Testen von Firewalls

Eine UDP-/ICMP-Erweiterung für fwtest

Beat Strasser <b8@student.ethz.ch>

Semester-Arbeit

Winter-Semester 2005/06
ETH Zürich, 28. Februar 2006

Betreuerin: Diana Senn <dsenn@inf.ethz.ch>

Professor: David Basin <basin@inf.ethz.ch>

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information Security
www.infsec.ethz.ch

Zusammenfassung

Ein gutes Analyse-Programm zum Testen einer Firewall muss auch gängige Protokolle der Netzwerk- und Transport-Schicht unterstützen können. Dies umfasst insbesondere IP inkl. ICMP sowie TCP und UDP.

Diese Semesterarbeit befasst sich mit einer UDP- und ICMP-Erweiterung des Firewall-Test-Programms *fwtest* von Gerry Zaugg. Neben einer generellen Einführung in UDP und ICMP wird die Funktionsweise von *fwtest* erläutert und auf die nötigen Design-Entscheide und Implementierungstechniken eingegangen. Umfassendere Tests zeigen die volle Funktionalität der neuen Version *fwtest*-1.0 auf.

Abstract

A sophisticated firewall testing tool should also support the most common protocols of the network and transport layer. This includes in particular IP incl. ICMP as well as TCP and UDP.

This semester thesis deals with an extension to the firewall testing tool *fwtest* developed by Gerry Zaugg to handle UDP and ICMP. In addition to a general introduction to these protocols, the *fwtest*'s internal functionality is elucidated and important decisions in design and implementation are elaborated on. Extensive tests show the new version's full functionality.

Inhaltsverzeichnis

1. Einleitung 7
1.1. Aufgabe . . . 7
1.2. Umstrukturierung . . . 7

2. Protokolle 8
2.1. User Datagram Protocol (UDP) . . . 8
2.2. Internet Control Message Protocol (ICMP) . . . 8

3. Design 13
3.1. Übersicht von fwtest in der Version 0.5 . . . 13
3.2. Mehrere Protokolle gleichzeitig . . . 15
3.3. Paket-Spezifizierung . . . 15
3.4. UDP-Pakete . . . 16
3.5. ICMP-Pakete . . . 17
3.6. Paket-ID-Übertragung . . . 18

4. Implementierung 20
4.1. Mehrere Protokolle gleichzeitig . . . 20
4.2. Parsen der Paket-Spezifikationen . . . 20
4.2.1. Datenstruktur . . . 21
4.2.2. Eigenschaften . . . 21
4.2.3. Präprozessor M4 . . . 22
4.3. UDP . . . 22
4.3.1. Pakete generieren . . . 23
4.3.2. Pakete abfangen . . . 23
4.4. ICMP . . . 23
4.4.1. Pakete generieren . . . 23
4.4.2. Pakete abfangen . . . 24

5. Ergebnisse 25
5.1. Testumgebung . . . 25
5.2. Zwei Testfälle . . . 25

6. Abschluss 29
6.1. Zusammenfassung . . . 29
6.2. Fazit . . . 29
6.3. Ziele für die Zukunft . . . 30
6.4. Danksagung . . . 30

A. Format Test-Pakete 31

B. Standard-Makros 32

C. README **33**

D. Aufgabenstellung **41**

E. Zeitplan **44**

Abbildungsverzeichnis

1. Aufbau UDP-Datagramm 8
2. Paket-Aufbau “ICMP Echo (Reply)” 9
3. Paket-Aufbau “ICMP Destination unreachable” 10
4. Paket-Aufbau “ICMP Redirect” 10
5. Paket-Aufbau “ICMP Time Exceeded” 11
6. Paket-Aufbau “ICMP Timestamp (Reply)” 12
7. Firewall-Test-Szenario mit *fwtest*-0.5 13
8. Testpaket-Spezifikation: Beispiel eines TCP-Verbindungsaufbaus . . . 13
9. Programm-Logik von *fwtest*-0.5 (aus [Zau05]) 14
10. Aufgebaute Ereignis-Datenstruktur 21
11. Testumgebung 25
12. `test.tp`: Spezifikation unserer Testfälle im neuen Format 26
13. Firewall-Regeln 26
14. Ausgabe der Statistik von *fwtest*-1.0 27
15. Logdatei von *fwtest*-1.0 27
16. *TCPdump*-Ausgabe aus dem Netzwerk 172.16.70.0/29 28
17. *TCPdump*-Ausgabe aus dem Netzwerk 192.168.72.0/29 28

