI2-Code konnte "phcct" den Protokollwechsel transparent (also ohne Wissen des Anwenders) und selbstständig durchführen. Außerdem verwendet das Programm ein Mikroprotokoll zur Sortierung von eingehenden Paketen. Das Programm wurde in [WEND07] vorgestellt und in [BEJTLI07] analysiert und als detektierbar eingestuft. Die Detektierbarkeit von Protocol Hopping Covert Channels wird ab Seite 15 beschrieben.

2.1.3 Protocol Hopping Covert Channels mit Mikroprotokollen

Als Mikroprotokoll im Sinne verdeckter Kommunikation wird im Folgenden ein kleiner Daten-Header bezeichnet. Ein Mirkoprotokoll wird, wie alle anderen versteckten Daten auch, in den eigentlichen Daten versteckt. Er beinhaltet Informationen über den versteckten Payload eines Datenpaketes. Zur Weiterentwicklung von Protocol Hopping Covert Channels gegenüber der primitiven Variante aus dem LOKI2-Code wird aus zweierlei Gründen ein Mikroprotokoll benötigt:

1. Wird ein Protocol Hopping Covert Channel zwischen zwei Hosts aufgebaut, dann muss (es sei denn, der Angreifer akzeptiert eine gewisse Fehlertoleranz[3]) sichergestellt werden, dass die Kommunikation zwischen beiden Hosts fehlerfrei abläuft. Da bspw. UDP-basierte Protokolle keine Reliability-Funktionen nutzen können, wie sie

[2] Vgl. [GOLTZ03], S. 4ff.
[3] Etwa einen gewissen Prozentualen Anteil an falsch übertragenen, mitgelesenen Passwörtern.

TCP bereitstellt, wäre es wünschenswert, wenn der Covert Channel selbst solche Funktionen bereitstellen würde.

Die Besonderheit für Reliability-Funktionen innerhalb von Protocol Hopping Covert Channels stellt die Möglichkeit dar, Pakete des Protokolltyps X mit einem Paket eines anderen Protokolls Y zu bestätigen, was den Implementierungsaufwand ansteigen lässt.[4]

2. Mikroprotokolle sind außerdem die Voraussetzung zur Weiterentwicklung der manuellen Protokollwechsel von LOKI2 zu automatisierten Protokollwechseln durch eine integrierte Sequenznummer im Mikroprotokollheader. Selbst, wenn Pakete in der richtigen Reihenfolge verschickt werden, ist schließlich nicht sichergestellt, dass diese auch in der gesendeten Reihenfolge beim Empfänger ankommen. Bei einem Protocol Hopping Covert Channel kann auch TCP nicht für die richtige Paketreihenfolge garantieren, da TCP sich nur um die Sequenznummer *innerhalb einer Verbindung* kümmert, ein Protocol Hopping Covert Channel jedoch *mehrere* TCP-Verbindungen aufbauen kann, die unabhängig voneinander bestehen.

Aus diesen beiden Gründen lohnt sich die Implementierung eines Mikroprotokolls in einen Protocol Hopping Covert Channel.

Mikroprotokolle werden bereits seit Jahren zur Übertragung von Daten durch Covert Channels verwendet (etwa in "Ping Tunnel"[5]). Ein solches Mikroprotokoll muss mindestens die Paket-ID (die Bezeichnung "Sequenznummer" ist auch zutreffend) beinhalten – die Paketreihenfolge kann schließlich durch beabsichtigte falsche Sendereihenfolgen oder durch verschiedene Routingpfade durcheinander geraten.[6] "Ping Tunnel" implementiert zusätzlich zur Sequenznummer auch noch ein ACK-Feld für Bestätigungen, um Paketverlußte zu vermeiden.

Platzbedarf von Mikroprotokoll-Headern in PHCC ohne Payload

Für die Implementierung von Mikroprotokollen ist es sinnvoll, für jedes Protokoll das gleiche Mikroprotokoll zu verwenden (das senkt den Implementierungsaufwand, lässt Fehler leichter erkennen und steigert dadurch die Robustheit eines Protocol Hopping Covert Channels). Entsprechend ist die Größe eines statisch aufgebauten Mikroprotokoll-Headers

[4] Durch eine solche Technik wird Überwachung von Protocol Hopping Covert Channels zusätzlich erschwert.

[5] S. [STODLE09].

[6] Dass dieses rein Paket-ID basierte Verfahren bei ausschließlich TCP basierten Protokollen innerhalb von PHCC funktioniert, habe ich bereits durch den Proof-of-Concept-Code "phcct" ([WEND08]) belegt.

konstant. Zu diesem Zweck werde ich im Folgenden ein paar einfache Berechnungen und Symbole einführen.

Sei ψ_{Gesamt} die maximale Größe der Daten, die sich in einem Protocol Hopping Covert Channel in einem Paket *jedes* verwendeten Protokolls verstecken lassen und sei ψ_{Header} die maximale Größe des Mikroprotokollheaders. Dann gilt für einen Protocol Hopping Covert Channel *ohne* Payload, dass $\psi_{Gesamt} = \psi_{Header}$ ist, weil der gesamte Platz für den Mikroprotokoll-Header verwendet werden kann.

Das bedeutet anders formuliert, dass in diesem Fall ψ_{Header} der verfügbare Platz zur Unterbringung verdeckter Kommunikationsdaten ist, der in allen n verwendeten Protokollen (P_1 bis P_n) vorhanden ist.

Wenn A_n der verfügbare Platz in einem Protokoll P_n ist, dann gilt: $\psi_{Gesamt} = \psi_{Header} = Min(A_1, A_2, A_3, ..., A_n)$. Entsprechend ist A_n die Summe der einzelnen Bereiche, in denen Daten in P_n untergebracht werden können. Abbildung 2.2 veranschaulicht dies anhand eines Protokoll-Headers ohne Payload für ein gedachtes Protokoll P_n. Dabei können in Teilen der ersten zwei Header-Bereiche versteckte Daten untergebracht werden. Der insgesamt im Protokoll somit zur Verfügung stehende Platz ist der A_n-Wert dieses Protokolls. Wird zusätzlich noch Payload übertragen, so kann natürlich auch im Payload Platz zur Unterbringung verdeckter Daten auffindbar sein. Der A_n-Wert wird entsprechend angepasst.

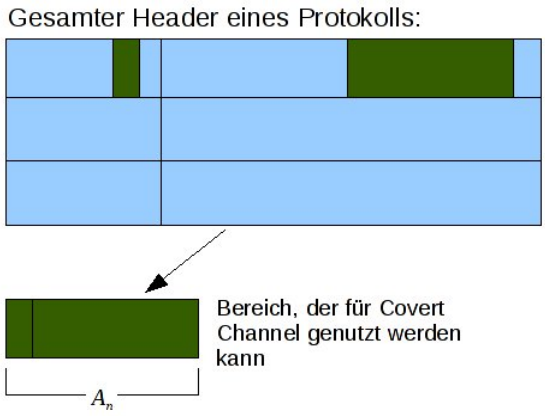

Abbildung 2.2: Insgesamt für Storage Channels verwendbare Header-Bereiche

Die Berechnung für ψ_{Gesamt} und die anderen besprochenen Werte ist nötig, weil der Mikroprotokoll-Header konstanter Größer aus dem oben genannten Grund in allen Protokollen, also auch in dem Protokoll mit dem geringsten zur Verfügung stehenden Platz, untergebracht werden muss. Ist es beispielsweise möglich im ICMP-Header ($A_{ICMP} = 12Bit$) und im DNS-Header ($A_{DNS} = 14Bit$) unterzubringen, dann stehen für den Mikroprotokoll-Header maximal 12 Bit zur Verfügung, da ein größerer Mikroprotokoll-Header nicht mehr

im ICMP-Header untergebracht werden könnte.[7]

Platzbedarf von Mikroprotokoll-Headern in PHCCs mit Payload

Da die Übertragung von Mikroprotokoll-Headern durch Protocol Hopping Covert Channels allein wenig sinnvoll ist, muss allerdings noch berechnet werden, wie groß der Header des Mikroprotokolls sein darf, wenn zusätzlich verdeckter Payload übertragen werden soll.

Dazu sei $\psi_{Payload}$ als der in jedem Fall minimal benötigte Platzbedarf für den Payload definiert. $\psi_{Payload}$ ist entsprechend mindestens ein Bit groß. Da der insgesamt verfügbare Platz ψ_{Gesamt} nun auch für den Payload benutzt wird, bleibt für den Mikroprotokoll-Header nur noch der restliche Teil von ψ_{Gesamt} übrig: $\psi_{Header} = \psi_{Gesamt} - \psi_{Payload}$. Folglich entspricht ψ_{Gesamt} bei Protocol Hopping Covert Channels *mit* Payload der Summe aus ψ_{Header} und dem Platzbedarf des Payloads $\psi_{Payload}$.

$$
\begin{aligned}
(2.1)\ \psi_{Gesamt} &= Min(A_1, A_2, A_3, ..., A_n) \\
(2.2) &= \psi_{Header} + \psi_{Payload}
\end{aligned}
$$

Das obige Beispiel mit den Protokollen ICMP und DNS lässt sich auch auf einen Protocol Hopping Covert Channel *mit* Payload anwenden: Es seien nach wie vor $A_{ICMP} = 12Bit$ und $A_{DNS} = 14Bit$. Außerdem werden mindestens 3 Bit für den Payload benötigt, also ist $\psi_{Payload} = 3Bit$. Abbildung 2.3 veranschaulicht dieses Beispiel. Dabei entsteht beim DNS-Protokoll ein ungenutzter Restbereich, der als zusätzlicher Payloadbereich benutzt werden kann. Die maximale Größe für einen Mikroprotokoll-Header kann nun folgendermaßen berechnet werden:

$$
\begin{aligned}
(2.3)\ \psi_{Header} &= \psi_{Gesamt} - \psi_{Payload} \\
(2.4) &= Min(A_{ICMP}, A_{DNS}) - 3Bit \\
(2.5) &= Min(12, 14) - 3Bit \\
(2.6) &= 12Bit - 3Bit \\
(2.7) &= 9Bit
\end{aligned}
$$

[7] Der Exaktheit halber muss erwähnt werden, dass sich in den meisten Protokollen, je nach Inhalt und Typ (etwa bei den ICMP-Typen "Destination unreachable" und "Echo Request") unterschiedlich viele Daten verstecken lassen. Diese Variationen müssen bei der Berechnung als eigene Protokoll-Werte (etwa $P_{ICMP-EchoReq}$ mit zugehörigem $A_{ICMP-EchoReq}$) behandelt werden. Der Verständlichkeit halber soll im Folgenden jedoch nur die Unterscheidung zwischen Protokollen, nicht aber zwischen ihren Typen und Varianten, durchgeführt werden.

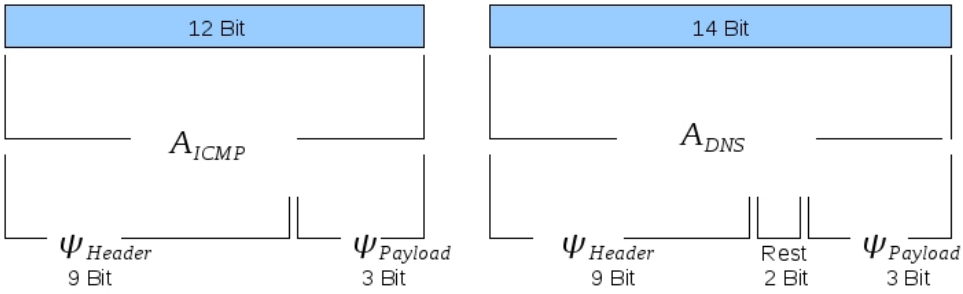

Abbildung 2.3: Beispielhafte Aufteilung von A_{ICMP}/A_{DNS} für Mikroprotokoll-Header und Payload

Verfügbarer Platz für den Payload in PHCCs mit Payload

Diese letzte Überlegung lässt sich auch umkehren, so dass man nicht die minimale Größe des Payloads, sondern die minimale Größe des Headers festlegt. Entsprechend würden in der obigen Rechnung bei einem minimalen Platzbedarf des Mikroprotokoll-Headers von 3 Bit beim ICMP-Protokoll $12 - 3 = 9$ Bit und beim DNS-Protokoll $14 - 3 = 11$ Bit für den Payload zur Verfügung stehen.

2.1.4 Anpassungsfähige Covert Channels

Im Dezember 2008, also innerhalb der Entstehungszeit dieser Arbeit, wurde in [YADALI08] eine den Protocol Hopping Covert Channels gleichzusetzende Idee vorgestellt, die letztlich aus eben diesem Grund nicht neu ist, da ich Protocol Hopping Covert Channels bereits mehr als ein Jahr früher vorgestellt hatte. Interessant und neu ist jedoch das besprochene Kommunikationsprotokoll zum Aufbau solcher Covert Channels. Dieses Kommunikationsprotokoll besteht aus zwei Phasen: In der ersten Phase (*Network Environment Learning Phase*) lernen die Systeme, die miteinander verdeckt kommunizieren sollen, die von den anderen Systemen sowie die vom eigenen System benutzten Netzwerkprotokolle kennen. Das System mit der Nummer n wird dabei mit A_n und die vom System verwendeten Netzwerkprotokolle mit P_n bezeichnet. Als gemeinsame Kommunikationsbasis wird anschließend die Schnittmenge der von zwei Hosts verwendeten Netzwerkprotokolle ausgesucht.

Benutzt System A_1 etwa die Netzwerkprotokolle $P_1 = \{DNS, HTTP, OSPF\}$ und das eigene System A_2 die Netzwerkprotokolle $P_2 = \{DNS, Telnet, HTTP\}$, so einigen sich beide Hosts auf eine gemeinsame Kommunikationsbasis, die sich aus der Schnittmenge $P_1 \cap P_2 = \{DNS, HTTP\}$ zusammensetzt. Des Weiteren wird von der gebildeten Schnittmenge die Menge der nicht gerouteten bzw. geblockten Netzwerkprotokolle (P_b) abgezogen, sodass letztlich $(P_1 \cap P_2) \notin P_b$ verwendet wird.

In der zweiten Phase des Protokolls, der *Communication Phase*, wird anschließend die Kommunikation über die in der ersten Phase ausgewählten Netzwerk-Protokolle abgewickelt. Parallel läuft der Prozess der Network Environment Learning Phase weiter, um Veränderungen in den Mustern der Protokollnutzung dynamisch zu erkennen und die Schnittmenge der verwendeten Protokolle ggf. anzupassen.

2.1.5 Detektionsmöglichkeiten

Intrusion Detection Systeme (IDS) wie Snort[8] können mit entsprechenden Filterregeln seit langem einige Varianten von Storage Channels detektieren. Da Protocol Hopping Covert Channels eine Form der Storage Channels darstellen, können auch diese – sofern sie entsprechend detektierbare Varianten von Storage Channels einsetzen – mit diesen Mitteln detektiert werden.

Der Unterschied für die Detektion besteht allerdings darin, dass Protocol Hopping Covert Channels mehrere Protokolle benutzen. Wird eines der benutzten Protokolle von einem IDS-System detektiert, so sind zu diesem Zeitpunkt keine anderen Protokolle des Channels von der Detektion betroffen und die Kommunikation läuft zum Teil weiterhin verdeckt ab.

Werden etwa vier Netzwerkprotokolle verwendet und wurde eines dieser Protokolle detektiert und wechselt der Covert Channel zudem im Round-Robin-Verfahren die Protokolle durch, so kann bei jedem vierten Paket mitgelesen werden. Dies bedeutet jedoch nicht automatisch, dass 25% des versteckten Traffics mitgelesen werden können. Begründet wird dies durch die Tatsache, dass die einzelnen Protokolle unterschiedlichen Platz für verdeckte Daten bieten können, wie auf Seite 14 erläutert wurde.

Detektion über Mikroprotokoll-Header

Da die Sequenznummern in Microprotokoll-Headern meist von Paket zu Paket um geringe Werte erhöht werden, kann mit diesem Wissen ein Mikroprotocol-Header von einer Detektionssoftware gefunden werden. Ich möchte folgenden simplen Algorithmus vorstellen, mit dem Microprotokoll-Header – und damit auch Protocol Hopping Covert Channels und andere Storage Channels, die in irgendeiner Form einen Mikroprotokoll-Header mit Sequenznummer verwenden – gefunden werden können. Dieser Algorithmus basiert auf meinen nicht umgesetzten Überlegungen aus [WEND07] und wird im Rahmen dieser Arbeit durch einen Proof-of-Concept-Code implementiert und einer kritischen Analyse unterzogen.

[8] www.snort.org

Funktionsweise des Algorithmus: Für jedes auf einem IDS-Gateway eintreffende Paket wird jedes vorherige Paket des gleichen Protokolls mit dem neu eingetroffenen verglichen. Dabei wird untersucht, ob es zwischen dem vorherigen und dem aktuellen Paket an einer Stelle im Header eine unübliche – d.h. an der jeweiligen Headerstelle vom Protokoll nicht vorgesehene – Wert-Erhöhung gibt. Findet der Algorithmus eine solche Erhöhung, geht er davon aus, die Sequenznummer eines Mikroprotokoll-Headers gefunden zu haben. Dementsprechend betrachtet der Algorithmus die Pakete eines Protokolls unabhängig von den Paketen anderer Protokolle eines Protocol Hopping Covert Channels. Das folgende Listing zeigt die Pseudocode-Darstellung des Algorithmus.

Listing 2.2: Detektionsalgorithmus für Mikroprotokolle in Protocol Hopping Covert Channels (Pseudocode)

```
1 # NextNormalStaticPartOfHeader gibt das jeweilig nächste Offset
2 # eines zu überprüfenden Teils des Headers zurück. Die Funktion
3 # merkt sich den zuletzt zurückgegebenen Header-Bereich und gibt
4 # (0, 0) zurück, falls alle Header-Bereiche durchgelaufen sind.
5 function NextNormalStaticPartOfHeader(Proto)
6 {
7  # Array mit Zeigern auf den nächsten zu überprüfenden Bereich
8  # eines Pakets. Alle Werte zunächst mit NULL (leer) intialisieren
9  static next_part = {NULL, NULL, ..., NULL}
10
11  switch (Proto) {
12  # Implementierung für ein erstes Protokoll
13  case PROTO_X:
14   switch (next_part[Proto]) {
15   default:
16    next_part[Proto] = PART_1
17    return (Offset_Part1, Len_Part1)
18    break;
19   case PART_1:
20    next_part[Proto] = PART_2
21    return (Offset_Part2, Len_Part2)
22    break;
23   case PART_2:
24    next_part[Proto] = FIN
25    return (Offset_Parte, Len_Parte)
26    break;
27   case FIN:
```

```
28      next_part[Proto] = NULL
29      return (0, 0)
30      break;
31    }
32    break;
33  # Implementierung für ein zweites Protokoll
34  case PROTO_Y:
35    switch (next_part[Proto]) {
36    default:
37      next_part[Proto] = PART_1
38      return (Offset_Part1, Len_Part1)
39      break;
40    case PART_1:
41      next_part[Proto] = FIN
42      return (Offset_Part2, Len_Part2)
43      break;
44    case FIN:
45      next_part[Proto] = NULL
46      return (0, 0)
47      break;
48    }
49    break;
50  # Abfangen nicht implementierter Protokolle
51  default:
52    error "Protocol not supported!"
53    exit
54  }
55 }
56
57 # Die FindInc-Funktion sucht nach unüblichen Inkrementierungen
58 # im Header des angegebenen Protokolls 'Proto'. Es vergleicht
59 # das vorherige mit dem aktuellen Paket und untersucht alle
60 # vorgegebenen Teilbereiche des Protokollheaders, die es von
61 # der Funktion NextNormalStaticPartOfHeader() erhält.
62 function FindInc(Proto, OldPkt, Pkt)
63 {
64  # Prüfe, ob das Paket zur gleichen Verbindung gehört. Falls
65  # das Protokoll Ports kennt, vergleiche auch den Quell- und
66  # Zielport. Falls die Protokolle andere verbindungsidentifi-
```

```
67 # zierende Eigenschaften haben, vergleiche diese ebenfalls.
68  if (OldPkt->Source != Pkt->Source
69   || OldPkt->Destination != Pkt->Destination
70   # Je nach Protokoll weitere identifizierende Vergleiche
71   # durchführen.
72   ... {|| OldPkt->DestinationPort != Pkt->DestinationPort}
73   ... {|| OldPkt->SourcePort != Pkt->SourcePort ...}
74   ) {
75   return;
76  }
77
78  # Jeden Bytebereich der Länge 'Len' am Offset 'Offset'
79  # mit dem vorherigen Wert vergleichen
80  while (((Offset, Len) = NextNormalStaticPartOfHeader(Proto))
81        != (0, 0)) {
82   OldVal = GetVal(OldPkt->Data + Offset, Len)
83   NewVal = GetVal(Pkt-Data + Offset, Len)
84   if (OldVal < NewVal) {
85    echo "Found unsual incrementation"
86   }
87  }
88 }
89
90 # OldPkt ist ein Array der Kopien der zuletzt eingetroffenen
91 # Pakete eines Protokolls. OldPkt[Protokoll] ist das letzte
92 # Paket eines Protokolls 'Protokoll'. Alle Werte zunächst mit
93 # NULL (leer) initialisieren.
94 OldPkt = {NULL, NULL, ..., NULL}
95
96 # Für jedes Protokoll wird das letzte Paket gesichert und mit
97 # einem neu eintreffenden über die FindInc-Funktion verglichen
98 while (Pkt = recv()) {
99  if (OldPkt[Pkt->Protocol]) {
100  FindInc(Pkt->Protocol, OldPkt[Pkt->Protocol], Pkt)
101  }
102  OldPkt[Pkt->Protocol] = Pkt
103 }
```

Analyse des Algorithmus: Der Algorithmus eignet sich für teilweise und vollständig statisch aufgebaute Protokoll-Header.[9] Für nichtstatische Protokollheader (etwa bei HTTP, in dem einzelne Header-Bestandteile vertauscht werden können (Vgl. [WONG00], S. 12ff. und [FIELD97], S. 31ff.)) wäre es (besonders in Hinblick auf die Performance bei hohem Traffic und die Rechenlast des Algorithmus) möglich, aber rechenintensiv, die einzelnen Offsets der zu überprüfenden Header-Bereiche zu ermitteln und zu überprüfen. Da es zudem enorm viele zu überprüfende Header-Bestandteile in den zahlreichen populären Netzwerkprotokollen gibt, wird der Algorithmus in der Praxis durch den Code-Bestandteil, der die Offsets der zu überwachenden Protokollbestandteile einzelner Protokolle ermittelt, äußerst komplex – schließlich benötigt jedes Protokoll eine eigene Implementierung, die diverse Sonderbedingungen überprüfen muss. Das es Sonderbedinungen gibt, zeigen zwei durchgeführte ICMP-Testfälle mit dem Proof-of-Concept-Code:

- Wird auf einem Host ein Ping auf einer lokalen Netzwerkschnittstelle durchgeführt, ist er also gleichzeitig Empfänger und Sender von Paketen, so würden "ausgesendete" Echo-Requests nicht von "eingehenden" Echo-Replies unterschieden. Der Wechsel des ICMP Types zwischen Echo Request/Response (Typ 8 und 0) führt zu einer permanenten False-Positiv-Detektion, da der Algorithmus denkt, es würde sich um eine versteckte Sequenznummer eines Mikroprotokolls handeln.[10]

- Host A pingt Host B an und Host B sendet einen Echo-Reply (ICMP Typ 0) zurück an Host A. Anschließend pingt Host B Host A an (ICMP Type 8) und Host A sendet den Echo-Reply zurück an Host B. Der Algorithmus sieht nun bei den von Host B verschickten Paketen die Erhöhung des ICMP-Type Feldes vom Wert 0 auf 8 und detektiert fälschlicher Weise eine Mikroprotokoll-Sequenznummer.

Um den Algorithmus zu umgehen, kann wiederum seitens des Angreifers eine Modifikation im Mikroprotokoll-Header implementiert werden. Dabei wird nicht die vermutete einfache Inkrementierung der ID durchgeführt, sondern eine Dekrementierung, wodurch der Algorithmus keine Inkrementierung mehr finden kann.

Die Proof-of-Concept-Implementierung: Die Proof-of-Concept-Implementierung "phcc_detect" besteht aus zwei Programmen: "phcc_detect.c" implementiert eine Detektion von Mikroprotokoll-Sequenznummern für die ICMP-Felder Type und Code. Das Perlskript

[9] Zu den teilweise statisch aufgebauten Headern zählen etwa ICMP, dass je nach ICMP-Typ eine unterschiedliche zweite Protokollhälfte bekommt ([POSTEL81A]), und auch IPv4, dass durch Optionen erweitert werden kann ([POSTEL81B]).

[10] Der Proof-of-Concept-Code umgeht dieses Problem, indem er Pakete, bei denen Quell- und Zieladresse übereinstimmen und es sich um einen Echo-Reply bzw. eine Echo-Response handelt, detektiert.

"micoproto_sim.pl" simuliert den zugehörigen Traffic (zunächst die Veränderung des Type-Wertes und dann die Veränderung des Code-Wertes). Benötigt werden das Perl-Modul Net::RawIP, sowie die Entwicklungspakete der Bibliothek libpcap für das C-Programm. Zum Ausführen werden die Rechte des root-Users benötigt:

Listing 2.3: Aufruf von "phcc_detect"

```
1 $ sudo ./phcc_detect eth0
2 RECEIVING MESSAGES - PRESS CTRL-C TO FINISH
```

Listing 2.4: Aufruf von "microproto_sim.pl"

```
1 $ sudo perl ./microproto_sim.pl 192.168.2.22 192.168.2.21
```

Listing 2.5: Warnmeldungen von "phcc_detect"

```
1 -init done-
2 FOUND A POTENTIAL MICRO PROTOCOL SEQUENCE NUMBER!
3 FOUND A POTENTIAL MICRO PROTOCOL SEQUENCE NUMBER!
4 FOUND A POTENTIAL MICRO PROTOCOL SEQUENCE NUMBER!
5 FOUND A POTENTIAL MICRO PROTOCOL SEQUENCE NUMBER!
6 FOUND A POTENTIAL MICRO PROTOCOL SEQUENCE NUMBER!
7 FOUND A POTENTIAL MICRO PROTOCOL SEQUENCE NUMBER!
8 FOUND A POTENTIAL MICRO PROTOCOL SEQUENCE NUMBER!
```

Abschließendes Fazit: Der Algorithmus kann nur nach ausgefeilten Tests der einzelnen Protokolle (und unter Berücksichtigung diverser Sonderfälle) in der Praxis eingesetzt werden, bietet jedoch eine erste Grundlage zur Detektion von Mikroprotokollen und damit auch eine erste Detektionsmöglichkeit, die speziell für Protocol Hopping Covert Channels ausgelegt ist.

Lokale Detektion

Wie alle Programme, sind auch Sende- und Empfangsprogramme von Covert Channels lokal sichtbar und wie bei allen Formen von Covert Channels können auch Protocol Hopping Covert Channels (besonders beim Empfänger) durch offene Sockets (speziell LISTEN-Sockets, die von hostbasierten Intrusion Detection Systemen (HIDS) detektiert bzw. von Programmen wie "netstat" angezeigt werden können) auffallen. Wird nicht direkt mit Sockets gerarbeitet, sondern etwa mit Packet Capturing auf dem Link-Layer, so wird die Netzwerkkarte *oftmals*[11] unbedacht in den Promiscuous-Modus geschaltet und somit ebenfalls für HIDS auffällig.

[11] Dies kann etwa bei libpcap konfiguriert werden.

An dieser Stelle sei angemerkt, dass Programme selbstverständlich wesentlich umständlicher und besser lokal versteckt werden können (etwa durch kernelseitige Manipulation von Syscalls, um Programmdateien und Prozesse nicht mehr im Dateisystem sichtbar zu machen oder bestimmte offene Sockets nicht mehr anzuzeigen[12]), doch dieses Thema weiter zu vertiefen würde vom eigentlichen Inhalt des Kapitels Abstand nehmen.

2.2 Protocol Channels

Den Protocol Hopping Covert Channels als eine Spezialform untergeordnet ist eine weitere Art von Covert Channels, die ich nun vorstelle. Ich bezeichne sie als *Protocol Channels*. Protocol Channels übertragen Informationen lediglich durch das von einem Datenpaket benutzte Protokoll.

Definition 2.2.1. *(Protocol Channel)* Ein Protocol Channel ist ein Storage Channel, bei dem Daten *ausschließlich* durch die Information des verwendeten Protokolls gesendeter Pakete übertragen werden (1). Ein Protocol Channel enthält *keine* statischen Identifikationsmerkmale (2). Bei den verwendeten Protokollen muss es sich um für das jeweilige Netzwerk typische, d.h. unauffällige, Protokolle handeln (3). ■

Diese Definition mit ihren drei Aussagen werde ich im Folgenden detailliert begründen und erläutern. Zunächst soll zu diesem Zweck die grundlegende Funktionsweise von Protocol Channels betrachtet werden um *Aussage (1) von Definition 2.2.1* zu erklären.

Im Protocol Channel wird verschiedenen Protokollen eine eindeutige Bitkombinationen zugeordnet. Jede Bitkombination entspricht genau einem Protocol, es können also nicht mehrere Protokolle der gleichen Bitkombination entsprechen.

Um eine Übertragung von Bits zu ermöglichen, werden mindestens zwei Zustände, also zwei unterschiedliche Protokolle, benötigt. So könnte etwa dem HTTP-Protokoll der erste Zustand (Bit: 0) und dem DNS-Protokoll der zweite Zustand (Bit: 1) zugeordnet werden. Optimaler Weise stehen mehr als zwei unterschiedliche Protokolle zur Verfügung, sodass ein Protokoll mehr als nur ein Bit repräsentiert. Bei vier Protokollen würden dementsprechend vier verschiedene Zustände pro zu übertragendem Datenpaket zur Wahl stehen, wodurch sich zwei Bits repräsentieren ließen. Tabelle 2.1 zeigt dies am Beispiel von vier Protokollen. Wendet man diese vier Protokolle nun zur Übertragung der Bitkombination “00 01 01 11” an, wie dies in Abbildung 2.4 veranschaulicht wird, so wird folgende Paketkombination an den Empfänger geschickt: HTTP, DNS, DNS, POP3.

[12] Vgl. [GASGOD06], S. 248 sowie [JONES01], S. 48.

Protokoll	Zustand	Übertragene Bits
HTTP	1	00
DNS	2	01
ICMP	3	10
POP3	4	11

Tabelle 2.1: Beispiel für Bitübertragung durch einen Protocol Channel

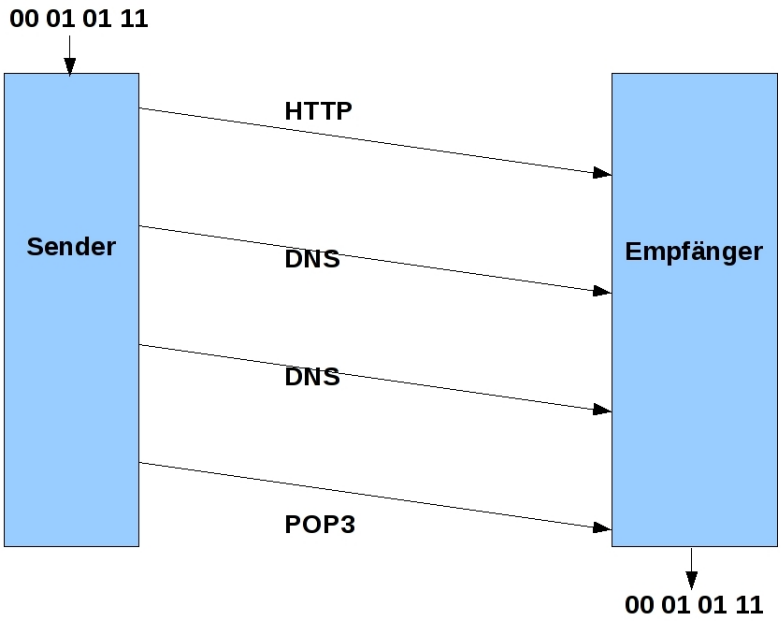

Abbildung 2.4: Beispiel eines Protocol Channels

2.2.1 Weitere Eigenschaften der Protocol Channels

Vergleicht man Protocol Channels mit typischen Storage Channels, etwa dem Verstecken von Daten im IPv4 TTL-Feld oder im Payload eines ICMP Echo-Requests, dann fallen die besonderen Eigenschaften von Protocol Channels auf, die ich im Folgenden erläutern werde. Dabei werde ich parallel auch die Aussagen (2) und (3) von *Definition 2.2.1* begründen.

Pakete sind nicht von regulären Paketen unterscheidbar

Protocol Channels manipulieren *ausschließlich* eine einzige Information (die des Protocol Identifiers) in Protokoll-Headern (Vgl. Tabelle 2.2). Andere Bereiche des Headers werden nicht manipuliert, somit ist auch kein Identifikationsmerkmal (etwa eine gefälschte Absender-Adresse oder ein festgelegtes Identifikations-Bit) im Protokollheader enthalten

Schicht	Protokoll	Feldbezeichnung
Network-Access	Ethernet	Ether-Type
Network-Access	Point-to-Point Protocol (PPP)	Protocol
Internet	IPv4	Protocol
Internet	IPv6	Next Header
Transport	TCP, UDP	Quell-/Zielport

Tabelle 2.2: Besonders verbreitete Protokolle, die sich für Protocol Channels eignen und ihre Identifier für eingekapselte Protokolle.

(*Aussage (2) von Definition 2.2.1*).[13]

Die manipulierte Header-Information stimmt anschließend exakt mit der Darstellung überein, die für das jeweilige Netzwerk unauffällig ist, da übliche Netzwerkprotokolle verwendet werden (*Aussage (3) von Definition 2.2.1*). Aussage (2) und (3) der Definition hängen also in ihrer Bedeutung voneinander ab, was ich noch etwas genauer erläutern möchte:

Während bei Protocol Channels die Protocol Identifier im Header eines Paketes immer auf die gleichen, für das jeweilige Netzwerk üblichen, Werte gesetzt werden, sorgen andere Varianten von Storage Channels zwar für ebenfalls sehr schwer zu detektierende Veränderungen (etwa IP TTL-Feld), also Veränderungen, die *Aussage (2) von Definition 2.2.1* nicht entsprechen, doch sind diese durchaus auffällig.[14] Kümmert sich der Benutzer des Protocol Channels nicht um diese Eigenschaft, also verwendet er unübliche Protocol-ID-Werte, dann geht diese besondere Eigenschaft verloren und die *Aussage (3) von Definition 2.2.1* ist nicht mehr erfüllt, weshalb es sich nicht mehr um einen echten Protocol Channel handelt. Entsprechend wäre der Channel über unübliche Werte im Protocol-Header detektierbar.

Aussage (2) und (3) bringen in Kombination für den Protocol Channel also überhaupt erst den eigentlichen Nutzen: Sie machen den Protocol Channel fast undetektierbar, da er sich nicht von regulärem Traffic unterscheiden lässt.

Implementierung auf verschiedenen Layern möglich

Ein Protocol Channel kann im Gegensatz zu festgelegten Verfahren (etwa Nutzung des ICMP Payloads, TCP ISN, Domain-Name bei DNS oder IPv4 TTL) auf verschiedensten Layern implementiert werden. Es muss nur im jeweiligen Protokoll-Header die ID für

[13] Der Protocol-Identifier dient dabei nicht als Identifikationsmerkmal, da er verändert werden muss, um Daten zu verstecken und zudem Aussage (3) von *Definition 2.2.1* gelten muss.

[14] Beispielsweise ist ein Storage Channel, der die IPv4 TTL benutzt, dadurch detektierbar, dass ein Empfänger eine TTL mit niedrigem Wert vorfindet, obwohl der Sender nur einen Hop entfernt und keine lange Alternativroute vorhanden ist.

das eingekapselte Protokoll ausgewertet werden können. Das bedeutet, dass ein Protocol Channel seine Information in jedem Protocol einbringen kann, das ein eingekapseltes Protokoll beinhaltet und dessen Typ angibt. Tabelle 2.2 zeigt beispielhaft die Werte, die ein Protocol Channel bei verschiedenen Protokollen manipulieren darf. Dabei ist zu beachten, das es nicht festgelegt ist, dass auf einem TCP- bzw. UDP-Zielport ein bestimmtes Protokoll verwendet wird, es könnte auch ein HTTP-Server auf dem Port des POP3-Protokolls Verbindungen entgegennehmen. Für einen Protocol Channel eignet sich die Portinformation jedoch trotzdem, da empfangsseitig nur ihr Wert ausgewertet wird und das tatsächlich verwendete Protokoll nebensächlich ist.

Indirekter Empfang verdeckter Informationen

Zum Senden/Empfangen von Informationen durch Protocol Channels muss nicht immer direkt auf die Protocol Identifier-Information des zugrunde liegenden Layers zugegriffen werden können. Es genügt ein indirekter Zugriff. Dies lässt sich durch folgendes Beispiel beweisen: Zwischen Sender und Empfänger werden zwei Protokolle (HTTP, Port 80 und UDP, Port 53) für den Protocol Channel verwendet. In diesem Fall kann der Empfänger einen Stream-Socket an Port 80 und einen Datagram-Socket an Port 53 binden. Der Betriebssystemkern kümmert sich um die Auswertung der Portnummern von TCP (die die eigentliche Information, also das verwendete Protooll, beinhalten) und der Empfänger wartet schlicht auf neu eintreffende Pakete, ohne direkten Zugriff auf die Portwerte in den Headern des Transport-Layers zu erhalten.[15] Der Sender kann analog mit solchen Sockets arbeiten.

Gleichzeitige Verwendung verschiedener Layer

Wie bereits erwähnt, kann ein Protocol Channel auf verschiedenen Layern implementiert werden. Daraus schließt sich eine weitere Eigenschaft der Protocol Channels: Ein Protocol Channel kann gleichzeitig Protokolle verschiedener Layer zur Informationsübertragung verwenden. Dazu muss der Empfängerprozess lediglich Paket Capturing auf verschiedenen Ebenen betreiben. Etwa kann der Sende-Prozess sowohl ICMP (die entsprechende Information ist dann im Internet-Layer (und zwar in der IP Protocol ID) enthalten) als auch HTTP (die Information ist dann im TCP Quell- bzw. Ziel-Port enthalten) benutzen.

[15] Dieses Verfahren erzeugt eine hostbasiert hohe Detektionswahrscheinlichkeit durch offene Sockets im System.

2.2.2 Nutzungsprobleme

Zwar bieten Protocol Channels ihren Anwendern einige wichtige Vorteile, frei von Problemen sind sie allerdings nicht. Es ergeben sich bei genauer Betrachtung sogar einige sehr bedeutsame Probleme.

Empfangsbestätigungen und andere Sicherungen

Durch den geringen Platz, den Protocol Channels für die Übertragung von Nachrichten bereitstellen (typischer Weise 1 oder 2 Bit), ist es nicht ohne Weiteres möglich, Korrekturinformationen unterzubringen. Hierzu zählt etwa eine Checksum, die Angabe von Nachrichtenlängen, und die Integration von Sequenznummern zur Sortierung von Paketen.

Der Anfang einer neuen Nachricht könnte theoretisch mit einer Preamble signalisiert werden, diese allerdings würde zusätzlichen Traffic und damit zusätzliche Aufmerksamkeit erzeugen, was für verdeckte Kommunikationskanäle nicht akzeptabel ist.

Die Bestätigung von Paketen mit ACK-Flags ist ebenfalls nicht möglich, da das Verhalten von Protocol Channels dabei analog zum Zwei-Armeen-Problem (also der Frage: “Ist die Bestätigungsnachricht angekommen, oder ging sie unterwegs verloren?”) ohne endgültige Sicherheit verlaufen würde (eine Sequenznummer hat schließlich ebenfalls kein Platz und so könnten bestätigte Pakete auch nicht identifiziert werden). Da Protocol Channels zudem keine Header-Bereiche als ACK-Flags benutzen dürfen, um *Definition 2.2.1* zu entsprechen, müsste die Bestätigung von Paketen äußerst minimal (nämlich etwa durch das Zurücksenden empfangener Pakete, die genauso verloren gehen können, wie die zuvor empfangenen Pakete) geschehen. Durch die damit einhergehende erhöhte Verlustwahrscheinlichkeit der Bestätigungspakete und die wiederum damit verbundene Zusatzgefahr einer Desynchronisierung (dazu in den nächsten Absätzen mehr), ist das Versenden von ACK-Paketen in Protocol Channels keineswegs nützlich.

Störungen durch eigentlichen Traffic (Rauschen)

Ein weiteres Problem stellt die *eigentliche Kommunikation* im Netzwerk, also die Kommunikation, die nicht zum Protocol Channel gehört, dar. Ein Empfänger von Protocol Channel Daten hat bisher keine Möglichkeit, zwischen Daten des Protocol Channels und denen der eigentlichen Kommunikation zu unterscheiden. Problematisch ist dies, da sobald ein Paket, das nicht vom Sender verschickt wurde, jedoch beim Empfänger als Teil des Protocol Channels empfangen wird (etwa, weil ein anderer Host ein Paket eines entsprechenden Typs an den Empfänger oder via Broadcast sendet), die komplette nachfolgende Kommunikation desynchronisiert, da ein unpassendes Bit die Reihenfolge der Folgebits

des Protocol Channels zerstört.[16] Eine Detektionsmöglichkeit von Übertragungsfehlern über ein Parity-Bit wird durch die Proof-of-Concept-Implementierung in Abschnitt 2.2.3 vorgestellt.

Zwar könnten Identifikationsmerkmale in den Header-Bereichen der verwendeten Protokolle untergebracht werden, um zugehörige Pakete eindeutig identifizieren zu können[17], doch dies würde mehr Aufmerksamkeit erzeugen (etwa: immer gleiches Feld in einer Nachricht, wie bei "Ping Tunnel") oder Abhängigkeiten aufbauen (etwa: die Quell-Adresse der Pakete muss immer Adresse X entsprechen); und es würde von *Definition 2.2.1* abweichen, d. h. es würde sich nicht mehr um einen Protocol Channel handeln.

Desynchronisierung durch Angreifer

Wie bereits besprochen, können Protocol Channels durch verlorene Pakete und durch fälschlicherweise als zugehörig interpretierte Pakete desynchronisiert werden. Würde ein Angreifer (ggf. durch nichttechnische Maßnahmen, wie Erpressung) in Erfahrung bringen können, dass zwischen Alice und Bob ein Protocol Channel aufgebaut wurde, so könnte er den Channel desynchronisieren, indem er Pakete in den Protocol Channel injiziert.

Ein solcher Angriff würde folgendermaßen ablaufen und wäre relativ einfach möglich: Der Angreifer muss alle Pakete eines Netzwerks für einen gewissen Zeitraum überwachen und dadurch herausfinden, welche Protokolle verwendet werden. Anschließend schickt er jeweils ein Paket jedes benutzten Protokolltyps in das Netzwerk (entweder über Broadcast oder an jeden einzelnen Host). Durch diesen Angriff wird ein im Überwachungszeitraum aktiver Protocol Channel zwangsläufig desynchronisiert.

Fragmentierung

Fragmentierung muss als weiteres Problem betrachtet werden. Sendet ein Host ein Paket des Protokolls X, dann besteht durch Fragmentierung die Möglichkeit, dass mehr als ein Paket gleichen Typs beim Empfänger ankommt. Als Folge dessen würde die Bitkombination des Paketes mehrmals hintereinander empfangen werden. Dieses kann durch Überprüfung auf fragmentierte Pakete seitens des Empfängers verhindert werden, vergrößert allerdings den Implementierungsaufwand.

Als Lösung für dieses Problem kann der Sender versuchen, möglichst kleine Pakete zu senden, die nicht fragmentiert werden. Im Falle von IPv4 kann zudem das "More

[16] Das gilt ebenso für die nicht empfangenen Pakete, die ein "Loch" in die Bitfolge reißen würden.