Tabellenverzeichnis

1. Codes vom Typ “ICMP Destination unreachable” 10
2. Codes vom Typ “ICMP Redirect” 11
3. Codes vom Typ “ICMP Time Exceeded” 11
4. Nötige Werte aller Protokoll-Typen 16

1. Einleitung

Firewalls trennen Netze und schützen somit unter anderem auch interne, private Netzwerke vor anderen Netzwerken, welche nicht der eigenen Kontrolle unterliegen, wie zum Beispiel das Internet. Eine sichere Firewall ist zentral für viele Organisationen, die eine Verbindung ins Internet benötigen.

Eine Firewall ist aber nicht generell sicher. Sicherheits-Policies müssen für jeden einzelnen Fall individuell erstellt und durch die Firewall umgesetzt werden. Damit man der Firewall vertrauen kann, muss man sicher sein, dass sie die Policies erfüllt. Deshalb muss eine Firewall sorgfältig getestet werden.

Das Testen umfasst zwei Teile: Erstens müssen Testfälle entwickelt werden, die eine Firewall auf Konformität gegenüber der Sicherheits-Policy testen können. Zweitens müssen die Firewalls mit Hilfe dieser Testfälle überprüft werden.

Gerry Zaugg hat im Januar 2005 im Rahmen einer Diplomarbeit [Zau05] ein Test-Programm namens *fwtest* entwickelt, das letztere Aufgabe meistert. Leider unterstützte es nur Testfälle, die TCP-Pakete[1] enthalten.

1.1. Aufgabe

Unsere Aufgabe (genauer beschrieben in Appendix D) war es in dieser Semesterarbeit *fwtest* zu erweitern, so dass es auch Testfälle für UDP und ICMP (siehe Kapitel 2.1 und 2.2) bewältigen kann.

Dazu musste ein neues Format für die Testpaket-Spezifikationen aller implementierter Protokoll-Typen entworfen werden. Dies erforderte ein Anpassen des Parsers, damit dieser das Format akzeptiert und die Eingabe richtig internalisiert.

Weiter mussten Generierungs-Funktionen erstellt werden, so dass UDP- und ICMP-Pakete erstellt und versandt werden können. Eingegangene UDP- und ICMP-Pakete sollten korrekt analysiert und mit den Spezifikationen verglichen werden.

Zum Abschluss war es wichtig, *fwtest* selber umfassend zu testen. Das Firewall-Test-Programm sollte schliesslich nach dem Umbau wieder korrekt funktionieren.

1.2. Umstrukturierung

Zeitgleich mit dieser Erweiterung wurde an einer grösseren Umstrukturierung gearbeitet. Adrian Schüpbach baute *fwtest* so um, dass eine Programm-Instanz alleine die gesamte Test-Aufgabe bewältigen[2] und zusätzlich mit NAT umgehen kann. Seine Dokumentation [Sch06] ist in jedem Fall eine hilfreiche Lektüre bei der Einarbeitung in *fwtest*.

Dies hatte zur Folge, dass wir am selben Repository arbeiteten und gewisse Aufgaben gemeinsam erledigen mussten. Hier sei insbesondere die Entwicklung des neuen Dateiformats (siehe Kapitel 3.3 und Appendix A) und das Problem der Paket-ID-Übertragung (siehe Kapitel 3.6) erwähnt.

[1] Transmission Control Protocol, siehe [Pos81b].

[2] Siehe dazu die Übersicht über den 'alten' Zustand von *fwtest* in Kapitel 3.1.

2. Protokolle

An dieser Stelle wenden wir uns den Protokollen zu, um die es in dieser Arbeit hauptsächlich geht. Wir verschaffen uns einen groben Überblick, so dass wir die späteren Probleme beim Design und bei der Implementierung verstehen können.

2.1. User Datagram Protocol (UDP)

UDP ist ein Protokoll für die simple Daten-Übertragung über eine Ende-zu-Ende Verbindung. Ein Endpunkt wird definiert durch eine IP-Adresse und eine Port-Nummer. UDP ist wie TCP in der Transport-Schicht angesiedelt, ist aber nicht auf Zuverlässigkeit ausgelegt und bietet auch keine Flusssteuerung.