[17] Dies wird beispielsweise von dem Programm "Ping Tunnel" getan, dass eine "magic number" in Pakete einbaut (Vgl. [STODLE09]). Bei "Ping Tunnel" handelt es sich um einen Storage Channel ohne jeglichen Protokoll-Wechsel.

Fragments"-Flag überprüft werden und die zugehörige Indentification-Number für spätere Vergleiche zwischengespeichert werden (sodass doppelte Pakete aussortiert werden).[18] Alternativ kann der Empfänger auf einer höheren Abstraktionsebene arbeiten, die die Fragmentierung aufhebt (etwa im Application-Layer mit Stream-Sockets, an die das Betriebssystem nur defragmentierten Payload liefert).

2.2.3 Proof-of-Concept-Implementierung

Dass Protocol Channels auch umsetzbar sind, demonstriert der für diese Arbeit entwickelte Proof-of-Concept-Code "pct" (*Protocol Channel Tool*). Der Code besteht aus zwei Komponenten: *pct_sender.pl*, einem Perl-Skript, dass versteckte Daten versendet, und *pct_receiver*, einem C-Programm (aufbauend auf libpcap), dass Daten, die durch pct_sender.pl geschickt wurden, empfangen kann.

Das Programm initiiert einen unidirektionalen Protocol Channel zwischen zwei Hosts, die über Ethernet miteinander verbunden sind (auch über das Loopback-Device kann kommuniziert werden). Der Protocol Channel wird hierbei über zwei Protokolle (ICMP und ARP) umgesetzt. ARP-Pakete repräsentieren dabei ein 0er-Bit und ICMP-Pakete ein 1er-Bit. Um möglichst wenig Daten übertragen zu müssen und den Covert Channel damit besonders verdeckt zu halten, wurde eine 5-Bit-Kodierung der wichtigsten Zeichen (alle Großbuchstaben ohne Umlaute sowie einige Sonderzeichen) implementiert. Dies genügt zur Nachrichtenübertragung, da Kleinbuchstaben großgeschrieben; Umlaute durch AE, OE und UE; ß durch SS; und Zahlen durch Zahlwörter repräsentiert werden können.[19] Tabelle 2.3 stellt die Zeichenkodierung der Daten vor. Nach jedem fünften Bit (also jedem übertragenen Zeichen) wird ein Parity-Bit gesendet, um Fehler detektieren zu können.

Verwendung

Zunächst muss beim Empfänger-System "pct_receiver" gestartet werden. Übergeben wird die Bezeichnung der Netzwerkschnittstelle, auf der Daten entgegengenommen werden (etwa eth0):

Listing 2.6: Aufruf von pct_receiver

```
1 $ sudo ./pct_receiver eth0
2 RECEIVING MESSAGES - PRESS CTRL-C TO FINISH
```

[18] Vgl. [WASH98], S. 207. Im Falle von IPv6 ist zunächst zu überprüfen, ob als "Next Header" der "Fragment Extension Header" im Paket enthalten ist. Allerdings kommt erleichternd hinzu, dass IPv6-Pakete nicht mehr von Routern, sondern nur noch vom Sender fragmentiert werden (Vgl. [WASH98], S. 262f).

[19] Kleinbuchstaben werden vom Senderprogramm automatisch als Großbuchstaben übertragen.

Zeichen	Code	Zeichen	Code	Zeichen	Code	Zeichen	Code
A	00000	B	00001	C	00010	D	00011
E	00100	F	00101	G	00110	H	00111
I	01000	J	01001	K	01010	L	01011
M	01100	N	01101	O	01110	P	01111
Q	10000	R	10001	S	10010	T	10011
U	10100	V	10101	W	10110	X	10111
Y	11000	Z	11001	(Leerzeichen)	11010	_	11011
-	11100	$	11101	.	11110	,	11111

Tabelle 2.3: Die von pct verwendete Kodierung.

Nachdem dieser Schritt getan ist, muss das Sendeprogramm "pct_sender" gestartet werden. Es müssen folgende Parameter in dieser Reihenfolge übergeben werden:

- Netzwerk-Interface, über das verschickt werden soll
- Quell-IP-Adresse
- Ziel-IP-Adresse
- Quell-MAC-Adresse
- Ziel-MAC-Adresse
- Startwert für die ICMP-Sequenznummer[20]
- Die zu übertragende Nachricht.

Hinweis: Die CPAN-Module CPAN/Net::RawIP und CPAN/Net::ARP werden zur Ausführung des Codes benötigt. Außerdem werden root-Rechte vorausgesetzt, da ohne diese Rechte keine Rawsocket-Pakete verschickt werden können.

Dieser Beispielaufruf überträgt den String "Hallo". Dabei ist für jedes zu übertragene Zeichen die entsprechende Bitkombination zu sehen, die übertragen wird (hier für das Zeichen "H": 00111) und die Abfolge der einzelnen Pakete, die zu diesem Zweck verschickt werden (hier: ARP, ARP, ICMP, ICMP, ICMP):

Listing 2.7: Aufruf von pct_sender.pl

```
1 $ sudo perl ./pct_sender.pl eth0 192.168.2.22 192.168.2.21 \
2   00:1d:09:35:87:c4 00:17:31:23:9c:43 0x053c "Hallo"
```

[20] Ein immer gleicher Startwert würde das Programm leicht detektierbar machen, da nach dieser Sequenznummer von NIDS-Systemen Ausschau gehalten werden kann. Ein typischer Startwert für die ICMP-Sequenznummer auf Linux 2.6-Systemen ist 0x053c.

```
 3 sending payload[0]=H
 4 sending=00111
 5 sending bit 0=0 ARP
 6 sending bit 1=0 ARP
 7 sending bit 2=1 ICMP
 8 sending bit 3=1 ICMP
 9 sending bit 4=1 ICMP
10 Seqnr now=1343
11 ...
```

Beim Empfänger treffen nun die Pakete ein. Sobald der Benutzer STRG+C drückt, wird ein Signal ausgelöst, dass den Empfang beendet und den Empfangspuffer ausgibt. Die empfangenen Zeichen werden allerdings schon vorher dargestellt. Das nächste Listing zeigt die Ausgabe von "pct_receiver":

Listing 2.8: Ausgabe von pct_receiver

```
1   val=7 = H
2   val=0 = A
3   val=11 = L
4   val=11 = L
5   val=14 = O
6 ^CReceived signal 2
7 Received Message: HALLO
```

Sollte der 512-Byte große Empfangspuffer beim Empfänger zwischenzeitlich volllaufen, wird der bereits empfangene Text ausgegeben, der Empfangspuffer bereinigt und der Empfang weiterer Daten eingeleitet.

Analyse

Der Code ist anfällig für einen Teil der in Abschnitt 2.2.2 erläuterten Protocol Channel-typischen Nutzungsprobleme. Dazu gehört das Problem, verloren gegangene Pakete nicht erneut zu erhalten (aller anschließend empfangene Traffic ist unbrauchbar, da die Bitkombinationen durch Fehlen eines Bits verrückt werden) sowie das Problem des sonstig auf der empfangsseitigen Netzwerk-Schnittstelle vorhandenen Traffics, den der Empfänger als Teil des Protocol Channels interpretiert, wenn es sich um ARP- bzw. ICMP Type 8-Pakete handelt.

Um diese Fehler detektieren zu können, wurde der Implementierung von "pct" ein Parity-Bit hinzugefügt, dass nach jedem fünfstelligen Zeichen-Code als sechstes Bit übertragen wird. Beinhaltet der gesendete Code eine ungerade Zahl 0er-Bits, wird ein 1er-

Parity-Bit übertragen, anderenfalls ist das Parity-Bit 0. Sofern innerhalb der ersten fünf Bits eine ungerade Anzahl an Bits verfälscht wird, wird folglich, wenn das Parity-Bit nicht auch verfälscht wird, ein Fehler erkannt.

Von einer zweidimensionalen Parity-Prüfung wurde abgesehen, da die dafür zusätzlich zu übertragenen Bits ebenfalls verfällscht werden könnten und der zusätzliche Traffic (und die damit erhöhte Detektionswahrscheinlichkeit) nicht zielführend ist.

Natürlich könnten als Lösung des Problems auch Informationen, wie die Quell-Adresse ausgewertet und somit die Qualität des Channels verbessert werden, dann jedoch würde der Sender immer von der gleichen Adresse aus Daten schicken müssen. Dies ist durchaus eine vertretbare Einschränkung, widerspricht aber der *Aussage (2) von Definition 2.2.1* (es dürfen keinerlei feste Identifikationsmerkmale im Channel vorkommen).

Um dem ebenfalls angesprochenen Problem der Fragmentierung zu entgehen, ist die Entscheidung für die zu verwendenden Protokolle auf ARP- und kleine ICMP ECHO-Request-Pakete gefallen.

Beide Strategien, die Verwendung von Protokollen, die nicht fragmentiert werden, und die Verwendung des Parity-Bits, machen “pct” zu einem tatsächlich praktisch anwendbaren Programm. Problematisch ist jedoch die enorm schnelle Desynchronisierung der Verbindung, sobald ein ARP-Reply Paket eines anderen Hosts verschickt wird, ICMP ECHO-Request-Pakete sind hingegen nicht in so großer Zahl, wie ARP-Reply Pakete, anzutreffen. Starten Sender und Empfänger zeitlich abgestimmte Übertragungen, und unternehmen sie eventuell mehr als einen Versuch, dann sind die Chancen für eine erfolgreiche Datenübertragung hoch. Bezahlt wird dieser Nutzungsgrad jedoch auch mit der verloren gegangenen Routing-Fähigkeit der ARP-Pakete.

2.2.4 Vergleichbare Techniken

Es gibt einen gleichzeitig zu dieser Diplomarbeit entwickelten Ansatz von Tyler Borland, der dem des Protocol Channels recht ähnlich ist. Beschrieben wurde dieser Ansatz in [BORLAND08]. Der Covert Channel überträgt dabei Zeichen, die durch die Wahl der Quell-/Zielports von TCP bzw. UDP repräsentiert werden. Borland verwendet zur Implementierung des Covert Channels das Programm ”hping3”.

Da dieser Channel jedoch Identifikationsmerkmale verwendet (Kennzeichnung des Payloads durch den ”–sign”-Parameter) bzw. andere Header-Bereiche als die Portnummern verändert (nämlich die IPv4-TTL), entspricht der Covert Channel nicht den ersten beiden Sätzen von Definition 2.2.1.

2.2.5 Detektion von Protocol Channels

Detektion im Netzwerk

Sofern nur typische Protokolle für Protocol Channels genutzt werden, ist eine netzwerkseitige Detektion von Protocol Channels nach bisherigen Erkenntnissen nicht mit automatischen Mitteln möglich. Dies begründet sich durch die Tatsache, dass die Protocol-IDs in den einkapselnden Headern entsprechend unauffällige Werte beinhalten. Würden untypische Protokollwerte verwendet werden, würde der Covert Channel hingegen auf sich aufmerksam machen (und wäre nicht mehr definitionsgemäß). Entsprechend wäre auch die besondere Tarnung der Protocol Channels aufgegeben, die sie von anderen Storage Channels und normalen Protocol Hopping Covert Channels unterscheidet.

Sofern ein Protocol Channel dennoch detektiert werden würde, wäre noch das Darstellungsproblem zu lösen. Das bedeutet, dass die Instanz, die die verdeckten Informationen mitlesen will, mehrere Dinge in Erfahrung bringen muss:

- Welche und wieviele Protokolle sind im Protocol Channel eingesetzt?
- Welches Protokoll steht für welche Bitkombination?
- Wie werden die Bitkombinationen interpretiert? (Little-, Middle- oder Big-Endian? (Minimalisierte/Abgewandelte) ASCII-Kodierung? etc.)

Lokale Detektion

Es gelten die gleichen *lokalen* Detektionsmöglichkeiten, wie für Protocol Hopping Covert Channels (siehe Seite 20).

2.3 Zusammenfassung

In diesem Kapitel wurden zwei verschiedene Möglichkeiten vorgestellt, Covert Channels zu erzeugen. Zum einen handelt es sich dabei um “Protocol Hopping Covert Channels”, die bereits in der Fachwelt bekannt sind. Sie sind Storage Channels, die während der Datenübertragung ihr verwendetes Netzwerk-Protokoll wechseln. Zum anderen handelt es sich dabei um eine neu vorgestellte Technik namens “Protocol Channels”. Protocol Channels modifizieren einzig die Protocol ID eines Paket-Headers und sind durch diese minimale Veränderung, die nach völlig legitimem Traffic aussieht, kaum zu detektieren.

Protocol Hopping Covert Channels verwenden zur Synchronisation einzelner Pakete oft einen Mikroprotocol-Header, der eine ID enthält, um ein empfangenes Paket seinem

korrekten Platz im Empfangspuffer zuzuweisen. Über diese Mikroprotocol-ID sind sie detektierbar. Ein entsprechender Algorithmus wurde in diesem Kapitel vorgestellt. Sowohl Protocol Hopping Covert Channels als auch Protocol Channels können über die Verwendung unüblicher, also auffallender, Netzwerkprotokolle detektiert werden.

3 Header-Strukturveränderungen

In diesem Kapitel wird untersucht, inwiefern durch Veränderungen der Headerstruktur von Netzwerkpaketen die Vermeidung/Eindämmung von Storage Channels möglich ist. Ich werde zunächst die Grundidee der Header-Strukturveränderung vorstellen und anschließend deren praktische Umsetzung beschreiben.

3.1 Einleitung

Als Header-Strukturveränderung bezeichne ich in diesem Kapitel die durch eine geheime Information bestimmte Umstrukturierung des Header-Bereichs von Netzwerkpaketen. Normalerweise verfügen Netzwerkpakete über einen Header, dessen Aufbau durch einen Standard vorgegeben ist. Bei einer Header-Strukturveränderung wird dieser Aufbau gezielt verfälscht.

Ziel von Header-Strukturveränderungen ist es, Storage Channels innerhalb von Protokoll-Headern in Netzwerkpaketen zu verhindern. Zu diesem Zweck wird für jedes Netzwerkpaket der Aufbau des Headers nach einer geheimen Information neu gestaltet. Nur den jeweiligen Sende-/Empfangsfunktionen der kommunizierenden Systeme ist diese geheime Information bekannt. Prozessen, die eine solche Sende-/Empfangsfunktion nutzen (und damit auch Angreifer-Prozessen) darf die geheime Information nicht zugänglich sein, damit Angreifer nicht in der Lage sind, gefälschte Pakete, die Storage Channel-Daten enthalten, zu generieren. Im Folgenden werde ich das theoretisch zu Grunde liegende Modell einführen. Dabei wird zunächst ein vereinfachtes Modell, und anschließend ein erweitertes Modell vorgestellt.

3.2 Das theoretische Grundmodell

Ein unidirektionaler Kommunikationskanal soll zwischen den zwei Systemen A und B aufgebaut werden. Dabei soll A der Sender, und B der Empfänger von Informationen sein. Des Weiteren gelten für dieses Modell folgende Grundsätze:

- Informationen werden über Netzwerkpakete, die über Header verfügen, von System A nach System B übertragen.
- Es gibt nur eine Übertragungsschicht (einen Layer).
- Es gibt nur eine permanente Verbindung.
- Pakete können nicht verloren gehen.
- Pakete kommen in der Reihenfolge beim Empfänger an, wie sie vom Sender verschickt wurden.
- Pakete können nicht dupliziert werden.
- System A und B verfügen über ein gemeinsames Geheimnis.
- Das Geheimnis bestimmt den Aufbau der Header von Netzwerkpaketen.
- Zum Senden von Paketen muss eine vorgegebene Sendefunktion, zum Empfangen eine vorgegebene Empfangsfunktion benutzt werden.
- Nur Sende-/Empfangsfunktion haben Zugriff auf das Geheimnis. Dadurch ist nur die Sendefunktion in der Lage, Pakete mit korrektem Aufbau zu generieren.

Abbildung 3.1 stellt das bisherige Modell vor, wobei P1, P2 und P3 Pakete, die sich in der Übertragung befinden, darstellen.

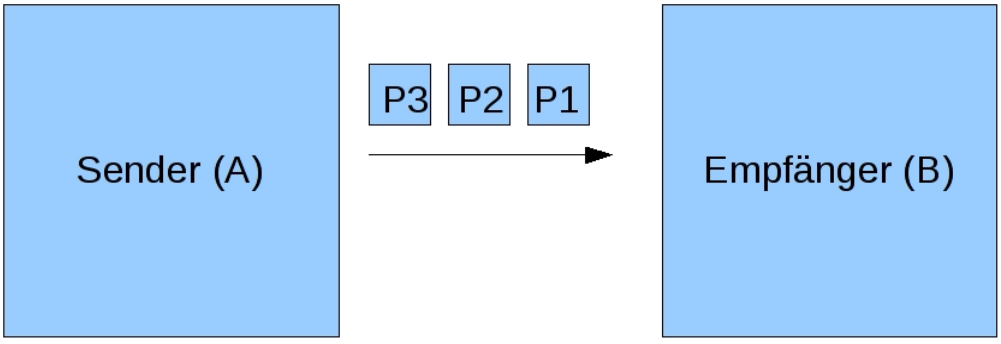

Abbildung 3.1: Grundlage des theoretischen Modells der Header-Strukturveränderung

3.2.1 Gemeinsames Geheimnis

System A und B haben beide Zugriff auf eine Kopie eines gemeinsamen Geheimnisses G. Dieses Geheimnis stellt die Information dar, die den Aufbau der zu sendenden bzw. zu empfangenen Header beschreibt. Das Geheimnis ist unendlich lang.

Werden für den Aufbau eines Headers n Informationseinheiten vom Geheimnis benötigt, so liest System A diese n Informationseinheiten und baut gemäß diesen den Header des Paketes auf. Er schickt das Paket an System B, dass die Daten entgegennimmt. B liest ebenfalls n Informationseinheiten des eigenen Geheimnisses und interpretiert mit Hilfe dieser Informationen den Header des übertragenen Paketes.

Beide Systeme besitzen einen Zeiger. Dieser zeigt auf eine Stelle des eigenen Geheimnisses. Zu Beginn ist dieser Zeiger auf eine unbestimmte, jedoch auf System A und B gleiche, Stelle des Geheimnisses gerichtet. Werden n Informationseinheiten vom Geheimnis gelesen, so muss der Zeiger um diese n Informationseinheiten vorwärts bewegt werden, sodass er auf die direkt folgenden Informationseinheiten zeigt.

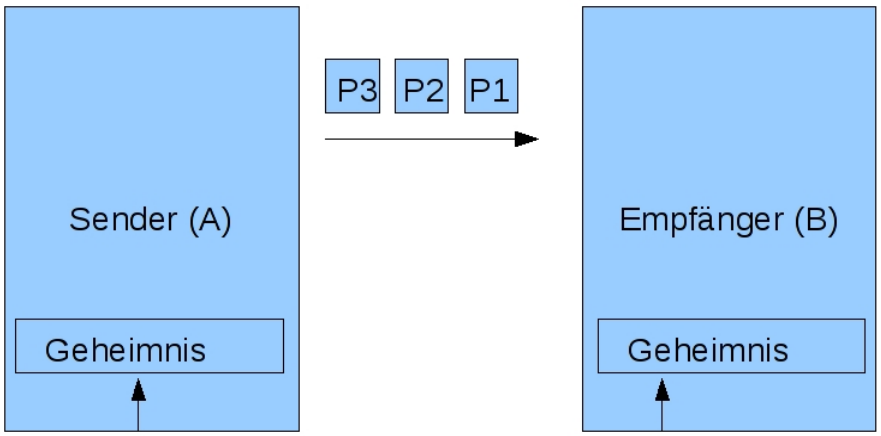

Abbildung 3.2: Header-Strukturveränderung mit Geheimnissen und Geheimniszeigern.

Auf beiden Systemen wird der Zeiger einzeln verwaltet. Daher können beide Zeiger zum gleichen Zeitpunkt auf eine vom anderen System verschiedene Position des Geheimnisses zeigen. Abbildung 3.2 veranschaulicht dies, wobei die drei vom Sender verschickten Pakete noch nicht empfangen wurden. Deshalb ist der Geheimniszeiger des Empfängers noch nicht so weit vorgestellt, wie der des Senders.

Ein konkretes Zwischenbeispiel soll dies verdeutlichen: Die Zeiger von System A und B zeigen beide auf den Wert 4. Sendet System A Informationen, und benötigt dazu 3 Informationseinheiten, setzt er den eigenen Zeiger auf die Position 7 (= 4 + 3 Informationseinheiten). Bis zu diesem Zeitpunkt muss bei B der Zeiger noch auf dem Wert 4 stehen. Erst nach dem Empfang und dem Lesen der notwendigen 3 Informationseinheiten steht

auch der Zeiger von System B auf dem Wert 7. Würden für das nächste Paket wieder 3 Informationseinheiten gelesen werden, so würden nach dem Senden/Empfangen die Zeiger beider Systeme auf den Wert 10 zeigen.

3.2.2 Sende- und Empfangsfunktion

System A hat eine Sendefunktion und System B eine Empfangsfunktion. Die Sendefunktion ist die einzige Komponente, die Zugriff auf das Geheimnis (lesend) und auf den Geheimnis-Zeiger (lesend/schreibend) von System A hat. Die Empfangsfunktion, ist die einzige Komponente, die Zugriff auf das Geheimnis (lesend) und den Geheimnis-Zeiger (lesend/schreibend) von System B hat.

Die Sendefunktion bekommt den zu sendenden Payload übergeben und generiert daraufhin den Header (basierend auf den Informationseinheiten des Geheimnisses) für das zu sendende Paket, hängt diesem den Payload an und verschickt das Paket.

Die Empfangsfunktion nimmt die Netzwerkpakete entgegen, interpretiert die Headerinformationen gemäß den Informationseinheiten des Geheimnisses, und gibt den Payload zurück.

Beide Funktionen manipulieren nach dem Lesen des Geheimnisses den Geheimnis-Zeiger entsprechend der vorherigen Beschreibung aus Abschnitt 3.2.1.