Deshalb ist UDP sehr geeignet für Anwendungen, die zeitkritisch sind und die den Verlust eines einzelnen Paketes verkraften können. Beispiele dazu sind Audio/Video Streaming, Voice over IP und Netzwerk-Spiele. Auch das Domain Name System (DNS) arbeitet auf UDP-Basis.

Der UDP-Header ist nur klein (8 Bytes) und enthält die Ursprungs- und Ziel-Port-Nummer, die Gesamt-Länge des UDP-Datagramms in Bytes sowie eine Prüfsumme. Da das Protokoll zustandslos ist, ist die Angabe des Ursprung-Ports optional und kann auf 0 gesetzt werden. Die IP-Protokollnummer für UDP ist 17.

Abbildung 1 zeigt, wie ein UDP-Paket aussieht (siehe auch [Pos80]).

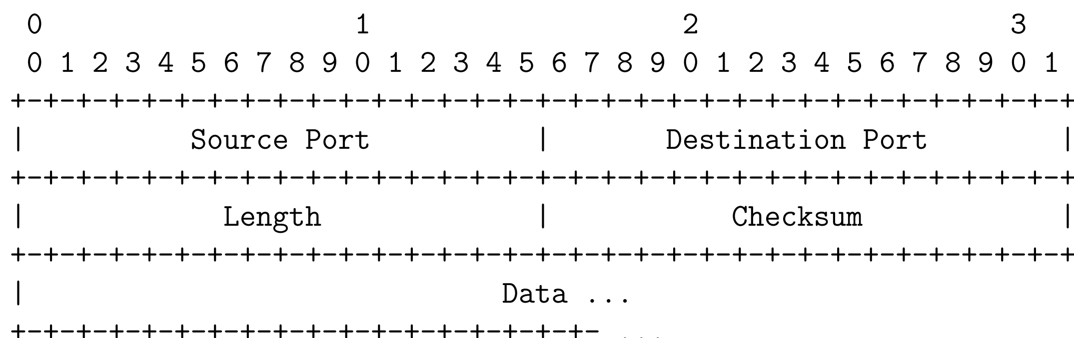


Abbildung 1: Aufbau UDP-Datagramm

2.2. Internet Control Message Protocol (ICMP)

ICMP dient in IP-Netzen zum Austausch von Fehler- und Informationsmeldungen. Obwohl es eigentlich auf IP-Paketen basiert, ist ICMP vor allem ein Bestandteil von IP und für die Diagnose im Netzwerk gedacht. Jeder Router und jeder PC im Netzwerk sollte daher ICMP verstehen und umsetzen können. Die Benutzer selber werden mit ICMP nur selten in Kontakt kommen (zum Beispiel bei *ping*).

Die IP-Protokollnummer für ICMP ist 1. ICMP selbst ist wieder unterteilt in verschiedene Nachrichten-Typen. Die ersten vier Bytes des ICMP-Pakets sind für alle

Typen gleich (Typ, Code und Prüfsumme). Der Code ist ein typ-spezifischer Parameter.

In [Pos81a] sind die wichtigsten ICMP-Typen spezifiziert. Generell unterscheidet man zwischen Fehler- und Informationsmeldungen. Fehlermeldungen enthalten den ganzen IP-Header der Original-Nachricht plus 64bit IP-Daten, damit der ursprüngliche Sender die Meldung richtig einordnen kann. Solche Fehlermeldungen dürfen keinesfalls andere ICMP-Pakete auslösen (siehe [Bra89], Kapitel 3.2.2).

Wir befassen uns in dieser Arbeit nur mit den fünf wichtigsten ICMP-Typen:

ICMP Echo / Echo Reply Dieser Meldungstyp (siehe dazu auch Abb. 2) wird zum Beispiel von *ping* verwendet, um herauszufinden, ob ein anderer Rechner am Netzwerk angeschlossen ist oder nicht. Dazu wird eine Echo-Anfrage (Typ 8) mit oder ohne Daten an einen Zielrechner gesandt. Dieser sendet die genau gleichen Daten in einer Echo-Antwort (Typ 0) zurück.

Zur genaueren Identifizierung eines Paketes wird eine Identifikations- und eine Sequenz-Nummer angegeben. Der Code muss 0 sein.

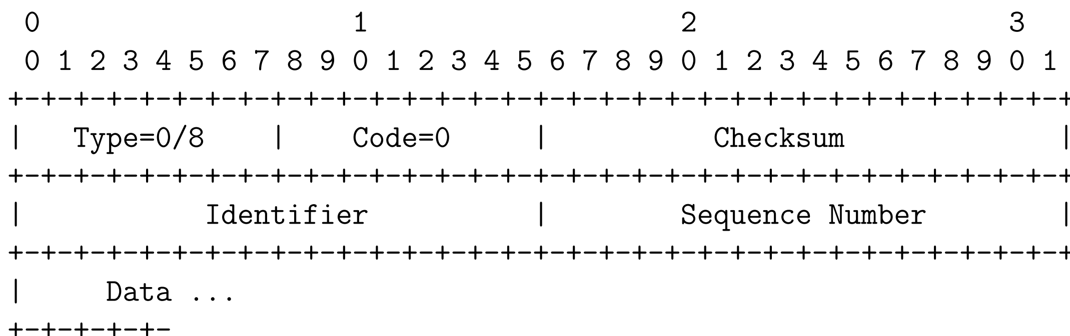


Abbildung 2: Paket-Aufbau "ICMP Echo (Reply)"

ICMP Destination unreachable Falls das Ziel eines IP-Paketes nicht erreichbar ist, sei es das Netzwerk, der Rechner, das Protokoll-Modul oder der Prozess-Port, so kann eine ICMP-Fehlermeldung vom Typ 3 zurückgesandt werden. [Pos81a] spezifiziert nur sechs Fälle/Codes, [Bra89] in Kapitel 3.2.2.1 weitere sieben (siehe Tabelle 1).

Die MTU[3] wird nur bei Code 4 spezifiziert. Dieser Fall tritt auf, wenn ein Paket das "Don't-Fragment"-Flag (DF) gesetzt hat, aber zu gross für die Übertragung ist (die Paket-Grösse überschreitet die MTU des nächsten Links) und folglich eine Fragmentierung nötig wäre. Somit wird nun eine ICMP-Meldung vom Typ 3 und Code 4 zurückgesandt mit der Angabe der grösstmöglichen MTU (siehe [MD90]), damit der Ursprungsrechner das Paket mit angepasster Grösse erneut senden kann.

Abbildung 3 zeigt den Aufbau einer solchen Fehlermeldung.

[3]Maximum Transmission Unit: hier die maximale Paket-Grösse für eine bestimmte Netzwerk-Verbindung (Link), so dass das Paket übertragen werden kann, ohne fragmentiert zu werden.

Code	*Bedeutung*
0	net unreachable
1	host unreachable
2	protocol unreachable
3	port unreachable
4	fragmentation needed and DF set
5	source route failed
6	destination network unknown
7	destination host unknown
8	source host isolated *(obsolete)*
9	communication with destination network administratively prohibited
10	communication with destination host administratively prohibited
11	network unreachable for type of service
12	host unreachable for type of service

Tabelle 1: Codes vom Typ "ICMP Destination unreachable"

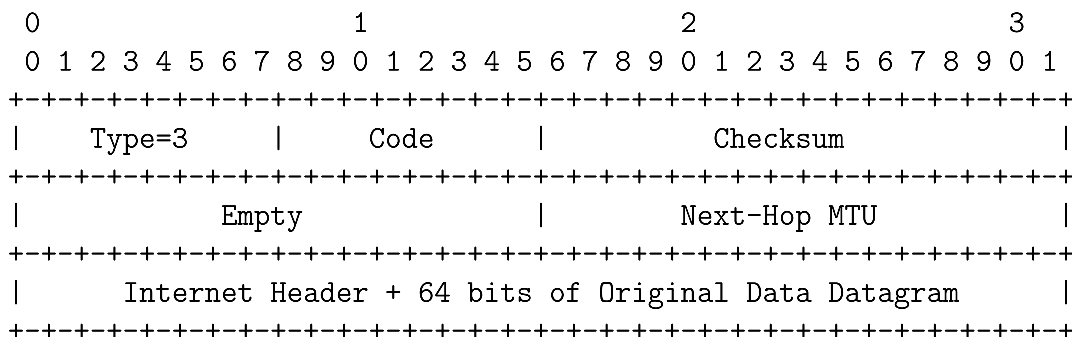


Abbildung 3: Paket-Aufbau "ICMP Destination unreachable"