3.2.3 Permutation der Header-Bestandteile

Der Aufbau von Headern wird bei der Header-Strukturveränderung, wie erwähnt, durch eine Sendefunktion getätigt, die zu diesem Zweck einen Teil des Geheimnisses ausliest, auf den der Geheimniszeiger zeigt. Die gelesene Information aus dem Geheimnis repräsentiert dabei eine bestimmte Würfelung der Header-Bestandteile.

Mathematisch betrachtet stehen für einen Header mit n Bestandteilen (analog ist dies eine Menge mit n Elementen) $n!$ Kombinationsmöglichkeiten zur Verfügung. Man spricht in diesem Zusammenhang von der Permutation einer Menge.[1]

Abbildung 3.3 zeigt eine Permutation für einen Header mit $n = 3$ Elementen und entsprechend $n! = 6$ möglichen Kombinationen.

3.2.4 Zusammenfassung des grundlegenden Modells

Die Header-Strukturveränderung funktioniert grundlegend also folgendermaßen: System A möchte ein Datenpaket senden. Dazu gibt ein Prozess den Payload an die Sendefunktion des Systems weiter. Die Sendefunktion liest die notwendige Information zum Aufbau des

[1] Vgl. [BRILL04], S. 49f.

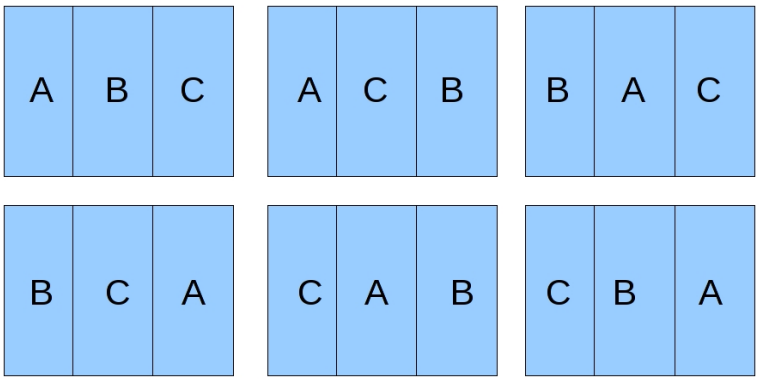

Abbildung 3.3: Permutation eines Headers mit drei Bereichen

Headers vom Geheimnis, generiert das gesamte Paket und schickt es an den Empfänger. Um beim nächsten Sendevorgang die nächste Aufbauinformation vom Geheimnis zu lesen, wird der Geheimnis-Zeiger der Sendefunktion vorwärts gesetzt.

Der Empfänger nimmt das Paket entgegen, liest zur Interpretation des Headers die Aufbauinformation vom Geheimnis, verarbeitet daraufhin das Paket und gibt den Payload an den Empfangsprozess weiter. Der Geheimnis-Zeiger wird entsprechend vorwärts gesetzt.

Werden weitere Pakete verschickt, läuft der Vorgang für jedes Paket erneut ab. Da die Verbindung permanent besteht, ist kein Verbindungaufbau/-abbau zu berücksichtigen.

3.3 Erweitertes theoretisches Modell

Dieses grundlegende Modell soll nun so erweitert werden, wie es sich in Abbildung 3.4 darstellt. Im Folgenden werden Schritt für Schritt weitere Funktionen zum Modell hinzugefügt, bis das Modell schließlich vollständig ist und auch der Angreiferprozess (der versucht, Storage Channels in den Protokollheadern zu erzeugen) hinzugefügt wird.

3.3.1 Kommunikation mit mehreren Systemen

Im Gegensatz zum bisherigen Modell befinden sich im erweiterten Modell mehr als zwei Kommunikationspartner. Die Verbindungen müssen nach wie vor permanent bestehen und direkt sein. Das bedeutet, dass jedes System mit jedem anderen ohne Zwischenhops verbunden sein muss.

Um die Kommunikation mit mehreren Systemen zu ermöglichen, muss jedes System mit jedem anderen System ein einzelnes Geheimnis teilen und für jede Verbindung einen separaten Geheimnis-Zeiger verwalten. $G(x, y)$ ist dabei das Geheimnis, dass die Systeme x und y teilen. Es gilt entsprechend, dass $G(x, y) = G(y, x)$ ist.

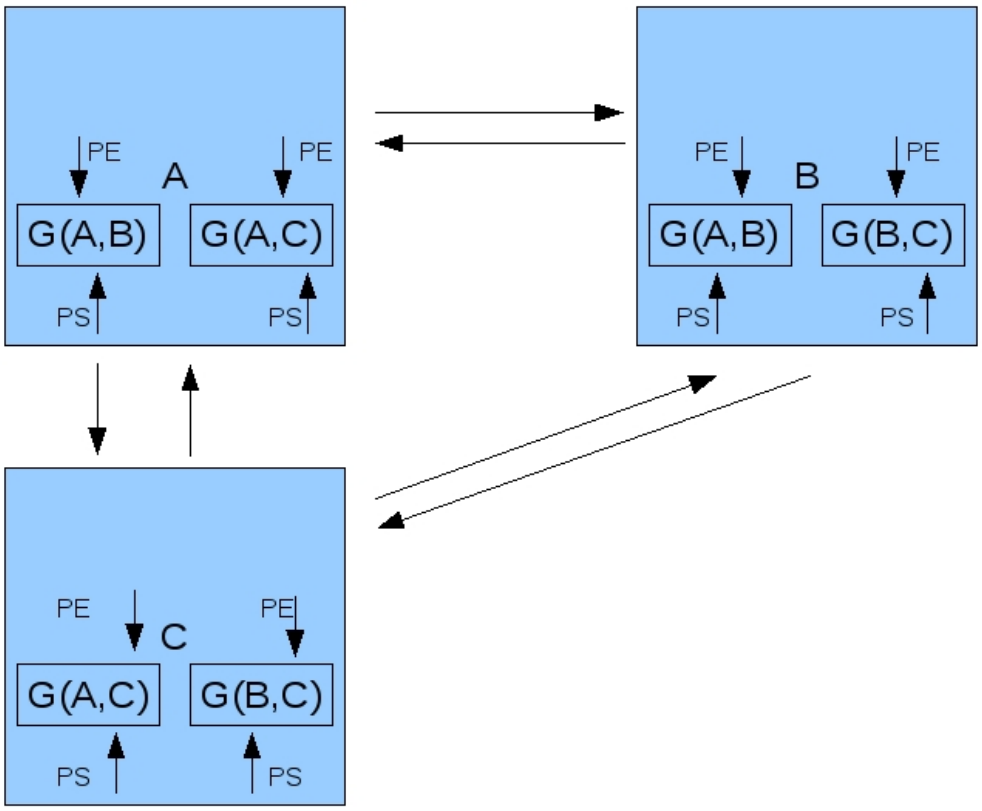

Abbildung 3.4: Erweitertes theoretisches Modell für Header-Strukturveränderungen

3.3.2 Erweiterte Sende-/Empfangsfunktion

Für das Senden zu und das Empfangen von einem System x (außer dem lokalen System) gibt es im erweiterten Modell für jedes System eine separate Funktion. Dies erspart die Adressübergabe an die Sende-/Empfangsfunktion und somit die Möglichkeit, direkten Einfluss auf den Paket-Header (durch Angabe der Ziel-/Quelladresse) zu nehmen. Die Sendefunktionen werden mit FS_x bezeichnet, wobei x der Empfänger ist. Die Empfangsfunktionen werden analog mit FE_x bezeichnet, wobei x hier die Quelle ist.

3.3.3 Funktionsweise der bidirektionalen Kommunikation

Das bisherige Modell kennt nur eine unidirektionale Kommunikation. Im erweiterten Modell ist hingegen eine bidirektionale Kommunikation möglich. Die bidirektionale Kommunikation funktioniert durch die Einführung eines zweiten Zeigers.

- Der erste Zeiger (PS) ist der Sende-Zeiger. Dieser zeigt auf die Geheimnis-Position der für den Header-Aufbau zu verwendenden Stelle (dies ist analog zur Verwendung des Zeigers des bisherigen Sendesystems A).

- Der zweite Zeiger (PE) ist der Empfangs-Zeiger. Dieser zeigt auf die Geheimnis-Position der für den Empfang von Paketen zu verwendenden Stelle (dies ist analog zur Verwendung des Zeigers des bisherigen Empfangssystems B).

Jedes System hat für jedes Geheimnis separate Zeiger PS und PE.

Damit es keine Überschneidungen gibt (also beide Zeiger irgendwann gleiche Bereiche des Geheimnisses abdecken, und dann vorhergesagt werden könnte, wie der nächste Header aussehen wird), müssen für beide Zeiger folgende Regeln gelten:

- Auf dem ersten System wird PS auf eine Position *hinter* der von PE gesetzt. PS wird nach dem Lesen *inkrementiert*, PE wird *dekrementiert*. Somit überschneiden sich beide Zeiger nicht, da das Geheimnis unendlich lang ist.

- Auf dem zweiten System wird PS auf eine Position *vor* der von PE gesetzt. PS wird nach dem Lesen *dekrementiert*, PE wird *inkrementiert*. Somit überschneiden sich auch hier beide Zeiger nicht. Die umgekehrte Bewegungsrichtung der Zeiger muss eingehalten werden, damit vom Sendezeiger auf dem sendenden System die gleichen Werte gelesen werden, wie vom Empfangszeiger auf dem empfangenden System.

Abbildung 3.5 veranschaulicht dieses Detail: PE des einen und PS des jeweils anderen Systems bewegen sich in die gleiche Richtung.

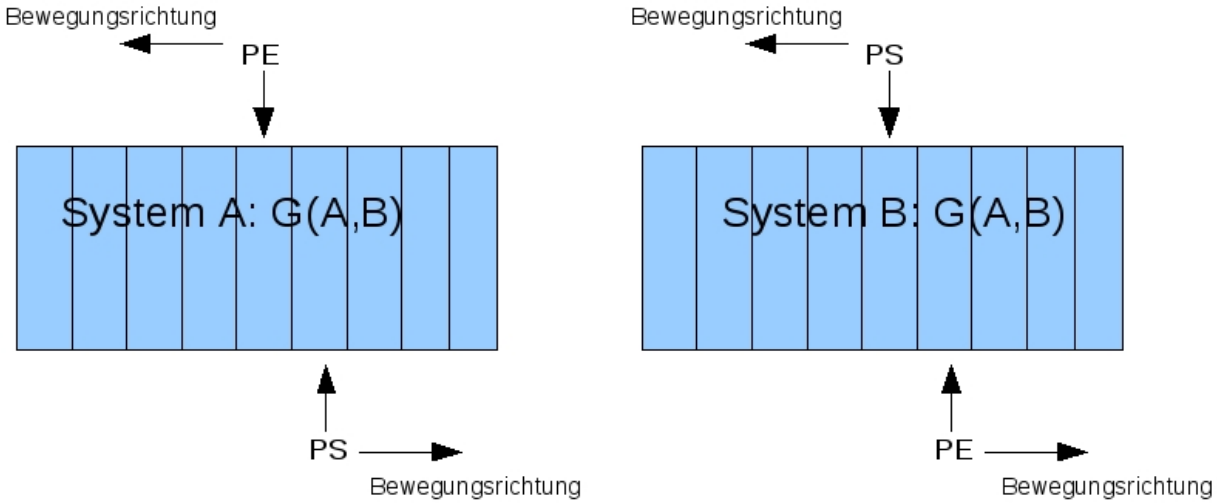

Abbildung 3.5: Zeigerbewegung im erweiterten theoretischen Modell

Überträgt man dieses Verfahren auf die drei Systeme in Abbildung 3.4, so gilt für $G(A, B)$ und $G(A, C)$ auf System A und für $G(B, C)$ auf System B: PS wird inkrementiert und PE wird dekrementiert. Für alle anderen Geheimnisse ($G(A, B)$ auf System B sowie beide Geheimnisse auf System C) gilt die umgekehrte Bewegungsrichtung.

3.3.4 Beispiel: Ablauf einer simplen Kommunikation

System A soll nun beispielhaft eine Nachricht an System B schicken. System B soll diese Nachricht anschließend an System C weiterleiten.

Senden der Nachricht von A nach B: Der Sendeprozess auf System A übergibt den Payload an die Sendefunktion FS_B. System A liest entsprechend die Aufbauinformation von $G(A, B)$, auf die der zugehörige Sendezeiger PS zeigt, baut damit den Header, generiert das Gesamtpaket und verschickt es an B. System A inkrementiert zum Schluss den Sendezeiger PS.

Empfang der Nachricht auf System B: Der Empfangsprozess ruft die Empfangsfunktion FE_A auf, um das Paket von System A entgegenzunehmen. Zur Interpretation des Headers wird vom Geheimnis $G(A, B)$ an der Stelle, auf die der zugehörige Empfangszeiger PE zeigt, die Interpretationsinformation gelesen. PE wird inkrementiert und der Payload wird an den Empfangsprozess zurückgegeben.

Um diese Nachricht an C zu schicken, ist folgender Ablauf notwendig:

Senden der Nachricht von B nach C: Der Sendeprozess auf System B übergibt den Payload an die Sendefunktion FS_C. System B liest entsprechend die Aufbauinformation von $G(B, C)$, auf die der zugehörige Sendezeiger PS zeigt, baut damit den Header, generiert das Paket und verschickt es an C. System B inkrementiert zum Schluss noch den Sendezeiger PS.

Empfang der Nachricht auf System C: Der Empfangsprozess ruft die Empfangsfunktion FE_B auf, um das Paket von System B entgegenzunehmen. Zur Interpretation des Headers wird vom Geheimnis $G(B, C)$ an der Stelle, auf die der zugehörige Empfangszeiger PE zeigt, die Interpretationsinformation gelesen. PE wird inkrementiert und der Payload wird an den Empfangsprozess zurückgegeben.

3.3.5 Zusammenfassung des erweiterten Modells

Das erweiterte theoretische Modell der Header-Strukturveränderungen unterstützt die Kommunikation beliebig vieler direkt miteinander verbundener Systeme. Dazu muss jedes System x mit jedem anderen System y ein Geheimnis $G(x, y)$ austauschen. Jedes System verfügt über eine separate Sende- und Empfangsfunktion für jedes andere System. Zur Einführung bidirektionaler Kommunikation wird nun zwischen Empfangs- und Sendezeigern für jedes Geheimnis unterschieden. Der Sendezeiger eines sendenden Systems wird in die gleiche Richtung bewegt, wie der Empfangszeiger des empfangenen Systems. Gleiches gilt für die jeweiligen Empfangszeiger.

Da alle Systeme dieses Modells direkt miteinander verbunden sind, ist keine Routingfunktionalität notwendig. Explizites Weiterleiten (der Sender gibt dabei nicht das tat-

sächliche Ziel im Protokollheader an, sondern nur das des nächsten Nachbarn und kann auch nicht über die Weiterleitung bestimmen, so wie in Abschnitt 3.3.4 beschrieben) von Paketen ist hingegen möglich.

3.4 Einführung eines Angreifers

Um das theoretische Modell der Header-Strukturveränderungen zu vervollständigen, fehlt nun noch eine Angreifer-Instanz, die die Aufgabe hat, einen Storage Channel innerhalb von Protokollheadern aufzubauen.

Dazu werde ich zwischen zwei Angreifer-Instanzen unterscheiden: Einem *internen*, also systemlokalen, Angreiferprozess und einem *externen*, autonomen Angreifersystem, damit beide Angriffspunkte des Modells untersucht werden können.

3.4.1 Interner Angreiferprozess

Abbildung 3.6 zeigt einen internen Angreiferprozess (*Mallory*) auf System A. Das Ziel von Mallory besteht darin, ein Paket an ein anderes System zu senden, und dabei manipulierte Daten im Paketheader unterzubringen.

Mallory kann zu diesem Zweck entweder die Sendefunktion SF_x benutzen, um ein Paket an System x zu schicken, oder das Paket manuell zusammenbauen und senden.

Im ersten Fall erlaubt die Sendefunktion nur die Übergabe von Payload, ein direkter Zugriff auf den Header und damit die Errichtung eines Storage Channels im Header, ist also ausgeschlossen. Damit ist die Funktion der Header-Strukturveränderung erfüllt.

Im zweiten Fall, also beim manuellen Aussenden der Pakete, fehlt Mallory der Zugriff auf das Geheimnis, der nur der Sende- und Empfangsfunktion vorbehalten ist. Mallory kann also den Paketaufbau nur erraten. Der Rateaufwand ist bei n Header-Bestandteilen allerdings nur $n!$ (Vgl. Tabelle 3.1), was bei wenigen Bestandteilen keinen großen Aufwand ausmacht. Mallory wird also entsprechend zur Verbindung gehörige Pakete versenden können. Über direkten Zugriff auf den Empfang der Pakete beim Empfänger (also Umgehung der Empfangsfunktion) kann Mallory die geschleusten Paketdaten lesen.

Außerdem würde jedes Paket mit korrekt erratenem Aufbau die Verbindung zwischen zwei Kommunikationspartnern desynchronisieren. Es würde also ein Denial of Service-Angriff stattfinden.

Aus diesem Grund muss dem Modell die Regel hinzugefügt werden: Nur Sende- und Empfangsfunktionen dürfen Zugriff auf das Übertragungsmedium haben. Damit ist das Problem der Injizierung von Mallorys Paketen und das Problem der Desynchronisierung lokal beseitigt.

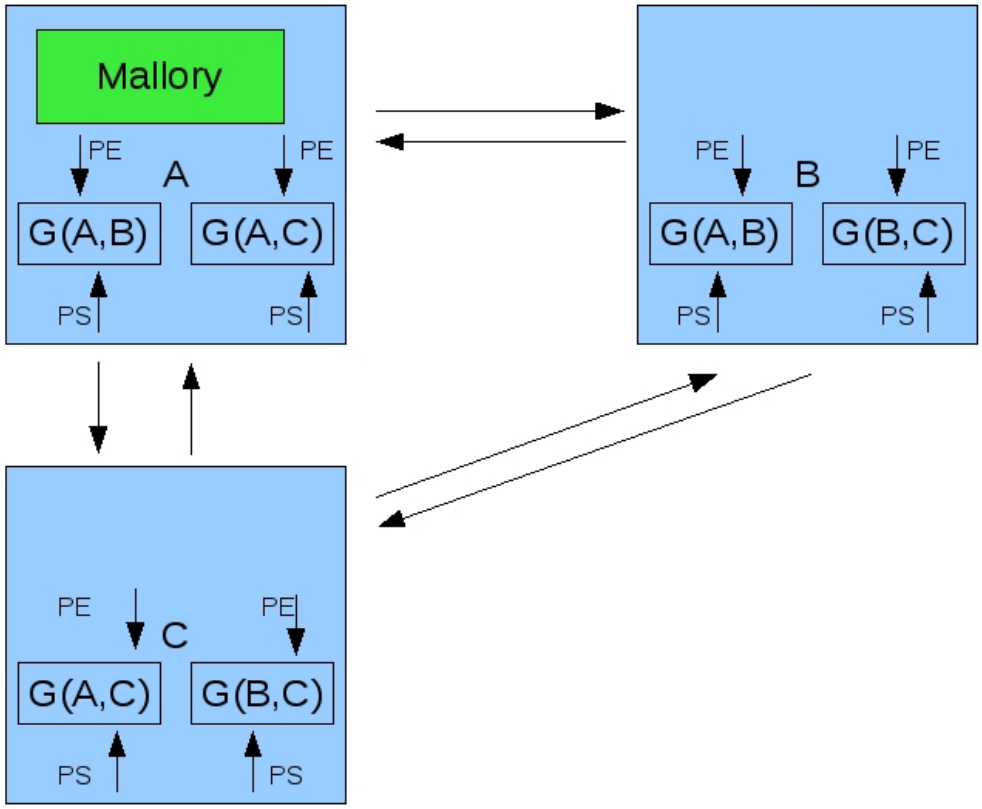

Abbildung 3.6: Interner Angreiferprozess bei Header-Strukturveränderung

3.4.2 Externes Angriffssystem

Ein externer Angreifer in Form eines autonomen Angreifersystems, dass direkte Verbindungen zu anderen Systemen aufbaut, hat keinen Zugriff auf die Geheimnisse zwischen einzelnen Systemen und auch kein eigenes Geheimnis mit anderen Systemen ausgetauscht. Zudem existieren auf den legitimen Systemen wegen des fehlenden gemeinsamen Geheimnisses keine Empfangsfunktionen für Pakete des externen Angreifersystems. Abbildung 3.7 veranschaulicht dieses Szenario. Dabei sei es dem Angreifer gelungen, eine direkte Verbindung zu den legitimen Systemen aufzubauen.