ICMP Redirect Abbildung 4 zeigt den Nachrichten-Typ 5, mit welchem ein Gateway einen Rechner über bessere Routen informieren kann. Dazu gibt er die IP-Adresse des für den Rechner "näheren" Gateways an. Der Code spezifiziert für welche Art Dienst und Netzwerk die Information gilt (siehe Tabelle 2).

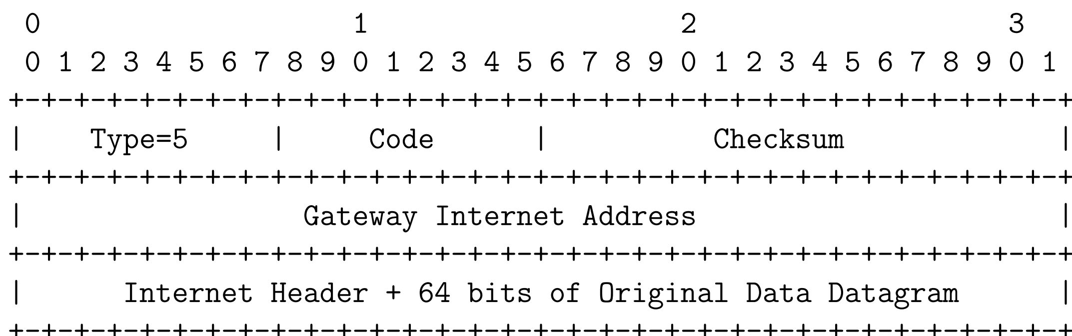


Abbildung 4: Paket-Aufbau "ICMP Redirect"

Code	*Bedeutung*
0	redirect for network error
1	redirect for host error
2	redirect for type of service and network error
3	redirect for type of service and host error

Tabelle 2: Codes vom Typ "ICMP Redirect"

ICMP Time Exceeded Die "Time-to-Live"-Angabe (TTL) im IP-Header verhindert eine, durch Routing-Fehler durchaus mögliche, unendlich lange Weiterleitung eines IP-Pakets: Der Ursprungsrechner setzt die TTL auf einen Anfangs-Wert (bei Linux zum Beispiel 64) und jeder Gateway im Netz, der das Paket verarbeitet, dekrementiert die TTL um 1.

Wenn nun bei einem Gateway die TTL des IP-Pakets abläuft, die TTL also null ist, muss er das Paket ignorieren und dem Ursprungsrechner eine "ICMP Time Exceeded"-Fehlermeldung (Typ 11) zurücksenden mit dem Code 0 (siehe auch Tabelle 3).

Falls ein Rechner beim Zusammensetzen fragmentierter Pakete zu lange auf ein Fragment warten muss, schickt er ebenfalls eine solche Fehlermeldung zurück, allerdings mit dem Code 1.

Der in Abbildung 5 dargestellte Paket-Aufbau sieht etwa ähnlich aus wie bei den anderen ICMP-Fehlermeldungen.

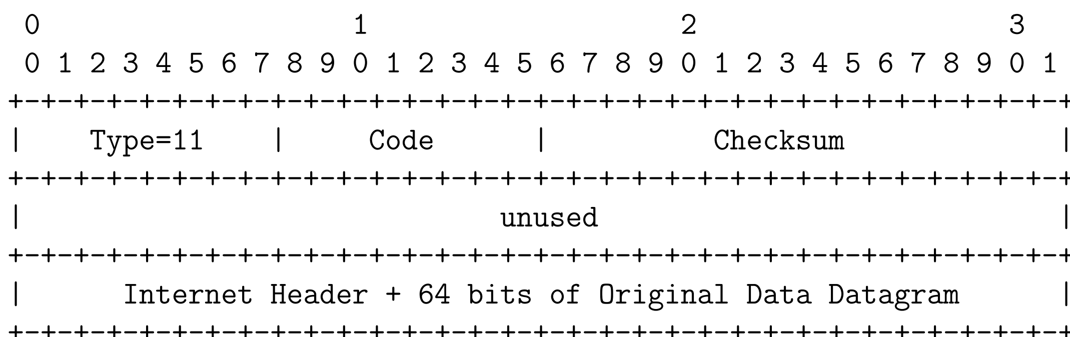


Abbildung 5: Paket-Aufbau "ICMP Time Exceeded"