Das externe Angreifersystem muss entsprechend ohne eine Sende-/Empfangsfunktion

Bereiche	**Kombinationen**	**Bereiche**	**Kombinationen**
1	1	6	720
2	2	7	5040
3	6	8	40320
4	24	9	362880
5	120	10	3628800

Tabelle 3.1: Anzahl der Kombinationen pro Anzahl von Header-Bereichen

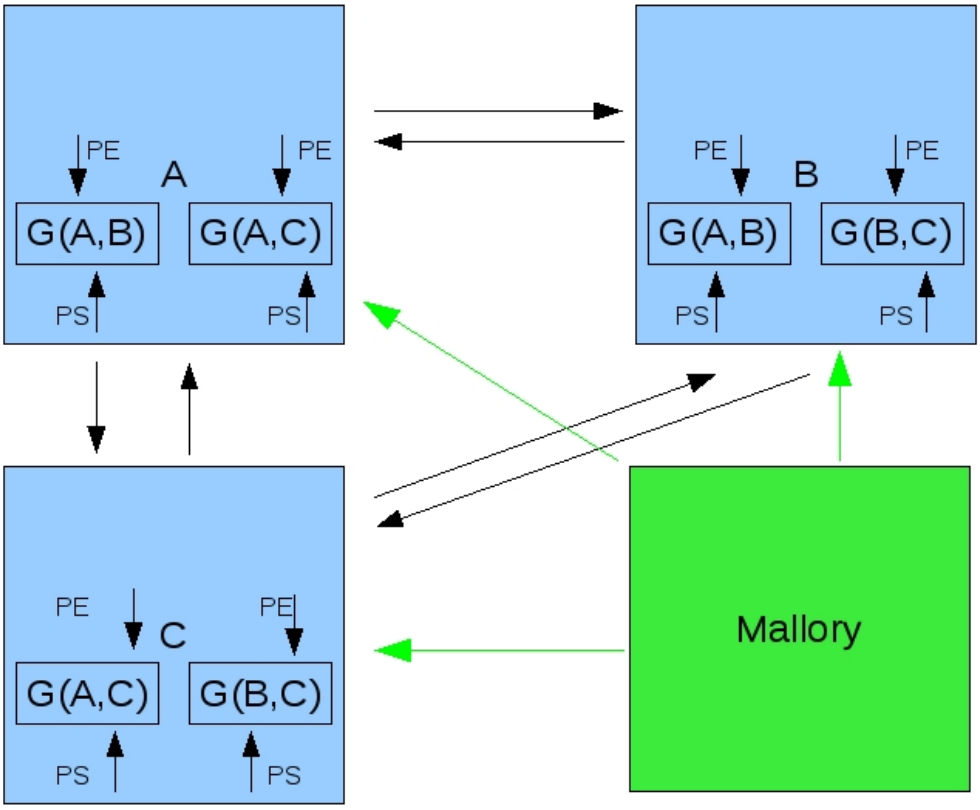

Abbildung 3.7: Externes Angreifersystem bei Header-Strukturveränderung

mit Zugriff auf ein anerkanntes Geheimnis auskommen, wie sie auf den legitimen Systemen vorhanden sind, kann aber sehr wohl Pakete generieren und empfangen. Die von ihm generierten Pakete werden jedoch von legitimen Empfängern – auf Grund fehlender Empfangsfunktionen – verworfen. Obwohl Mallory Pakete empfangen kann, werden keine Pakete von einem legitimen System an ihn verschickt, da dort keine Sendefunktion vorhanden ist, die Daten an Mallory zustellt.

Da im erweiterten theoretischen Modell keine Adressübergabe an Sendefunktionen möglich ist und auch keine Adressinformation im Header von Paketen vorhanden ist (Vgl. Abschnitt 3.3.2), ist ein Senden von Paketen mit gefälschter Adresse nicht möglich. Aus diesem Grund kann ein externes Angreifersystem auch keine Verbindung zwischen zwei legitimen Systemen desynchronisieren.

Die einzige Möglichkeit zur Kommunikation mit Storage Channeln bestünde zwischen zwei externen Angreifersystemen. Diese könnten unabhängig von jeglicher Headerstruktuveränderung kommunizieren und entsprechend Storage Channels aufbauen. Die Sinnhaftigkeit dieser Möglichkeit steht allerdings in Frage, da legitime Systeme aus den oben genannten Gründen nicht von einer solchen Kommunikation betroffen sein können.

3.5 Praktische Umsetzung

Nach der Vorstellung des theoretischen Modells der Header-Strukturveränderungen stellt sich die Frage nach ihrer praktischen Umsetzbarkeit. Das Modell praktisch komplett umzusetzen, würde bedeuten, dass die Netzwerk-Stacks in Betriebssystemkernen verändert werden müssen, damit die Header-Strukturveränderung angewendet werden kann.

Solch eine aufwändige Lösung wird jedoch bestenfalls in Spezialfällen (etwa der Entwicklung militärischer Kommunikation, die zugleich unabhängig von anderen Netzwerksystemen sein soll) lohnenswert sein. Zudem ist es nur begrenzt sinnvoll, dem theoretischen Modell soweit zu folgen, dass tatsächlich gar kein direkter Einfluss auf den Inhalt eines Headers mehr genommen werden kann. Der Grund dafür liegt darin, dass sich die Funktion vieler Header-Komponenten, auf die kein direkter Einfluss genommen werden kann, erübrigt.[2]

Statt einer kompletten Implementierung des Modells ist daher eine Hybridlösung sinnvoller. Im Rahmen dieser Diplomarbeit wurde eine solche Hybridlösung in Form eines Proof-of-Concept-Codes implementiert.

3.5.1 Grundlegendes zur praktischen Umsetzung

Da das Modell der Header-Strukturveränderung nicht ohne Weiteres direkt in die Praxis umgesetzt werden kann, muss ein Kompromiss zwischen theoretischem Modell und der realen Welt gefunden werden. Diesen Kompromiss werde ich im Folgenden in einem für die Praxis tauglichen Modell beschreiben.

Im praktischen Modell soll ein expliziter Dienst der Anwendungsschicht mit Hilfe der Header-Strukturveränderung vor Storage Channels geschützt werden. Dabei kann es sich sowohl um klassische Client-Server-Kommunikation als auch um Kommunikation zwischen Peers handeln. Außerdem lässt sich das Modell auf alle von einem System angebotenen bzw. verwendeten Dienste anwenden, sodass letztlich alle Dienste parallel geschützt werden können. Da die Header-Strukturveränderung jedoch protokollabhängig ist, muss für jedes Protokoll (und damit für alle Dienste mit unterschiedlichem Protokoll) eine eigene Software(-Komponente) entwickelt werden.

Dabei wird entweder nur eines von jeweils zwei miteinander kommunizierenden Netzwerksystemen die Header-Strukturveränderung anwenden, um Pakete zu senden (dies entspräche einer unidirektional abgesicherten Verbindung), oder es wird beidseitig die Header-Strukturveränderung auf zu sendende Pakete angewendet (damit würde es eine in

[2] Beispielsweise macht es keinen Sinn, die gewünschte HTTP-URL nicht bei einem HTTP-Request vom Benutzer wählen zu lassen – in diesem Fall bräuchte es keinen Platz für die URL im Header zu geben.

beide Richtungen abgesicherte Verbindung geben).

3.5.2 Schutz der Verbindung vor Storage Channels

Nach wie vor steht aber noch die Frage offen, wie die Header-Strukturveränderung im praktischen Modell angewendet werden kann. Ich sehe dabei zwei Anwendungsfelder, die, wenn sie kombiniert werden, einen guten Schutz vor Storage Channels bieten.

Die Header-Strukturveränderung wird in diesem Kompromiss zwischen Theorie und Praxis in beiden Anwendungsfeldern nicht zur direkten Vermeidung von Storage Channels, sondern zur Detektion nicht erlaubter Kommunikation, und damit automatisch auch der Detektion von Storage Channels, eingesetzt. Nach einer Detektion kann allerdings eine automatisierte Aktion (etwa das Beenden der Verbindung und/oder das Protokollieren des Angriffs) erfolgen.

Erste Funktion: Markierung von legitimen Traffic

Sendet ein System ein Paket an einen mit der Header-Strukturveränderung gesicherten Dienst, kann der Empfänger prüfen, ob das Paket dem Aufbau der aktuellen Geheimnisinformation entspricht.

Entspricht ein auf dem Empfänger-System eintreffendes Paket vom Sender dem jeweiligen geheimen Aufbau nicht, wird es verworfen und der verdeckte Kommunikationskanal ist detektiert worden. Daraufhin kann der Kanal unterbunden werden. Entspricht ein Paket hingegen dem Aufbau, wird es vom Empfangsdienst verarbeitet. Dadurch, dass nur Sender und Empfänger über den geheimen Aufbau Bescheid wissen und es relativ viele Zusammensetzungsmöglichkeiten gibt, ist es für einen Angreifer schwierig, selbst Pakete zu erzeugen, die dem aktuellen Geheimnis entsprechen. Der legitime Traffic wird durch die Header-Strukturveränderung also *markiert*. Im Gegensatz zu einer einfachen Markierung (etwa dem Setzen eines bestimmten Bits) ist die Header-Strukturveränderung daher schwieriger zu fälschen. Würde ein Angreifer mittels Brute-Force versuchen, den korrekten Paketaufbau zu erraten, würde bereits das erste unpassende Paket detektiert werden.

Die erste Funktion zur Vermeidung von Storage Channels, die über einen vermeidlich legitimen, weil üblichen Kanal senden, filtert also Pakete heraus, die nicht explizit die korrekte Header-Strukturveränderung anwenden. Da für die Header-Strukturveränderung kein Zusatzheader (etwa in Form eines einkapselnden Protokolls) erforderlich ist, werden zu diesem Zweck keine zusätzlichen Headerdaten übertragen und die Auslastung des Netzwerks bleibt normal. Zur Generierung und Interpretation der entsprechenden Header mittels Geheimnis steigt allerdings die Auslastung beim Sender- und beim Empfängersystem: Senderseitig ist dies die Headergenerierung und empfangsseitig die Headerinterpretation.

Zweite Funktion: Header-Normalisierung

Nun wäre es aber denkbar, dass ein Angreifer Netzwerkpakete über die Sendefunktion eines Systems verschickt, die Zugriff auf das Geheimnis hat. Und da – wie bereits auf Seite 44 erwähnt – einige Header-Bestandteile direkt gesetzt werden können müssen, um ihren Zweck zu erfüllen, kann ein Angreifer zumindest diese manipulieren.

Die zweite (aber indirekte) Funktion der Header-Strukturveränderung besteht daher darin, dass der Sender, der das Geheimnis zum Headeraufbau kennt, die Pakete vor dem Senden einer Normalisierung unterzieht um bereits darin enthaltene Storage Channels zu unterbinden. Indirekt ist diese Funktion der Header-Strukturveränderung, weil sie auch hier ausschließlich zur Markierung des normalisierten Traffics verwendet wird.

Die Normalisierung funktioniert so, dass etwa bei einer POP3-Verbindung die Möglichkeit zur Übertragung einzelner Bits durch Interpretation der Groß- bzw. Kleinschreibung einiger Buchstaben unmöglich gemacht wird.[3]

Solche Storage Channels können verhindert werden, indem ein POP3-Server bzw. -Client Befehle wie “RETR” immer in Großbuchstaben abändert. Somit kann keine Übertragung einzelner Bits durch Befehle wie “Retr”, “retr” oder “ReTr” erfolgen. Der Empfänger muss Pakete, auf die die korrekte Header-Strukturveränderung angewandt wurde, nicht mehr auf solche Storage Channels überprüfen, da außer ihm nur der autorisierte Sendeprozess Zugriff auf die geheimen Aufbauinformationen hat. Auch hier kommt der Sicherungsfunktion zu Gute, dass es schwer ist, den Traffic korrekt zu fälschen und dass durch die Tatsache, dass keine zusätzlichen Header-Bestandteile übertragen werden, die Netzwerklast nicht erhöht wird.

3.5.3 Proof-of-Concept-Implementierung

In der Proof-of-Concent-Implementierung, die ich ihm Rahmen dieser Diplomarbeit erstellt habe, wird die Header-Strukturveränderung für eine unidirektionale Client-Server-Umgebung über das POP3-Protokoll realisiert. Abbildung 3.8 veranschaulicht diese Kommunikation. Der POP3-Client sendet dabei Daten an einen lokalen Dienst, den Modifier für die Header-Strukturveränderung (im Folgenden HSV-Modifier). Dieser normalisiert die Daten und wendet die Header-Strukturveränderung auf den Paket-Header an. Anschließend werden die Daten an den POP3-Server weitergeleitet, der die Header-Strukturveränderung rückgängig macht und eine Antwort zurückschickt. Der HSV-Modifier leitet die Antwort des Servers wiederum an den Client weiter. Die Verbindung wird dabei unidirektional in Richtung der vom Client an den Server gehenden Pakete abgesichert.

[3] Dabei würde etwa ein großes “A” als 1er Bit-und ein kleines “a” als 0er-Bit interpretiert werden und andersherum.

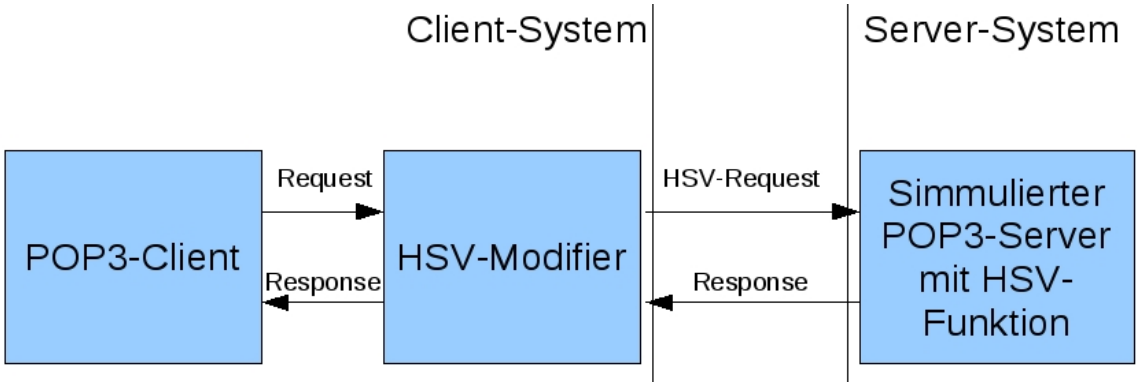

Abbildung 3.8: Proof-of-Concept-Implementierung der Header-Strukturveränderung

Das POP3-Protokoll bietet eine sehr anschauliche Möglichkeit, die Header-Strukturveränderung in dieser Arbeit an den ASCII-Daten zu zeigen. Des Weiteren ist das Protokoll TCP-basiert, was den Vorteil mit sich bringt, dass TCP sich um die Synchronisation der Verbindung kümmert und somit der Code, der sich um die Header-Strukturveränderung und die Verwaltung des gemeinsamen Geheimnisses kümmert, nicht mit einer Desynchronisierung umgehen muss. Bei einer Desynchronisierung würden (wegen der nicht mehr zusammenpassenden Zeiger auf die zu verwendenden Geheimnispositionen) die verschickten Pakete mit einer anderen Information aufgebaut werden als mit der sie interpretiert werden würden. Mehr dazu in Abschnitt 3.7.5.

Der POP3-Server wird dabei nur simuliert, um zu zeigen, wie Befehle vom Client über den HSV-Modifier erhalten und beantwortet werden können. Da der Client beim POP3-Protokoll einen blanken Header ohne Payload sendet, muss der gesamte Traffic an den Server einer Header-Strukturveränderung unterzogen werden. Würde man auch den Traffic vom Server an den Client mit einer Header-Strukturveränderung versehen wollen, so dürften nur die Return-Codes (etwa "+OK mailbox has 1 message (721 octets)\r\n") verändert werden, da etwa zurückgegebene Mail-Header und -Texte nicht zum POP3-Header gehören. Dies bringt Probleme mit sich, da der Header-Text eine variable Länge nutzt, doch dazu mehr in Abschnitt 3.7.1.

Der HSV-Modifier stellt, abgebildet auf das theoretische Modell der Header-Strukturveränderung, somit die Sende- und Empfangsfunktion bereit. Gleiches gilt für die für die Headerstruktuveränderung zuständige Serverkomponente.

Permutation der Header-Bestandteile

Im theoretischen Modell der Header-Strukturveränderung wurde nicht explizit festgelegt, welche "Bestandteile" eines Headers einer Permutation unterzogen werden sollen. In der Praxis ist diese Überlegung aber entscheidend für die Implementierung. Es bestehen fol-

gende Möglichkeiten, die Header-Strukturveränderung anzuwenden:

- Würfelung einzelner Bereiche des Headers (etwa für IPv4 die Felder der Checksum, der Protokoll-Version, der Header-Länge usw.). Diese Aufteilung ist (je nach Protokoll-Header) die mit der geringsten Granularität.[4]
- Auf einzelne Bytes, um somit etwa Strings zu verwürfeln (bei POP3 wäre dies die Permutation der zu übertragenden Zeichen der Befehle und Return-Codes).
- Ganz feingranular auf einzelne Bits. Die Anzahl der möglichen Kombinationen ist hierbei am größten und damit der Rateaufwand für Angreifer am höchsten.
- Auf Gruppen von Bytes oder Bits.

In der Proof-of-Concept-Implementierung werden 4er-Gruppen von Bytes einer Permutation unterzogen. Dabei gibt es pro 4-Byte-Gruppe 4! (also 24) verschiedene Möglichkeiten, diese Byte-Gruppe zu übertragen. Ein üblicher POP3-Befehl besteht aus mindestens zwei vollen 4-Byte-Gruppen und ist in der Regel maximal vier Byte-Gruppen lang. Unvollständige Bytegruppen werden dabei mit speziellen Padding-Bytes aufgefüllt, sodass schließlich immer volle 4-Byte-Gruppen übertragen werden.

Beispiel: Der Client sendet den Befehl "RETR 1\r\n" an den Server. Dies sind acht Bytes, also zwei Gruppen mit je vier Byte. Der String wird entsprechend zweigeteilt. Für jede Hälfte wird nun ein Wert vom Geheimnis gelesen, der die zu verwendende Permutation – und damit, welches Zeichen an welche Stelle verschoben wird, festlegt. Der String wird sich anschließend beispielsweise aus den zwei Teilen "TERR" und "1\r \n" zusammensetzen. Dieser String wird daraufhin zu "TERR1\r \n" zusammengesetzt und an den Server übertragen, der die Daten mit Hilfe des eigenen Geheimnisses wieder als 4er-Gruppen interpretiert um am Ende den Originalstring zu erstellen. Dabei rechnet er über die Geheimnisinformation, die beinhaltet, welches Byte an welche Stelle verschoben wurde, die Verschiebung rückwärts. ■

Bei durchschnittlich zwei Byte-Gruppen mit je 24 Möglichkeiten gibt es pro typischem POP3-Befehl also 2 * 24 = 48 Möglichkeiten zum Aufbau eines Headers. Dies genügt, da beim ersten unpassend geratenen Befehl eine Detektion erfolgen kann.

Bei der Übertragung von Nachrichten mit nicht enorm kurzer Länge über einen POP3-basierten Storage Channel werden zudem mehrere Nachrichten – und bei diesem Szenario damit mehrere Befehle vom Client an den Server – notwendig. Entsprechend muss für jeden

[4] Während sich Header, die vor allen Dingen aus größeren Bereichen bestehen, nur grob aufteilen lassen, wäre bei Headern, die etwa aus lauter 4-Bit-Bestandteilen bestehen, bereits eine feinere Aufteilung als bei der einzelner Bytes gegeben.

zu übertragenen Storage Channel-Befehl erneut der korrekte Aufbau geraten werden. Die Detektion eines Storage Channels steigt somit mit der Nachrichtenlänge an.

Geheimnisaustausch

Der Proof-of-Concept-Code verfügt zur Demonstration über ein einziges statisches Geheimnis. Auf Grund der Tatsache, dass der Austausch von Geheimnissen in der Vergangenheit bereits in vielen Veröffentlichungen diskutiert wurde, und sich diese Diplomarbeit vor allen Dingen auf die eigentliche Technik der Header-Strukturveränderung konzentriert, soll dies zu Demonstrationszwecken genügen. Das Geheimnis wird direkt im Quellcode definiert und muss daher nicht ausgetauscht werden. Für jede neue Verbindung wird das Geheimnis erneut verwendet und ein intelligenter Angreifer kann durch Mitlesen der übertragenen Daten auf das gemeinsame Geheimnis schließen.

Die Mittel der modernen Kryptographie bieten mit Public-Key basierten Verfahren die Möglichkeit, einen gemeinsamen Schlüssel (ein gemeinsames Geheimnis) auszutauschen. Typische Verfahren sind hierbei der Diffie-Hellmann-Algorithmus und seine Verbesserungen (etwa das Station-to-Station-Protokoll).[5] Solche Verfahren könnten auch zum Austausch dynamisch generierter Geheimnisse für Header-Strukturveränderungen einer Netzwerkverbindung verwendet werden. Dies hätte allerdings die Folge, dass die Implementierung der Header-Strukturveränderung deutlich umfangreicher und komplizierter werden würde; zudem würde durch das Protokoll für den Geheimnisaustausch ein Overhead erzeugt werden.

Anwendung der Proof-of-Concept-Implementierung

Der Proof-of-Concept-Code besteht aus zwei Komponenten: “hsv_send” und “hsv_recv”. “hsv_send” ist der HSV-Modifier. Er nimmt lokal die Pakete eines POP3-Clients entgegen und leitet sie, abgesichert durch die Header-Strukturveränderung, an den POP3-Server weiter. Der POP3-Server wird von dem Programm “hsv_recv” simuliert. Er dekodiert die Pakete und interpretiert ihren eigentlichen Inhalt.

Gestartet werden beide Programme zunächst unabhängig voneinander. Dabei muss dem HSV-Modifier die IP des POP3-Servers und dessen Port übergeben werden.

Listing 3.1: Aufruf von hsc_send

```
1 $ ./hsc_send -s 127.0.0.1 -p 110
2 Waiting for local connection on Port 10001...
```

Der Server wird ohne weitere Parameter gestartet.

[5] Vgl. [SCHNEI06], S. 587ff.

Listing 3.2: Aufruf von hsc_recv

```
1 $ sudo ./hsc_recv
2 Waiting for a connection on Port 110...
```

Die Verbindung zwischen beiden Diensten besteht zu diesem Zeitpunkt noch nicht. Erst dann, wenn ein POP3-Client sich mit dem HSV-Modifier verbindet, stellt der HSV-Modifier auch die Verbindung zum Server her. Am einfachsten lässt sich ein Test mit einem Telnet-Client durchführen, letztlich kann aber jedes Mailprogramm hierfür verwendet werden. Zu beachten ist dabei, dass sich das Mailprogramm mit dem lokalen Port 10001, auf dem der HSV-Modifier Pakete entgegennimmt, verbindet.[6]

Listing 3.3: Telnet als POP3-Client für den HSV-Modifier "pct_send"

```
1 $ telnet localhost 10001
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^]'.
5 +OK Header-Structure-Changing POP3-Server-Simmulator
6 stat
7 +OK 0 0
8 retr 99
9 -ERR Invalid message number.
10 quit
11 +OK closing.
12 Connection closed by foreign host.
```

Angriffs-Simmulation

Würde ein Client eine direkte Verbindung zum Server aufbauen, so würde der Server (nach der Detektion der unüblichen Daten) auf einen Storage Channel schließen und die Verbindung beenden.