Code	*Bedeutung*
0	time to live exceeded in transit
1	fragment reassembly time exceeded

Tabelle 3: Codes vom Typ "ICMP Time Exceeded"

ICMP Timestamp / Timestamp reply Dieser ICMP-Typ kann zum Beispiel für eine simple Uhrensynchronisation[4] verwendet werden. Rechner A sendet seinen aktuellen Zeitstempel in einem "Timestamp"-Paket (Typ 13) an Rechner B. Dieser sendet ein "Timestamp Reply" (Typ 14) zurück an A mit dem erhaltenen Zeitstempel, der Ankunftszeit des Original-Pakets sowie der neuen Sendezeit.

Beide Typen sehen vom Aufbau her gleich aus, wie in Abbildung 6 gezeigt.

Zur besseren Identifizierung kann eine Identifikations- und eine Sequenznummer angegeben werden.

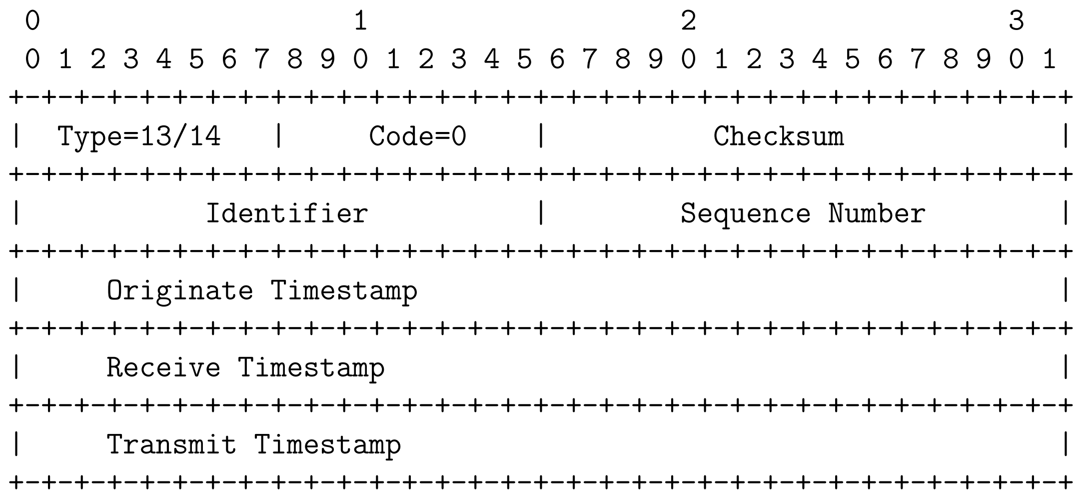


Abbildung 6: Paket-Aufbau "ICMP Timestamp (Reply)"

[4]Nicht zu verwechseln mit dem "Network Time Protocol" (NTP), das auf UDP basiert.

3. Design

Um zu verstehen, welche Änderungen im Design von *fwtest* für eine Erweiterung von UDP und ICMP nötig waren, wollen wir uns erst einen Überblick über die Struktur des Programmes von *fwtest* in der alten Version 0.5 verschaffen:

3.1. Übersicht von fwtest in der Version 0.5

Das Testen einer Firewall sieht in der Version 0.5 folgendermassen aus (siehe auch Abbildung 7):

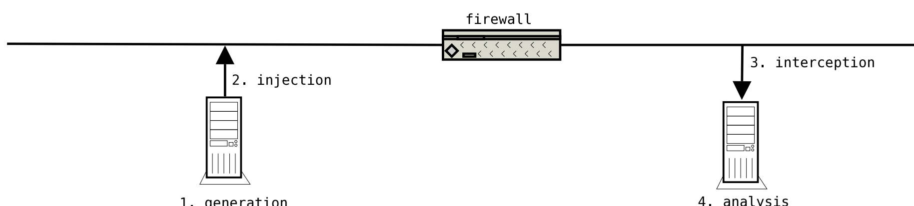


Abbildung 7: Firewall-Test-Szenario mit *fwtest*-0.5

Es werden zwei Instanzen A und B von *fwtest* gestartet, beide mit den gleichen Paket-Spezifikationen. Die Instanz A auf dem ersten Testrechner bedient die eine Seite der zu testenden Firewall, B auf dem zweiten Rechner die andere[5]. Die beiden Maschinen werden zeitlich mit NTP[6] synchronisiert, damit die *fwtest*-Instanzen im Testablauf jeweils das gleiche Paket bearbeiten, das heisst je nachdem versenden oder erwarten.

Initialisierung Während der Initialisierung werden die Testpaket-Spezifikationen geparst. Diese sehen in *fwtest*-0.5 aus wie zum Beispiel in Abbildung 8. Jede Zeile spezifiziert ein Testpaket. Es können nur Felder in den Kopfdaten des Protokolls festgelegt werden (es werden keine eigentlichen Datenblöcke gesandt).

```
#id expect time     srcip   dstip   srcprt dstprt flags seq ack
 1  OK     12:00:00 1.2.3.4 4.3.2.1 1025   25     S     60  -
 2  NOK    12:00:01 4.3.2.1 1.2.3.4 25     1025   SA    70  61
 3  NOK    12:00:02 1.2.3.4 4.3.2.1 1025   25     A     61  71
```

Abbildung 8: Testpaket-Spezifikation: Beispiel eines TCP-Verbindungsaufbaus

Das Protokoll wird anhand der Datei-Endung bestimmt (*fwtest*-0.5 unterstützt nur TCP).

[5]*fwtest* kann ein ganzes Netzwerk simulieren, indem es für jede IP im angegebenen Subnetz eine (virtuelle) MAC-Adresse erstellt und auf ARP-Anfragen reagiert.
[6]Network Time Protocol, siehe [Mil92].

Aus diesen Spezifikationen wird eine Ereignis-Datenstruktur aufgebaut. Danach werden alle involvierten Testpakete zum Senden vorbereitet, das heisst die gesamten Netzwerk-Pakete mit den Daten aller Protokoll-Schichten werden generiert.

Als nächstes werden Timer für jede Sende- oder Empfangs-Zeit gesetzt, welche dann das Programm kurz in den Sende- oder in den Analyse-Modus versetzen. Der Timer-Interrupt für ein Empfangs-Ereignis erfolgt jeweils erst bei der Zeit $t + \delta$, also nach einem bestimmten Zeitlimit, weil die Pakete im Netzwerk schliesslich auch eine gewisse, wenn auch relativ kurze, Zeit benötigen.

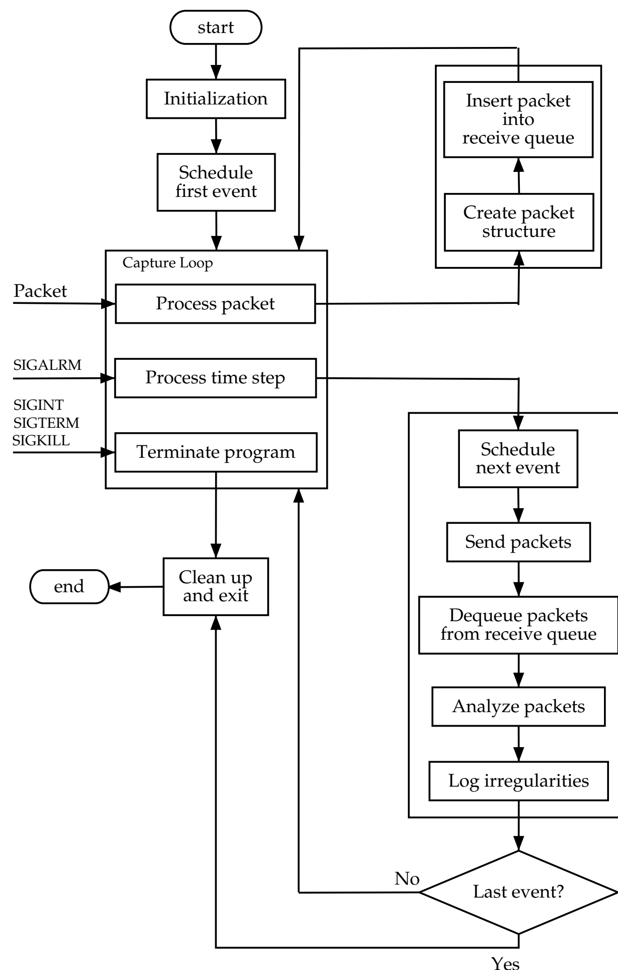


Abbildung 9: Programm-Logik von *fwtest*-0.5 (aus [Zau05])