Listing 3.4: Versuch einen Storage Channel zu erzeugen (Client-seitige Ausgabe)

```
1 $ telnet localhost 110
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^]'.
5 +OK Header-Structure-Changing POP3-Server-Simmulator
6 ReTr 1
```

[6] Tests wurden mit dem Mailprogramm "Sylpheed" durchgeführt.

```
7 -ERR Unknown command.
8 Connection closed by foreign host.
```

Listing 3.5: Versuch einen Storage Channel zu erzeugen (Server-seitige Ausgabe)

```
1 $ sudo ./hsc_recv
2 Waiting for a connection on Port 110...
3 Connection established.
4 Received command is unknown. This could be an attack.
```

3.6 Reine Traffic-Normalisierung als Alternative

Als Alternative zur Header-Strukturveränderung, die auf Client und Server durchgeführt wird, besteht die Möglichkeit, eine reine Traffic-Normalisierung durchzuführen. Im Gegensatz zur Header-Strukturveränderung wird die Traffic-Normalisierung entweder beim Empfänger oder auf einem Router durchgeführt. Hierbei wird die Umgestaltung des Traffic-Headers in allen Fällen gespart, bei denen keine verdächtigen Bits gefunden werden.

Implementiert ist Traffic-Normalisierung beispielsweise im Intrusion Detection System "Snort". Dort wird die Normalisierung eingesetzt um Netzwerk-Pakete in ein einheitliches Format zu bringen. Durch dieses einheitlichere Format ist es einfacher, die Scanregeln des Intrusion Detection Systems anzuwenden.[7] Die Normalisierung wird bei Snort mit so genannten "Präprozessoren" (wie etwa "HTTP Inspect") durchgeführt.[8]

Auch einige Firewalls verfügen über eine Funktionen zur Traffic-Normalisierung. Die pf-Firewall des OpenBSD-Betriebssystems unterstützt beispielsweise das "Scrubbing" von Netzwerkpaketen. Eintreffende Netzwerkpakete können dabei auf unterschiedlichste Weisen verändert werden, die sich vor allen Dingen auf IP und TCP beziehen. Darunter findet sich etwa die Ersetzung der Identifikationsnummer von IP-Paketen, womit sich dann auch Storage Channels innerhalb der Identifikationsnummer löschen lassen. Neben dieser Funktion gibt es derzeit noch sieben weitere, die aber nur bedingt zur Vermeidung von Storage Channels eingesetzt werden können.[9]

[7] Vgl. [NORTHC07], S. 293.
[8] Vgl. [COXGERG04], S. 84. sowie [SPENN05], S. 256.
[9] Vgl. [OBSDPF08].

3.7 Probleme der Header-Strukturveränderung

Es ergeben sich in Verbindung mit der Headerstruktuveränderung allerdings auch Probleme, die ich im Folgenden beschreiben werde.

3.7.1 Variable Headerlänge

Werden gleichzeitig Header und Payload in einem Datenpaket übertragen, muss der Empfänger wissen, wieviele Bytes des Datenpaketes als Header interpretiert werden müssen. Dies ist notwendig um den originalen Header wiederherzustellen.

Ist, wie etwa beim IPv4-Protokoll[10], die Headerlänge variabel, kann der Empfänger nicht sicher wissen, wieviele Bytes zu interpretieren sind, da sich die Angabe der Headerlänge im Header befindet, dessen Position in den Daten jedoch noch nicht bekannt ist.[11] In anderen Fällen gibt es nicht einmal eine Angabe der Headerlänge (etwa bei textbasierten Protokollen, die einen Parser benötigen, so wie HTTP).

Als Lösung des Problems für beide Problemvarianten bietet sich die Möglichkeit an, vor dem Headeranfang die Länge des Headers zu übertragen, damit der Empfänger weiß, wieviele Bytes zu interpretieren sind. Diese Vorgehensweise macht die Header-Strukturveränderung allerdings etwas unattraktiver, da die Datenpakete wachsen und aus diesem Grund die Eigenschaft verloren geht, keinen zusätzlichen Traffic zu erzeugen.

3.7.2 False Positives

Durch die Header-Strukturveränderung wird beim Empfänger eine Plausibilitätsprüfung notwendig. Diese prüft, ob die empfangenen und anschließend interpretierten Daten sinnvoll sind (also etwa einem bekannten Befehl entsprechen). Ist dies nicht der Fall wird von einem Storage Channel ausgegangen. Kommt es nun zu einem Übertragungs- oder Tippfehler (etwa bei einer Telnet-Verbindung), schlägt die Plausibilitätsprüfung fehl und ein Storage Channel wird vermutet, obwohl es sich nicht um einen Storage Channel handelt. Man spricht in diesem Zusammenhang in der Angriffserkennung von "False Positives", also fälschlicher Weise als "Angriffsdaten" erkanntem Input.[12] Bei Protokollen mit Prüfsummen und gleichzeitiger Datenübertagung durch Clients (die keine Tippfehler produzieren) fällt die Wahrscheinlichkeit einer derartigen False Positive-Detektion jedoch vernachlässigbar gering aus.

[10] Die Länge des Headers wird bei IPv4 über das Feld "IP Header Length" angegeben [POSTEL81B].

[11] Schließlich ist die Permutation abhängig von der Länge des Headers und der Empfänger muss wissen, wieviele Daten permutiert wurden und wieviele Informationen entsprechend vom Geheimnis gelesen werden müssen.

[12] Vgl. [ZWINKY00], S. 749.

3.7.3 Zusätzliche Rechenlast

Bei der Generierung von Headern muss Rechenzeit investiert werden, um die Header-Strukturveränderung anzuwenden. Genauso wird bei der Interpretation der Daten empfangsseitig Rechenzeit benötigt. Dadurch sinkt die Anzahl der Pakete die von Netzwerksystemen pro Zeiteinheit verarbeitet werden können. Aus diesem Grund ist abzuwägen, ob der Schutz vor Storage Channels im Einzelfall wichtiger ist als die Performance.

3.7.4 Keine vollständige Vermeidung von Covert Channels

Die Header-Strukturveränderung kann nicht zur vollständigen Vermeidung von Storage Channels beitragen, da nicht alle Bereiche eines Headers normalisiert werden können (sonst würde ihre Funktion verloren gehen, etwa bei einem IPv4-TTL-Feld). Zudem trägt sie nichts zur Vermeidung von Timing Channels bei. Dennoch verringert sie die Möglichkeiten, mit denen Storage Channels erstellt werden können gezielt und kann damit ähnliches bieten, wie es bei den alternativen Techniken zur Normalisierung der Fall ist.

3.7.5 Möglichkeit der Desynchronisation

Wird eine Verbindung mit Header-Strukturveränderung durch einen aktiven Angreifer mit injizierten Netzwerkpaketen oder durch einen Störfall mit Paketverlust desynchronisiert, kann diese nicht wieder re-synchronisiert werden. Das liegt daran, dass ein System nicht die Position des Geheimniszeigers beim Kommunikationspartner kennt.

Aus diesem Grund ist es empfehlenswert, die Header-Strukturveränderung vor allen Dingen für Dienste TCP-basierter Protokolle der Anwendungsschicht zu implementieren. Die Gründe hierfür sind, dass TCP das Spoofing von Netzwerkpaketen (dies gilt besonders für das Blind-TCP-Spoofing) erschwert[13], über eine Fehlererkennung/-korrektur verfügt und zudem doppelt empfangene Segmente aussortiert.[14]

3.7.6 Eingeschränktes Routing

Wird auf Ebene des IP-Protokolls eine Header-Strukturveränderung durchgeführt, so kann kein Routing mehr stattfinden, weil kein Router eine Unterstützung für diese Technik bereitstellt. Es müsste eine Hop-by-Hop-Übertragung durchgeführt werden, bei der jeder Router ein Paket empfängt, die Header-Strukturveränderung rückgängig macht, das anschließend normale Paket erneut einer Header-Strukturveränderung unterzieht (eine, die

[13] Angriffe auf TCP-Verbindung (TCP Session-Hijacking und Desynchronisierung) können trotzdem erfolgreich sein. Vgl. [SPENN05], S. 762 sowie [CHESW04], S. 89.
[14] Vgl. [WASH98], S. 279f.

der nächste Empfänger dekodieren kann) und es dann weiterleitet (Vgl. Abschnitt 3.3.4). Dieses Verfahren ist sehr aufwändig und unterliegt selbstverständlich auch dem bisherigen Synchronisationsproblem.

Hält man sich jedoch an die in Abschnitt 3.7.5 erläuterte Empfehlung, TCP-basierte Dienste der Anwendungsschicht mit der Header-Strukturveränderung zu versehen, so kann Routing durchgeführt werden. Auch bei einer Header-Strukturveränderung in der Transportschicht kann Routing bedingt durchgeführt werden.[15]

3.8 Vorteile der Header-Strukturveränderung

Die Header-Strukturveränderung bringt neben den besagten Problemen auch Vorteile mit sich. Dazu zählen selbstverständlich die beiden bereits erwähnten Schutzmechanismen gegen Storage Channels (also die Kennzeichnung legitimen Traffics und die Normalisierung des Traffics). Darüber hinaus gibt es aber noch weitere positive Eigenschaften zu nennen.

3.8.1 Erweiterung bestehender Dienste

Integriert man Header-Strukturveränderungen in Bibliotheken für die Benutzung von Netzwerkprotokollen oder als Erweiterungsbibliothek für bestehende Netzwerkkommunikation, so lassen sich bestehende Client-Server- und Peer-to-Peer-Systeme mit relativ geringem Aufwand umbauen. Die Header-Strukturveränderung wird zu diesem Zweck als eine Zwischenschicht im TCP/IP-Stack integriert, die im Userspace eines Betriebssystems abgewickelt werden kann. Übertragen auf das bisherige Beispiel der POP3-Verbindung würde dies bedeuten, dass zwischen Transport-Layer und Application-Layer eine Schicht für die Header-Strukturveränderung integriert wird.

Eine Erweiterung eines Dienstes auf die Header-Strukturveränderung muss jedoch immer auf allen Peers bzw. sowohl Client-seitig als auch server-seitig durchgeführt werden.

3.8.2 Betrieb bei eingeschränkten Zugriffsrechten

Während eine Normalisierung über eine Firewall im Kernelspace abläuft und ein Intrusion Detection System erweiterte Rechte für den Low-Level-Zugriff auf die Netzwerkschnittstellen benötigt, sind bei der Header-Strukturveränderung keine erweiterten Zugriffsrechte notwendig. Sie kann – wie im Proof-of-Concept-Code demonstriert wurde – im Userspace implementiert werden und läuft zudem mit den Rechten eines normalen Benutzers.

[15] Durch Techniken wie NAT mit Port Translation (NAPT), die mit Portbereichen arbeiten, und Firewalls, die auch die Protokolle der Transportschicht untersuchen oder gar normalisieren, ist davon jedoch abzuraten.

Hierzu ist anzumerken, dass ein Dienst, der auf einem privilegierten Port läuft, dennoch die erweiterten Rechte benötigen wird, um einen Socket an einen solchen Port zu binden. Dies hat aber zum einen nichts mit der Header-Strukturveränderung zu tun und kann zum anderen durch einen anschließenden Benutzerwechsel entschärft werden.[16]

3.9 Zusammenfassung

Die Header-Strukturveränderung verfolgt das Ziel, durch die Modifikation des Headeraufbaus von Netzwerkpaketen Storage Channels zu vermeiden. Dies wird erreicht, indem es einem Angreifer, der einen Storage Channel errichten will, nicht möglich gemacht wird, vorherzusagen, wie das nächste Netzwerkpaket aufgebaut werden muss. In der praktischen Umsetzung wird die Header-Strukturveränderung dabei zur Markierung von legitimen Traffic eingesetzt, der zudem normalisiert werden sollte.

Die Nachteile des Verfahrens liegen bei der Möglichkeit "false positives" und zusätzliche Rechenlast zu erzeugen. Außerdem entsteht ein zusätzliches Synchronisationsproblem und der Umgang mit Headern variabler Länge ist zwar möglich, erzeugt jedoch zusätzlichen Overhead.

Allerdings ist das Verfahren geeignet, um Storage Channels einzudämmen (indem es Möglichkeiten nimmt, diese zu erzeugen). Dazu ist zu sagen, dass bestehende Dienste für Storage Channels attraktiver sind, da eine Kommunikation über diese bekannten Dienste weniger auffällt als ein komplett neuer Dienst, den der Angreifer (etwa als Backdoor) auf einem neuen Port startet. Sind aber durch die Header-Strukturveränderung die Möglichkeiten eingeschränkt, diesen Dienst für Storage Channels zu verwenden, engt man den Spielraum des Angreifers effektiv ein.

Bestehende Dienste sind einfach für die Header-Strukturveränderung erweiterbar (wobei die Erweiterung jedoch Client- und Server-seitig durchgeführt werden muss) und im Gegensatz zur alternativen reinen Traffic-Normalisierung sind keine erweiterten Zugriffsrechte für dieses Verfahren notwendig.

[16] Dabei wechselt ein Dienst nach dem Socket-Setup mit einem Syscall – unter Linux etwa setuid() – zu einem weniger privilegierten Benutzer. Diese Sicherungsmaßnahme kann unter Umständen auch ein IDS-System durchführen. Für Kernelspace-Firewalls trifft das jedoch nicht zu.

A Quellcodelistings

Dieser Anhang enthält die Quellcode-Listings der im Rahmen dieser Diplomarbeit erstellten Programme. Für die bessere Druckdarstellung der Quellcodes wurden Zeilenumbrüche und Einrückungen angepasst. Die unmodifizierten Dateien finden sich auf der beiliegenden CD.

Zum Ausführen der Perl-Skripte werden die Perl-Module Net::RawIP und Net::ARP benötigt. Die C-Programme benötigen zur Übersetzung die Entwicklungsdateien von libpcap, einer Bibliothek zum Abfangen von Netzwerkpaketen auf dem Link-Layer, die von Programmen wie "tcpdump" benutzt wird. Den C-Programmen liegen entsprechende Makefiles bei. Getestet wurden alle Programme mit zwei Linux-Systemen, die sich nur durch ihre Prozessorarchitektur unterschieden:

- i686 bzw. x86_64 Architektur
- Kernel 2.6.27
- gcc 4.3.2 (4.3.2-1ubuntu11)
- Perl 5.10.0
- libpcap 0.9.8
- libnet-rawip-perl 0.23-2 (Ubuntu-Package)
- libnet-arp-perl 1.0.2-1 (Ubuntu-Package)

A.1 "phcc_detect"-Quellcode

A.1.1 phcc_detect.c

Listing A.1: Das Programm phcc_detect.c

```
1 /* phcc_detect.c, a proof of concept implementation of a
2  * algorithm that is able to find micro protocol headers
```

```
 3  * within protocol hopping covert channels and usual
 4  * storage channels. This implementation can detect
 5  * micro protocol sequence numbers in ICMPv4 packets.
 6  * v. 1.1, 2008-12-09
 7  * (C) 2008 Steffen Wendzel, <steffenwendzel (at) gmx.de>
 8  */
 9
10 #include <stdio.h>
11 #include <string.h>
12 #include <unistd.h>
13 #include <sys/time.h>
14 #include <sys/types.h>
15 #include <sys/socket.h>
16 #include <netinet/in.h>
17 #include <arpa/inet.h>
18 #include <netdb.h>
19 #include <net/if.h>
20 #include <netinet/if_ether.h>
21 #include <netinet/ip.h>
22 #include <netinet/ip_icmp.h>
23 #include <pcap.h>
24 #include <stdlib.h>
25 #include <err.h>
26 #include <signal.h>
27
28 #define FRAME_LEN       500
29
30 pcap_t *SetIf(char *);
31 void SigHandler(int);
32 void NewPkt(struct iphdr *ip, struct icmphdr *);
33
34 /* Init the interface for libpcap */
35 pcap_t *
36 SetIf(char *dev)
37 {
38   char errbuf[PCAP_ERRBUF_SIZE];
39   pcap_t *descr;
40   struct bpf_program filter;
41
```

```
42   if (!(descr = pcap_open_live(dev, FRAME_LEN, 1, 100, errbuf)))
43     err(1, "%s\n", errbuf);
44
45   /* Compile Pcap-Filter to only accept ICMP */
46   if (pcap_compile(descr, &filter, "icmp", 0, 0) == -1)
47     err(1, "pcap_compile error\n");
48
49   if (pcap_setfilter(descr, &filter))
50     err(1, "pcap_setfilter error\n");
51   return descr;
52 }
53
54 void
55 SigHandler(int sig)
56 {
57   if (sig == SIGINT)
58     exit(0);
59 }
60
61 void
62 alert()
63 {
64   printf("FOUND A POTENTIAL MICRO PROTOCOL SEQUENCE NUMBER!\n");
65 }
66
67 /* Compares ICMP Type+Code */
68 void
69 NewPkt(struct iphdr *ip_hdr, struct icmphdr *icmp_hdr)
70 {
71   static u_int32_t last_saddr = 0, last_daddr = 0;
72   static u_int8_t last_type = 0, last_code = 0;
73
74   if (last_saddr == 0 || last_daddr == 0) {
75         last_saddr = ip_hdr->saddr;
76         last_daddr = ip_hdr->daddr;
77         last_type = icmp_hdr->type;
78         last_code = icmp_hdr->code;
79         printf("-init done-\n");
80         return;
```

```
81    }
82
83    if (icmp_hdr->code > last_code) {
84      alert();
85      last_type = icmp_hdr->type;
86      last_code = icmp_hdr->code;
87      return;
88    }
89
90    /* local interface ping refuses this algorithm to work since an
91     * ICMP ECHO REPLY is sent after every ECHO REQUEST what means
92     * that there is a change between ICMP type 0 and 8 with equal
93     * src/dst addresses, what the algorithm would detect as an
94     * incremented micro protocol sequence number since the next
95     * check would not work.
96     */
97    if (ip_hdr->saddr == ip_hdr->daddr &&
98        (icmp_hdr->type == 8 || icmp_hdr->type == 0)) {
99           printf("-local interface ping-\n");
100          return;
101   }
102
103   /* check, if type or code incremented */
104   if (icmp_hdr->type > last_type) {
105     alert();
106     last_type = icmp_hdr->type;
107     last_code = icmp_hdr->code;
108   }
109 }
110
111 int
112 main(int argc, char *argv[])
113 {
114   pcap_t *descr;
115   const u_char *packet;
116   struct pcap_pkthdr hdr;
117   struct ether_header *eptr;
118   int datalink;
119   struct iphdr *ip_hdr;
```

```
120   struct icmphdr *icmp_hdr;
121   int icmphdr_offset;
122
123   if (argc != 2) {
124     printf("usage: %s [device]\n", argv[0]);
125     return 1;
126   }
127
128   if (signal(SIGINT, SigHandler) == SIG_ERR)
129     err(1, "signal error");
130
131   printf("RECEIVING MESSAGES - PRESS CTRL-C TO FINISH\n");
132
133   if ((descr = SetIf(argv[1])) == NULL)
134     return 1;
135
136   datalink = pcap_datalink(descr);
137
138   while (1) {
139     if ((packet = pcap_next(descr, &hdr)) != NULL) {
140       switch(datalink){
141       case DLT_EN10MB:
142         eptr = (struct ether_header *) packet;
143         if (ntohs(eptr->ether_type) == ETHERTYPE_IP) {
144           /* Must be ICMP since PCAP filter is set to "icmp" */
145
146           ip_hdr = (struct iphdr *) (packet
147                   + sizeof(struct ether_header));
148           icmphdr_offset = (ip_hdr->ihl * 4);
149           icmp_hdr = (struct icmphdr *) (((char *)ip_hdr)
150                     + icmphdr_offset);
151
152           NewPkt(ip_hdr, icmp_hdr);
153         }
154         break;
155       default:
156         fprintf(stderr, "datalink type %i not supported.\n",
157                         datalink);
158         exit(0);
```

```
159         }
160       }
161       fflush(stdout);
162     }
163     return 0;
164 }
```

A.1.2 microproto_sim.pl

Listing A.2: Das Skript microproto_sim.pl

```
 1 #!/usr/bin/perl
 2
 3 # microproto_sim.pl, simmulates a micro protocol sequence number
 4 # that can be detected using phcc_detect.c.
 5 # v. 1.0, 2008-12-09
 6 # (C) 2008 Steffen Wendzel, <steffenwendzel@gmx.de>
 7
 8 use strict;
 9 # Dependencies: CPAN/Net::RAWIP und CPAN/Net::ARP
10 use Net::RawIP;
11
12 # Send out ICMP packets with given ICMP type/code
13 sub SendICMPPkt($srcip, $dstip, $seqnr, $type, $code)
14 {
15   my $srcip = shift;
16   my $dstip = shift;
17   my $type = shift;
18   my $code = shift;
19   # ICMP-Payload like sent by Linux 2.6
20   my $icmp_payload = "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a"
21     . "\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29"
22     . "\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35";
23
24   my $icmppkt = Net::RawIP->new({
25     ip  => {
26       saddr => $srcip,
27       daddr => $dstip
28     },
29     icmp => {
```

```
30          type => $type,
31          code => $code,
32          id => 0x053c,
33          sequence => 0,
34          data => $icmp_payload
35        },
36    });
37
38    $icmppkt->send;
39 }
40
41 my $argc = $#ARGV + 1;
42
43 die "usage: [src IP] [dest IP]\n"
44   . "example: microprotocol_sim.pl 192.168.2.21 192.168.2.1\n"
45   unless $argc == 2;
46
47 my $srcip = $ARGV[0];
48 my $dstip = $ARGV[1];
49
50 # send different ICMP TYPES
51 my $i = 20;
52 while ($i < 30) {
53   &SendICMPPkt($srcip, $dstip, $i, 0);
54    $i+=2;
55 }
56
57 # send ICMP ECHO packet with different codes
58 $i = 0;
59 while ($i < 10) {
60   &SendICMPPkt($srcip, $dstip, 0, $i);
61    $i+=3;
62 }
```

A.2 “pct”-Quellcode

A.2.1 pct_sender.pl

Listing A.3: Das Skript pct_sender.pl

```
1 #!/usr/bin/perl
2
3 # pct_sender.pl, a sender tool for protocol channel messages (sends
4 # messages which can be received and interpreted by the tool
5 # 'pct_receiver.c').
6 # (C) 2008 Steffen Wendzel, steffenwendzel (at) gmx.net
7
8 use strict;
9 # Dependencies: CPAN/Net::RAWIP und CPAN/Net::ARP
10 use Net::RawIP;
11 use Net::ARP;
12
13 # Send out ICMP packets from $srcip to $dstip with
14 # sequence number $seqnr
15 sub SendICMPPkt($srcip, $dstip, $seqnr)
16 {
17   my $srcip = shift;
18   my $dstip = shift;
19   my $seqnr = shift;
20   # ICMP-Payload like sent by Linux 2.6
21   my $icmp_payload = "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a"
22     . "\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29"
23     . "\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35";
24
25   my $icmppkt = Net::RawIP->new({
26     ip  => {
27       saddr => $srcip,
28       daddr => $dstip
29     },
30     icmp => {
31       type => 8,
32       code => 0,
33       id => $seqnr,
34       sequence => 0,
```

```
35          data => $icmp_payload
36        },
37    });
38
39    $icmppkt->send;
40 }
41
42 # Send out ARP reply packets
43 sub SendARPPkt
44 {
45    my $dev = shift;
46    my $srcip = shift;
47    my $dstip = shift;
48    my $srcmac = shift;
49    my $dstmac = shift;
50
51    Net::ARP::send_packet($dev, $srcip, $dstip, $srcmac, $dstmac,
52                          'reply');
53 }
54
55 # Find out what bit combination is needed for the
56 # character $char and call the needed ARP/ICMP
57 # sending functions. Also send a parity bit here.
58 sub SendChar
59 {
60    my $char = shift;
61    my $seqnr = shift;
62    my $dev = shift;
63    my $srcip = shift;
64    my $dstip = shift;
65    my $srcmac = shift;
66    my $dstmac = shift;
67    my %codes = (
68     'A' => "00000", 'B' => "00001", 'C' => "00010", 'D' => "00011",
69     'E' => "00100", 'F' => "00101", 'G' => "00110", 'H' => "00111",
70     'I' => "01000", 'J' => "01001", 'K' => "01010", 'L' => "01011",
71     'M' => "01100", 'N' => "01101", 'O' => "01110", 'P' => "01111",
72     'Q' => "10000", 'R' => "10001", 'S' => "10010", 'T' => "10011",
73     'U' => "10100", 'V' => "10101", 'W' => "10110", 'X' => "10111",
```

```
74   'Y' => "11000", 'Z' => "11001", ' ' => "11010", '_' => "11011",
75   '-' => "11100", '$' => "11101", '.' => "11110", ',' => "11111");
76   my $i = 0;
77   my $bits = $codes{$char};
78   my $zero_bits = 0;
79
80   print "sending=" . $bits . "\n";
81
82   while ($i < length($bits)) {
83     $char = substr($bits, $i, 1);
84     print "sending bit " . $i . "=" . $char . " ";
85     if ($char eq "0") {
86       print "ARP";
87       &SendARPPkt($dev, $srcip, $dstip, $srcmac, $dstmac);
88       $zero_bits = $zero_bits + 1;
89     } elsif ($char eq "1") {
90       print "ICMP";
91       &SendICMPPkt($srcip, $dstip, $seqnr);
92       $seqnr = $seqnr + 1;
93     } else {
94       print "Illegal bit value!";
95       exit(1);
96     }
97     print "\n";
98     $i++;
99   }
100
101  # send parity bit
102  if (($zero_bits % 2) == 1) {
103    # send 1
104    print "Sending parity 1\n";
105    &SendICMPPkt($srcip, $dstip, $seqnr);
106    $seqnr = $seqnr + 1;
107  } else {
108    # send 0
109    print "Sending parity 0\n";
110    &SendARPPkt($dev, $srcip, $dstip, $srcmac, $dstmac);
111  }
112
```

```
113   return $seqnr;
114 }
115
116 my $argc = $#ARGV + 1;
117
118 die "usage: [device] [src IP] [dest IP] [src MAC] [dest MAC] "
119  . "[ICMP start seq] [payload]\n"
120  . "example: pct_send eth0 192.168.2.21 192.168.2.1 "
121  . "00:1d:09:35:87:c4 00:1d:09:35:87:c5 0x053c \"Hello World\""
122  unless $argc == 7;
123
124 my $dev = $ARGV[0];
125 my $srcip = $ARGV[1];
126 my $dstip = $ARGV[2];
127 my $srcmac = $ARGV[3];
128 my $dstmac = $ARGV[4];
129 my $seqnr = hex($ARGV[5]);
130 my $payload = uc ($ARGV[6]);
131 my $max_payload_len = 511;
132 my $i = 0;
133
134 if (length($payload) > $max_payload_len) {
135   print "Payload too long. Max. " . $max_payload_len
136         . " Bytes allowed!\n";
137   exit 1;
138 }
139
140 while ($i < length($payload)) {
141   my $char = substr($payload, $i, 1);
142   print "sending payload[" . $i . "]=" . $char . "\n";
143   $seqnr = &SendChar($char, $seqnr, $dev, $srcip, $dstip, $srcmac,
144                      $dstmac);
145   print "Seqnr now=".$seqnr."\n";
146   $i++;
147 }
148
149 exit 0
```

A.2.2 pct_receiver.c

Listing A.4: Das Programm pct_receiver.c

```
1 /* pct_receiver.c, a receiver tool for protocol channel messages
2  * (receives messages sent by the tool 'pct_sender.pl').
3  * (C) 2008 Steffen Wendzel, steffenwendzel (at) gmx.net
4  */
5
6 #include <stdio.h>
7 #include <string.h>
8 #include <unistd.h>
9 #include <sys/time.h>
10 #include <sys/types.h>
11 #include <sys/socket.h>
12 #include <netinet/in.h>
13 #include <arpa/inet.h>
14 #include <netdb.h>
15 #include <net/if_arp.h>
16 #include <net/if.h>
17 #include <netinet/if_ether.h>
18 #include <pcap.h>
19 #include <stdlib.h>
20 #include <err.h>
21 #include <signal.h>
22
23 #define FRAME_LEN       500
24 #define BUFFERSIZ       512
25 #define PCTYP_ARP       0
26 #define PCTYP_ICMP      1
27
28 char plaintext_buf[BUFFERSIZ] = { '\0' };
29 short plaintext_buf_pos = 0;
30
31 pcap_t *SetIf(char *);
32 void SigHandler(int);
33 char ExtractCode(short int[6]);
34 void NewPkt(short);
35
36 /* Prepare interface settings for libpcap */
```

```
37 pcap_t *
38 SetIf(char *dev)
39 {
40   char errbuf[PCAP_ERRBUF_SIZE];
41   pcap_t *descr;
42   struct bpf_program filter;
43
44   if (!(descr = pcap_open_live(dev, FRAME_LEN, 1, 100, errbuf)))
45     err(1, "%s\n", errbuf);
46
47   /* Compile Pcap-Filter to only accept ICMP and ARP */
48   if (pcap_compile(descr, &filter,
49         "arp or (icmp and icmp[icmptype] = 8)", 0, 0) == -1)
50     err(1, "pcap_compile error\n");
51
52   if (pcap_setfilter(descr, &filter))
53     err(1, "pcap_setfilter error\n");
54   return descr;
55 }
56
57 /* Re-create the original content based on the protocol
58  * information received. Also do a parity check here.
59  */
60 char
61 ExtractCode(short int recv_buf[6])
62 {
63   /* 'A' => "00000", 'B' => "00001", 'C' => "00010", 'D' => "00011",
64      'E' => "00100", 'F' => "00101", 'G' => "00110", 'H' => "00111",
65      'I' => "01000", 'J' => "01001", 'K' => "01010", 'L' => "01011",
66      'M' => "01100", 'N' => "01101", 'O' => "01110", 'P' => "01111",
67      'Q' => "10000", 'R' => "10001", 'S' => "10010", 'T' => "10011",
68      'U' => "10100", 'V' => "10101", 'W' => "10110", 'X' => "10111",
69      'Y' => "11000", 'Z' => "11001", ' ' => "11010", '_' => "11011",
70      '-' => "11100", '$' => "11101", '.' => "11110", ',' => "11111");
          */
71   static char codes[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ _-$.,";
72   short val = 0;
73   short zero_bits = 5;
74
```

```
75    if (recv_buf[0]) { val += 16; zero_bits--; }
76    if (recv_buf[1]) { val += 8; zero_bits--; }
77    if (recv_buf[2]) { val += 4; zero_bits--; }
78    if (recv_buf[3]) { val += 2; zero_bits--; }
79    if (recv_buf[4]) { val += 1; zero_bits--; }
80
81    if ((zero_bits % 2) != recv_buf[5]) {
82      fprintf(stderr, "PARITY CHECK FAILED! Connection "
83                      "desynchronized!\n");
84      SigHandler(0);
85      exit(1);
86    }/* else printf("parity check: OK\n");*/
87
88    printf("  val=%hi = %c\n", val, codes[val]);
89    return codes[val];
90  }
91
92  /* Adds a new packet to the buffer */
93  void
94  NewPkt(short typ)
95  {
96    static short cnt = 0;
97    static short int recv_buf[6] = { '\0' };
98
99    switch(typ) {
100   case PCTYP_ARP:
101     recv_buf[cnt] = PCTYP_ARP;
102     break;
103   case PCTYP_ICMP:
104     recv_buf[cnt] = PCTYP_ICMP;
105     break;
106   default:
107     err(1, "Error on NewPkt(): unknown typ value\n");
108   }
109
110   if (cnt == 5) {
111     /* Received 6 Bits (a full character + parity bit) */
112     char c;
113
```

```
114     /*printf("INPUT=%hi%hi%hi%hi%hi-%hi\n", recv_buf[0], recv_buf[1],
115     recv_buf[2], recv_buf[3], recv_buf[4], recv_buf[5]);*/
116
117     c = ExtractCode(recv_buf);
118     plaintext_buf[plaintext_buf_pos] = c;
119     plaintext_buf_pos++;
120     if (plaintext_buf_pos == BUFFERSIZ) {
121       fprintf(stderr, "Error: Buffer full. Aborting.\n");
122       /* Output buffer */
123       SigHandler(0);
124       /* Reset buffer */
125       bzero(plaintext_buf, BUFFERSIZ);
126       plaintext_buf_pos = 0;
127     }
128     bzero(recv_buf, sizeof(recv_buf));
129     cnt = -1;
130   }
131   fflush(stdout);
132   cnt++;
133 }
134
135 void
136 SigHandler(int sig)
137 {
138   printf("Received signal %i\n", sig);
139   printf("Received Message: %s\n", plaintext_buf);
140   /* Only exit on CTRL-C, not on sig==0 (full buffer+reset) */
141   if (sig == SIGINT)
142     exit(0);
143 }
144
145 int
146 main(int argc, char *argv[])
147 {
148   pcap_t *descr;
149   const u_char *packet;
150   struct pcap_pkthdr hdr;
151   struct ether_header *eptr;
152   int datalink;
```

```
153
154   if (argc != 2) {
155     printf("usage: %s [device]\n", argv[0]);
156     return 1;
157   }
158
159   if (signal(SIGINT, SigHandler) == SIG_ERR)
160     err(1, "signal error");
161
162   printf("RECEIVING MESSAGES - PRESS CTRL-C TO FINISH\n");
163
164   if ((descr = SetIf(argv[1])) == NULL)
165     return 1;
166
167   datalink = pcap_datalink(descr);
168
169   while (1) {
170     if ((packet = pcap_next(descr, &hdr)) != NULL) {
171       switch(datalink){
172       case DLT_EN10MB:
173         eptr = (struct ether_header *) packet;
174         if (ntohs(eptr->ether_type) == ETHERTYPE_IP) {
175           /* Must be ICMP since PCAP filter is set to
176            * "arp or icmp[icmptype]=8" */
177           NewPkt(PCTYP_ICMP);
178         } else  if (ntohs(eptr->ether_type) == ETHERTYPE_ARP) {
179           NewPkt(PCTYP_ARP);
180         } else {
181           /* Not part of Protocol Channel */
182         }
183         break;
184       default:
185         fprintf(stderr, "datalink type %i not supported.\n",
186                         datalink);
187         exit(0);
188       }
189     }
190     fflush(stdout);
191   }
```

```
192   return 0;
193 }
```

A.3 “hsc”-Quellcode

Der Proof-of-Concept-Code für die Header-Strukturveränderung teilt sich in zwei Programme auf, die sich in den Unterverzeichnissen “hsc_recv” und “hsc_send” befinden. In diesen Verzeichnissen muss zur Übersetzung die jeweilige Makefile verwendet werden.

A.3.1 secret.h

Der Quellcode dieser Datei wird von beiden Programmen benötigt.

Listing A.5: Das Programm secret.h

```
1 /* A static header structure secret used for the
2  * demonstration within the proof of concept implementation
3  * of hsc_send and hsc_recv.
4  */
5 char hsc_secret[] = "irngoierungoweifit3408ncjkasnd09g"
6                     "n048gn3w408fn408gn340g3j04x,41^-8"
7                     "40fn4g8n340g8j40gf8ng23eornbeornb"
8                     "4ghgjrt09hj45noernbvwebinejrb40ng";
9
10 /* A special Padding character */
11 #define SPECIAL_CHAR     /*'_'*/ 0x03
12 #define NUM_PERMUTATIONS 24
13 int hsc_permutations[NUM_PERMUTATIONS][4] = {
14                 { 1,2,3,4 },
15                 { 1,2,4,3 },
16                 { 1,3,2,4 },
17                 { 1,3,4,2 },
18                 { 1,4,2,3 },
19                 { 1,4,3,2 },
20
21                 { 2,1,3,4 },
22                 { 2,1,4,3 },
23                 { 2,3,1,4 },
24                 { 2,3,4,1 },
25                 { 2,4,1,3 },
26                 { 2,4,3,1 },
```

```
27
28                  { 3,1,2,4 },
29                  { 3,1,4,2 },
30                  { 3,2,1,4 },
31                  { 3,2,4,1 },
32                  { 3,4,1,2 },
33                  { 3,4,2,1 },
34
35                  { 4,1,2,3 },
36                  { 4,1,3,2 },
37                  { 4,2,1,3 },
38                  { 4,2,3,1 },
39                  { 4,3,1,2 },
40                  { 4,3,2,1 }
41          };
```

A.3.2 hsc_recv.c

Listing A.6: Das Programm hsc_recv.c

```
1 /* Header Structure Changing implementation for a receiver.
2  *
3  * This programm simmulates a POP3 server with a header structure
4  * changing functionality included. It works in combination with
5  * the tool hsc_send.
6  *
7  * (C) 2009 Steffen Wendzel, steffenwendzel (at) gmx.net
8  * Last Modified: 2009-Apr-15
9  * Version: 1.1
10 */
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <unistd.h>
15 #include <string.h>
16 #include <strings.h>
17 #include <err.h>
18 #include <sys/types.h>
19 #include <sys/socket.h>
```

```
20 #include <arpa/inet.h>
21 #include <netinet/in.h>
22
23 #include "../secret.h"
24
25 /* local port for the client to connect to */
26 #define LISTEN_PORT     110
27 /* size of the receive buffer */
28 #define RECV_BUFSIZE            100*1024
29
30 /* Create a LISTEN socket here and wait for a remote
31  * connection.
32  */
33 int
34 socket_setup()
35 {
36         int listen_socket;
37         struct sockaddr_in sa;
38         socklen_t salen = sizeof(struct sockaddr_in);
39         int yup = 1;
40         int local_accept_socket;
41
42         /* Open a local TCP socket on Port LISTEN_PORT */
43         bzero(&sa, sizeof(sa));
44         sa.sin_addr.s_addr = INADDR_ANY;
45         sa.sin_port = htons(LISTEN_PORT);
46         sa.sin_family = AF_INET;
47         salen = sizeof(struct sockaddr_in);
48
49         if ((listen_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
50                 err(1, "socket()");
51
52         setsockopt(listen_socket, SOL_SOCKET, SO_REUSEADDR, &yup,
53                 sizeof(yup));
54         if (bind(listen_socket, (struct sockaddr *) &sa, salen) < 0)
55                 err(1, "bind()");
56
57         /* Wait for a connection and then try to connect to the
58          * server:port */
```

```
59          if (listen(listen_socket, 5) < 0)
60                  err(1, "listen()");
61          printf("Waiting for a connection on Port %i...\n",
62                  LISTEN_PORT);
63
64          if ((local_accept_socket = accept(listen_socket,
65                  (struct sockaddr *) &sa, &salen)) < 0)
66                  err(1, "accept()");
67          close(listen_socket);
68          printf("Connection established.\n");
69
70          return local_accept_socket;
71 }
72
73 /* Re-Calculate the original buffers content what means
74  * to remove the applied header structure changings sent
75  * by the client.
76  */
77 char *
78 de_hsc_buf(char *buf_hsc)
79 {
80          static int hsc_ptr = 0;
81          char *ret_buf;
82          int i;
83          char tmp_buf[5];
84          char *next_pos;
85          char secret_char;
86          int *cur_permut;
87          int q;
88          int len;
89
90          if (!(ret_buf = (char *) calloc(strlen(buf_hsc) + 1,
91                          sizeof(char))))
92                  err(1, "calloc()");
93
94          for (i = 0; i <= (int) strlen(buf_hsc) / 4; i++) {
95                  bzero(tmp_buf, sizeof(tmp_buf));
96                  next_pos = buf_hsc + (i*4);
97                  len = (strlen(next_pos) > 4 ? 4 : strlen(next_pos));
```

```
 98                 if (len == 0)
 99                         continue;
100                 strncpy(tmp_buf, next_pos, len);
101                 /* remove the SPECIAL_CHARs */
102                 for (q = 0; q < 4; q++) {
103                         if (tmp_buf[q] == SPECIAL_CHAR)
104                                 tmp_buf[q] = '\0';
105                 }
106
107                 /* now permutate the 4 characters in tmp_buf by
108                  * using the information found in the secret buffer.
109                  */
110                 secret_char = hsc_secret[hsc_ptr];
111                 hsc_ptr++;
112                 /* prevent an interger overflow here */
113                 if (hsc_ptr == sizeof(hsc_secret))
114                         hsc_ptr = 0;
115
116                 cur_permut = hsc_permutations[
117                         secret_char % NUM_PERMUTATIONS];
118                 /* apply de-permutation */
119                 for (q = 0; q < 4; q++) {
120                         int was_moved_to;
121                         /* find out where did the HSC client put
122                          * element [q] to?
123                          * e.g. q=0:  plain[0] -> buf_hsc[2] - 1
124                          * => get the element back from where it
125                          * was moved to. */
126                         was_moved_to = cur_permut[q] - 1;
127                         ret_buf[q + (i * 4)] = tmp_buf[was_moved_to];
128                 }
129         }
130         return ret_buf;
131 }
132
133 /* Check what is in the buffer and return an answer that
134  * looks like a real POP3 server just to demonstrated the
135  * functionality of Header Structure Changing in combination
136  * with POP3.
```

```
137  */
138 char *
139 handle_recv_buf(char *recv_buf, int *close_connection)
140 {
141         char *answer_buf;
142         char *hsc_free_buf;
143 
144         hsc_free_buf = de_hsc_buf(recv_buf);
145 
146         /* RETR will never have a real message available */
147         if (strncasecmp(hsc_free_buf, "RETR ", 5) == 0) {
148                 answer_buf = "-ERR Invalid message number.\r\n";
149         } else if (strncasecmp(hsc_free_buf, "STAT\r\n", 6) == 0) {
150                 answer_buf = "+OK 0 0\r\n";
151         /* Accept all user/password combinations because some clients
152          * like Sylpheed are unable to make POP3-Connections without
153          * an authentication.
154          */
155         } else if (strncasecmp(hsc_free_buf, "USER ", 5) == 0) {
156                 answer_buf = "+OK Please enter password\r\n";
157         } else if (strncasecmp(hsc_free_buf, "PASS ", 5) == 0) {
158                 answer_buf = "+OK mailbox locked and ready\r\n";
159         /* DELE must always return an -ERR since there are no real
160          * messages within this simmulation. */
161         } else if (strncasecmp(hsc_free_buf, "DELE ", 5) == 0) {
162                 answer_buf = "-ERR no such message\r\n";
163         } else if (strncasecmp(hsc_free_buf, "NOOP\r\n", 6) == 0) {
164                 answer_buf = "+OK\r\n";
165         } else if (strncasecmp(hsc_free_buf, "RSET\r\n", 6) == 0) {
166                 answer_buf = "+OK\r\n";
167         } else if (strncasecmp(hsc_free_buf, "QUIT\r\n", 6) == 0) {
168                 answer_buf = "+OK closing.\r\n";
169                 *close_connection = 1;
170         } else {
171                 answer_buf = "-ERR Unknown command.\r\n";
172                 printf("Received command is unknown. "
173                         "This could be an attack.\n");
174                 *close_connection = 1;
175         }
```

```
176         free(hsc_free_buf);
177         return answer_buf;
178 }
179
180 int
181 main()
182 {
183         int sockfd;
184         int len;
185         char *answer_buf;
186         char buf[RECV_BUFSIZE] = { '\0' };
187         int close_connection = 0;
188         char greeting_msg[] =
189         "+OK Header-Structure-Changing POP3-Server-Simmulator\r\n";
190
191         /* Create a open socket, accept only one connection a time,
192          * what is enough to demonstrate the functionality.. */
193         sockfd = socket_setup();
194
195         /* Send greeting message */
196         if (send(sockfd, greeting_msg, strlen(greeting_msg),
197                         0) == -1)
198                 err(1, "send() greeting message");
199
200         /* Mainloop */
201         while (!close_connection) {
202                 len = recv(sockfd, buf, sizeof(buf) - 1, 0);
203                 if (len <= 0)
204                         err(1, "recv");
205                 /* Handle the received buffer and return a answer */
206                 answer_buf = handle_recv_buf(buf, &close_connection);
207                 if (!send(sockfd, answer_buf, strlen(answer_buf), 0))
208                         err(1, "send");
209                 bzero(buf, sizeof(buf));
210         }
211         /* disconnect the client after 'quit\r\n' and shutdown
212          * the server */
213         close(sockfd);
214         return 0;
```

```
215 }
```

A.3.3 hsc_send.c

Listing A.7: Das Programm hsc_send.c

```
 1 /* Header Structure Changing Implementation for a Sender.
 2  *
 3  * This Programm makes it harder to establish storage channels
 4  * within an unidirectional POP3 connection from the client to a
 5  * server. It accepts the client connection proccess (e.g. telnet)
 6  * via a local socket and forwards packets to the server. The
 7  * packets are modified using header struture changing. Also the
 8  * content gets normalized before sending.
 9  * This prevents storage channels in parts of the header.
10  * Example: Client tries to create a storage channel by modifing the
11  * lower/upper case letters of POP3 commands: ReTr, RETr, rEtr, ...
12  * but after the normalization, all commands look like 'RETR'.
13  *
14  * This works only, if the server process is able to handle header
15  * structure changing packets (the tool hsc_recv implements that).
16  *
17  * (C) 2009 Steffen Wendzel, steffenwendzel (at) gmx.net
18  * Last Modified: 2009-Apr-15
19  * Version: 1.2
20  */
21
22 #include <stdio.h>
23 #include <stdlib.h>
24 #include <unistd.h>
25 #include <string.h>
26 #include <err.h>
27 #include <sys/types.h>
28 #include <sys/socket.h>
29 #include <arpa/inet.h>
30 #include <netinet/in.h>
31 #include <errno.h>
32 #include <ctype.h>
33
34 #include "../secret.h"
```

```
35
36 /* local port for the client to connect to */
37 #define LOCAL_LISTEN_PORT       10001
38 /* size of the receive buffer */
39 #define RECV_BUFSIZE            100*1024
40
41 void
42 usage()
43 {
44         extern char *__progname;
45
46         printf("usage: %s -s server-ip -p server-port\n",
47                 __progname);
48         exit(1);
49 }
50
51 /* Create a local LISTEN socket here and wait for
52  * a client connection.
53  */
54 int
55 make_local_connection()
56 {
57         int listen_socket;
58         struct sockaddr_in sa;
59         socklen_t salen = sizeof(struct sockaddr_in);
60         int yup = 1;
61         int local_accept_socket;
62
63         /* Open a local TCP socket on Port LOCAL_LISTEN_PORT */
64         bzero(&sa, sizeof(sa));
65         if (!inet_pton(AF_INET, "127.0.0.1", &sa.sin_addr))
66                 err(1, "inet_pton(127.0.0.1)");
67         sa.sin_port = htons(LOCAL_LISTEN_PORT);
68         sa.sin_family = AF_INET;
69         salen = sizeof(struct sockaddr_in);
70
71         if ((listen_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
72                 err(1, "socket()");
73
```

```
74          setsockopt(listen_socket, SOL_SOCKET, SO_REUSEADDR, &yup,
75                  sizeof(yup));
76          if (bind(listen_socket, (struct sockaddr *) &sa, salen) < 0)
77                  err(1, "bind()");
78
79          /* Wait for a connection and then try to connect to the
80           * server:port */
81          if (listen(listen_socket, 5) < 0)
82                  err(1, "listen()");
83          printf("Waiting for local connection on Port %i...\n",
84                  LOCAL_LISTEN_PORT);
85
86          if ((local_accept_socket = accept(listen_socket,
87                          (struct sockaddr *) &sa, &salen)) < 0)
88                  err(1, "accept()");
89          close(listen_socket);
90          printf("Local connection established.\n");
91
92          return local_accept_socket;
93  }
94
95  /* Connect to the POP3 server after the client connection
96   * is already established.
97   */
98  int
99  connect_to_server(char *server_ip, int server_port)
100 {
101         int server_socket;
102         struct sockaddr_in sa;
103         socklen_t salen = sizeof(struct sockaddr_in);
104
105         bzero(&sa, sizeof(sa));
106         if (!inet_pton(AF_INET, server_ip, &sa.sin_addr))
107                 err(1, "inet_pton() for server IP");
108         sa.sin_port = htons(server_port);
109         sa.sin_family = AF_INET;
110         salen = sizeof(struct sockaddr_in);
111
112         if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
```

```
113                 err(1, "socket() for server socket");
114
115         if (connect(server_socket, (struct sockaddr *) &sa,
116                         sizeof(struct sockaddr_in)) != 0)
117                 err(1, "connect() to server failed");
118         printf("Connection to server %s:%i established\n",
119                         server_ip, server_port);
120
121         return server_socket;
122 }
123
124 /* Add HSC to a buffer -- since POP3 has only a header with
125  * no payload on the client side, the whole buffer needs to
126  * get modified for HSC.
127  */
128 void
129 buf_to_hsc(char *plain_buf, char *hsc_buf)
130 {
131         static int hsc_ptr = 0;
132         int i, cnt;
133         char tmp_buf[5];
134         char *next_pos;
135         char secret_char;
136         int *cur_permut;
137         int len;
138
139         /* this loop always changes the structure of max. four
140          * bytes.
141          */
142         for (i = 0; i <= (int) strlen(plain_buf) / 4; i++) {
143                 bzero(tmp_buf, sizeof(tmp_buf));
144                 next_pos = plain_buf + (i*4);
145                 len = (strlen(next_pos) > 4 ? 4 : strlen(next_pos));
146                 if (len == 0)
147                         continue;
148                 strncpy(tmp_buf, next_pos, len);
149
150                 /* make chars upper case to prevent storage channels
151                  * based on upper/lower case characters. */
```

```
152                 for (cnt = 0; cnt < (int) strlen(tmp_buf); cnt++)
153                         if (tmp_buf[cnt] >= 'a'
154                                         && tmp_buf[cnt] <= 'z')
155                                 tmp_buf[cnt] = toupper(tmp_buf[cnt]);
156
157                 /* fill the rest of the buffer with special bytes */
158                 if (len < 4) {
159                         if (len < 2) tmp_buf[1] = SPECIAL_CHAR;
160                         if (len < 3) tmp_buf[2] = SPECIAL_CHAR;
161                         if (len < 4) tmp_buf[3] = SPECIAL_CHAR;
162                 }
163
164                 /* now permutate the 4 characters in tmp_buf by
165                  * using the information found in the secret buffer.
166                  */
167                 secret_char = hsc_secret[hsc_ptr];
168                 hsc_ptr++;
169                 /* prevent an interger overflow here */
170                 if (hsc_ptr == sizeof(hsc_secret))
171                         hsc_ptr = 0;
172
173                 cur_permut = hsc_permutations[
174                         secret_char % NUM_PERMUTATIONS];
175                 /* apply permutation */
176                 hsc_buf[cur_permut[0] - 1 + (i * 4)] = tmp_buf[0];
177                 hsc_buf[cur_permut[1] - 1 + (i * 4)] = tmp_buf[1];
178                 hsc_buf[cur_permut[2] - 1 + (i * 4)] = tmp_buf[2];
179                 hsc_buf[cur_permut[3] - 1 + (i * 4)] = tmp_buf[3];
180         }
181 }
182
183 /* Send data from sockfd_from to sockfd_to.
184  * if hsc_flag equals '1': Add header structure changing
185  * to the buffer before forwarding it.
186  */
187 void
188 send_from_to(int sockfd_from, int sockfd_to, int hsc_flag)
189 {
190         int len;
```

```
191         char buf[RECV_BUFSIZE] = { '\0' };
192         char hsc_buf[RECV_BUFSIZE] = { '\0' };
193
194         len = recv(sockfd_from, buf, sizeof(buf) - 1, 0);
195         if (len == -1) {
196                 err(1, "recv");
197         } else if (len == 0) {
198                 printf("Connection closed.\n");
199                 exit(0);
200         } else {
201                 if (hsc_flag) {
202                         buf_to_hsc(buf, hsc_buf);
203                         if (!send(sockfd_to, hsc_buf,
204                                         strlen(hsc_buf), 0))
205                                 err(1, "send");
206                 } else {
207                         if (!send(sockfd_to, buf, len, 0))
208                                 err(1, "send");
209                 }
210         }
211 }
212
213 /* Make sure that the connection is still alive */
214 int
215 check_connection_state(int sockfd)
216 {
217         struct sockaddr_in sa_conn_check;
218         socklen_t salen = sizeof(struct sockaddr_in);
219
220         if (getpeername(sockfd, (struct sockaddr *)
221                         &sa_conn_check, &salen) != 0)
222                 return 0;
223         return 1;
224 }
225
226 int
227 main(int argc, char *argv[])
228 {
229         int ch;
```

```
230         char *server_ip = NULL;
231         int server_port = 0;
232         int server_socket;
233         int local_accept_socket;
234         fd_set fds;
235         int peak;
236         int connected;
237         struct timeval tm;
238
239         while ((ch = getopt(argc, argv, "s:p:")) != -1) {
240                 switch (ch) {
241                 case 'p':
242                         server_port = atoi(optarg);
243                         break;
244                 case 's':
245                         if (!(server_ip = (char *) calloc(
246                           strlen(optarg) + 1, sizeof(char))))
247                                 err(1, "calloc()");
248                         strncpy(server_ip, optarg, strlen(optarg));
249                         break;
250                 case 'h':
251                         /* FALLTROUGH */
252                 default:
253                         usage();
254                         /* NOTREACHED */
255                 }
256         }
257
258         if (server_ip == NULL)
259                 usage();
260
261         if (server_port == 0)
262                 usage();
263
264         /* Create a local socket and wait for a connection from the
265          * client */
266         local_accept_socket = make_local_connection();
267
268         /* Connect to the Server */
```

```
269         server_socket = connect_to_server(server_ip, server_port);
270
271         FD_ZERO(&fds);
272         /* prepare for select() -- this prevents a blocking on
273          * select() on a already closed connection.
274          */
275         tm.tv_sec = 1;
276         tm.tv_usec = 0;
277
278         /* receive and forward until process gets a termination
279          * signal */
280         connected = 1;
281         while (connected) {
282                 /* check, if the sockets are still connected */
283                 if (check_connection_state(server_socket) == 0 ||
284                 check_connection_state(local_accept_socket) == 0) {
285                         connected = 0;
286                         continue;
287                 }
288
289                 FD_SET(server_socket, &fds);
290                 FD_SET(local_accept_socket, &fds);
291                 /* select() needs the max. file descriptor value */
292                 peak = (server_socket > local_accept_socket
293                         ? server_socket : local_accept_socket);
294
295                 if (select(peak + 1, &fds, NULL, NULL, &tm) == -1)
296                         err(1, "select");
297
298                 /* If the local socket is set: read the data and
299                  * forward it to the server. Make use of Header
300                  * Structure Changing here. */
301                 if (FD_ISSET(local_accept_socket, &fds))
302                         send_from_to(local_accept_socket,
303                                 server_socket, 1);
304
305                 /* If the server sent something: forward it to the
306                  * local client */
307                 if (FD_ISSET(server_socket, &fds))
```

```
308                         send_from_to(server_socket,
309                                 local_accept_socket, 0);
310         }
311
312         close(server_socket);
313         close(local_accept_socket);
314         return 0;
315 }
```

Literaturverzeichnis

[AHSAN02] Ahsan, K.: Covert Channel Analysis and Data Hiding in TCP/IP, Master-Thesis (University of Toronto), 2002.

[ALEX06] Alexander, M.: Netzwerke und Netzwerksicherheit. Das Lehrbuch, Hüthig, 1. Auflage, 2006.

[BORLAND08] Borland, T.: Guide to Encrypted Dynamic Covert Channels, 24. Dezember 2008. URL: http://turboborland.blogspot.com/2008/12/guide-to-encrypted-dynamic-covert.html, zuletzt geprüft am 10. Mai 2009.

[BCKK05] Barni, M., Cox, I., Kalker, T. et. al.: Digital Watermarking. 4th International Workshop, IWDW 2005, Siena, Italy, September 15-17, 2005, Proceedings, Springer, 1. Auflage, August 2005.

[BEJTLI07] Bejtlich, R.: Analyzing Protocol Hopping Covert Channel Tool, 14. November 2007. URL: http://taosecurity.blogspot.com/2007/11/analyzing-protocol-hopping-covert.html, zuletzt geprüft am 10. Mai 2009.

[BISHOP02] Bishop, M.: Computer Security: Art and Science, Addison-Wesley, 9th Printing, Oktober 2006.

[BRILL04] Brill, M.: Mathematik für Informatiker – Einführung an praktischen Beispielen aus der Welt der Computer, Hanser, 2. Auflage, 2005.

[CAST06] Castro, S. und das Gray World Team: How to cook a covert channel, Hackin9 01/2006, S. 50–57.

[CHESW04] Cheswick, W. R., Bellovin, S. M. und Rubin, A. D.: Firewalls und Sicherheit im Internet: Schutz vor cleveren Hackern, Addison-Wesley, 2. Auflage, 2004.

[COXGERG04] Cox, K. und Gerg, C.: Managing Security with Snort and IDS Tools, O'Reilly, 1. Auflage, 2004.

[DAEM97] daemon9: LOKI2 (the implementation), Phrack Magazine, Volume 7, Issue 51 September 1997. URL: http://gray-world.net/papers/projectloki2.txt, zuletzt geprüft am 10. Mai 2009.

[DOD85] Department of Defence: Trusted Computer System Evaluation Criteria (TCSEC, DoD 5200.28-STD), 26. Dezember 1985. URL: http://csrc.nist.gov/publications/history/dod85.pdf, zuletzt geprüft am 10. Mai 2009.

[ECKERT08] Eckert, C.: IT-Sicherheit. Konzepte, Verfahren, Protokolle., Oldenbourg Wissenschaftsverlag GmbH, 5. Auflage, 2008.

[ESSER05] Eßer, H.-G.: Ausnutzung verdeckter Kanäle am Beispiel eines Web-Servers, Diplomarbeit (RWTH Aachen), Februar 2005.

[FIELD97] Fielding, R. et al.: Hypertext Transfer Protocol – HTTP/1.1, RFC 2068 (Network Working Group), Juni 1999.

[GASGOD06] Gaspar, A., Godwin C.: Root-kits & loadable kernel modules: exploiting the Linux kernel for fun and (educational) profit, Journal of Computing Sciences in Colleges, Volume 22, Issue 2, Dezember 2006, S. 244–250.

[GOLTZ03] Goltz, J. P.: Under the radar: A look at three covert communications channels, GIAC security essentials (GSEC), 2003. URL: http://mmert.org/goltz/GSEC/Jim_Goltz_GSEC_edit.pdf, zuletzt geprüft am 10. Mai. 2009.

[JONES01] Jones, K. J.: Loadable Kernel Modules, ";login:"-Magazine, Volume 26, Number 7, November 2001, S. 43–49.

[LIGOCH08] Li, Z., Goyal, A. und Chen, Y.: Honeynet-based Botnet Scan Traffic Analysis, Advances in Information Security/Botnet Detection, Vol. 36, Springer, 2008, S. 25–44.

[LINGM02] Lingmann, T.: Datenverschlüsselung. Sichere Kommunikation mit Linux und BSD, Computer und Literatur, 1. Auflage, 2002.

[MURD05] Murdoch, S. J. und Lewis, S.: Embedding Covert Channels into TCP/IP, Information Hiding, 7th International Workshop, IH 2005, Barcelona, Spain, June 6-8, 2005. Revised Selected Papers, Lecture Notes in Computer Science 3727/2005, Springer, 2005, S. 247–261.

[NORTHC07] Northcutt, S., Baker, A. R., Kohlenberg, T., Esler, J., Beale, J., Caswell, B.: Snort: IDS and IPS toolkit, Syngress, 1. Auflage, 2007.

[OBSDPF08] Die OpenBSD Entwickler: PF: Scrub (Packet Normalization), Juli 2008. URL: http://www.openbsd.org/faq/pf/scrub.html, zuletzt geprüft am 10. Mai 2009.

[POSTEL81A] Postel, J.: Internet Control Message Protocol. DARPA Internet Program Protocol Specification, RFC 792, Network Working Group, September 1981.

[POSTEL81B] Postel, J.: Internet Protocol. DARPA Internet Program Protocol Specification, RFC 791, Information Sciences Institute (University of Southern California), September 1981.

[RUTK04] Rutkowska, J.: The Implementation of Passive Covert Channels in the Linux Kernel, Dezember 2004. URL: http://invisiblethings.org/papers/passive-covert-channels-linux.pdf, zuletzt geprüft am 10. Mai 2009.

[SCHNEI06] Schneier, B.: Angewandte Kryptographie. Protokolle, Algorithmen und Sourcecode in C, Pearson Studium, 2. Aufl., Nachdr. 2006.

[SIMMONS83] Simmons, G. J.: The Prisoner's Problem and the Subliminal Channel, Advances in Cryptology: Proceedings of CRYPTO '83, Plenum Press, 1984, S. 51–67.

[SPENN05] Spenneberg, R.: Intrusion Detection und Prevention mit Snort 2 & Co: Einbrüche auf Linux-servern erkennen und verhindern, Pearson Education, 1. Auflage, 2005.

[STODLE09] Stødle, D.: Ping Tunnel – For those times when everything else is blocked, 17. April 2009. URL: http://www.cs.uit.no/~daniels/PingTunnel/, zuletzt geprüft am 10. Mai 2009.

[WASH98] Washburn, K. und Evans, J.: TCP/IP – Aufbau und Betrieb eines TCP/IP Netzes, Addison-Wesley, 2. Auflage, 1. korrigierter Nachdr. 1998.

[WEND07] Wendzel, S.: Protocol Hopping Covert Channels. An Idea and the Implementation of a Protocol Switching Covert Channel, 11. November 2007.

URL: http://www.wendzel.de/dr.org/files/Papers/protocolhopping.txt, zuletzt geprüft am 10. Mai 2009.

[WEND08] Wendzel, S.: Protocol Hopping Covert Channels, Hakin9 03/2008, S. 20–21.

[WONG00] Wong, C.: HTTP kurz & gut, O'Reilly, 1. Auflage, 2002.

[YADALI08] F. Yarochkin, S.-Y. Dai, C.-H. Lin, Y. Huang, S.-Y. Kuo: Towards Adaptive Covert Communication System, Dep. of Electrical Engineering, National Taiwan University, 2008. URL: http://o0o.nu/files/yarochkin_adaptivecovertchannel.pdf, zuletzt geprüft am 10. Mai 2009.

[ZANDER06] Zander, S., Armitage, G. und Branch, P.: Covert Channels in the IP Time To Live Field, Centre for Advanced Internet Architectures (Swinburne University of Technology), Dezember 2006.

[ZWINKY00] Zwicky, E. D., Cooper, S. und Chapman, D. B.: Building Internet Firewalls, O'Reilly, 2. Auflage, 2000.

Index

Communication Phase, 15
Covert Channel, 1
    Anpassungsfähigkeit, 14
    Rauschfreiheit, 5

Einkapselung, 2

Header-Strukturveränderung, 33
    Angreifer, 41
    Beispielkommunikation, 40
    Desynchronisation, 53
    Empfangsfunktion, 36, 38
    Erweitertes theoretisches Modell, 37
    False Positives, 52
    Geheimnis, 35
    Geheimnisaustausch, 49
    Headerlänge, 52
    Normalisierung, 46
    Permutation, 36, 47
    Praktisches Modell, 44
    Rechenlast, 53
    Routing, 53
    Sendefunktion, 36, 38
    Theoretisches Grundmodell, 33
    Traffic-Markierung, 45
    Zugriffsrechte, 54
hping3, 30
HTTP Inspect, 51

Lokaler Covert Channel, 3

LOKI2, 8, 11

Network Environment Learning Phase, 14
Normalisierung, 46, 51
NUSHU, 1, 5

OpenBSD, 51

Passive Covert Channel, 5
pct, 27
pf, 51
phcct, 8, 10, 11
Ping Tunnel, 11
Protocol Channel, 21
    Definition, 21
    Desynchronisation, 26
    Detektion, 31
    Eigenschaften, 22
    Fragmentierung, 26
    Nutzungsprobleme, 25
Protocol Hopping Covert Channel, 7
    Definition, 7
    Detektion, 15
    LOKI2, 8
    Mikroprotokoll, 10
Protokollwechsel, 7

Scrubbing, 51
Sequenznummer, 11, 15
Snort, 15, 51
Steganographie, 1

Storage Channel, 4
Subliminal Channel, 1

TCP ISN Covert Channel, 5
Timing Channel, 4
Tunnel, 2

Verdeckte Kommunikation, 1
    Verwendung, 2

ERKLÄRUNG

gemäß §35 Abs. (7) der Rahmenprüfungsordnung für die Fachhochschulen in Bayern (RaPO) in der jeweils gültigen Fassung.

Ich versichere, dass ich diese Diplomarbeit selbständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Kempten, den 13.05.2009 .....

Steffen Wendzel

ERMÄCHTIGUNG

Hiermit ermächtige ich die Hochschule Kempten zur Veröffentlichung der nachstehenden Kurzzusammenfassung meiner Arbeit, z.B. auf gedruckten Medien oder auf einer Internetseite.

Kempten, den 13.05.2009 .....

Steffen Wendzel

Kurz-Zusammenfassung der Arbeit

Die Diplomarbeit beschäftigt sich mit verdeckten Kommunikationskanälen (Covert Channels) in Netzwerken und untersucht dabei zwei Themen. Zum Einen sind dies die Möglichkeiten, die ein Wechsel des Übertragungsprotokolls bei verdeckten Kommunikationskanälen bietet. Dabei werden die seit 1997 – zumindest in einfacher Form – vorhandenen Protocol Hopping Covert Channel sowie die in dieser Arbeit erstmals vorgestellte Form der Protocol Channel betrachtet. Außerdem werden die Detektionsmöglichkeiten der genannten Covert Channel-Arten untersucht. Im Zweiten Teil der Arbeit wird die Idee der Headerstrukturänderungen in Netzwerkprotokollen zur Vermeidung von Storage Channeln in Protokoll-Headern vorgestellt. Es folgt eine Analyse der praktischen Umsetzbarkeit von Headerstrukturänderungen.