Programm-Logik Nach der Initialisierungsphase betritt *fwtest* den "Abfang"-Modus, welcher auf eingehende Pakete wartet und diese dann abspeichert. Dies ist der Hauptmodus. Siehe dazu auch die Abbildung 9.

Zu den spezifizierten Zeiten werden Interrupts ausgelöst, wobei dann kurz in den Sende- oder Empfangs-Modus gewechselt wird, um die entsprechenden Pakete zu bearbeiten. Danach wird wiederum auf Pakete im Netzwerk gehorcht.

Senden Um Pakete in das Netzwerk einzuschleusen, bedient sich *fwtest* der *Libnet*-Bibliothek. Diese versteckt alle Lowlevel-Details der Netzwerk-Programmierung, um die wir uns in *fwtest* nicht kümmern wollen. Sie bietet Methoden an, um Pakete vieler verschiedener Protokolle aller möglichen Schichten zu generieren und zu versenden.

Die vollständigen Pakete wurden bereits am Anfang erstellt, damit sie alle beim Senden auch wirklich gleich zur Verfügung stehen. Zu den angegebenen Zeiten werden die Pakete dann in das Netzwerk geschickt.

Empfangen *fwtest* benutzt die Bibliothek *Libpcap*, um Pakete im Netzwerk im Original-Zustand und in vollem Umfang (inkl. Ethernet-Daten) abfangen zu können.

Analyse Sobald *fwtest* vom Timer in den Analyse-Modus geschaltet wird, analysiert es die eingegangenen Pakete und überprüft dann die Daten, bzw. vor allem die Kopfdaten der Internet- und Transport-Schicht, mit den Angaben in der Spezifikation der erwarteten Pakete.

Logging Um dem Benutzer die genauen Resultate des gesamten Tests ersichtlich zu machen, wird jede Abweichung zur Spezifikation genauestens im Logfile notiert.

3.2. Mehrere Protokolle gleichzeitig

Zusätzliche Protokolle zu TCP waren in *fwtest* bereits von Anfang an vorgesehen. Implementiert wurde zwar nur TCP, aber überall im Quell-Code wiesen `switch`-Blöcke auf die Unterscheidung der verschiedenen Protokolle hin. Somit waren wichtige Anhaltspunkte gegeben, wo wir mit der Implementierung ansetzen mussten.

In der alten Version bestimmte eine Testpaket-Datei das Protokoll, welches dann für den ganzen Programmlauf festgelegt war. Eine solche Beschränkung macht aber wenig Sinn im Zusammenhang mit ICMP: ICMP-Fehlermeldungen werden zum Beispiel generiert, wenn normale Pakete nicht ordnungsgemäss ausgeliefert werden können. Wollen wir dabei nicht nur auf fiktive Pakete verweisen, so müssen wir fähig sein, in einem Programmlauf mehrere Protokolle gleichzeitig zuzulassen. Ausserdem ist es bequemer alle Tests in einer Datei spezifizieren zu können als für jedes Protokoll eine eigene Testdatei zu schreiben.

Das Format der Testpaket-Datei sollte also so geändert werden, dass für jedes Testpaket auch das Protokoll angegeben werden muss.

3.3. Paket-Spezifizierung

Die Entwicklung des neuen Formats wurde hauptsächlich von Adrian Schüpbach in seiner Semesterarbeit [Sch06] erledigt, welche, wie bereits in der Einleitung erwähnt, gleichzeitig zu dieser Arbeit statt fand. Dabei trugen wir von unserer Seite nur bei, welche Werte wir für die Paket-Generierung der neuen Protokolle zusätzlich

benötigten (siehe auch Tabelle 4). Wir präsentieren hier deshalb nur die endgültige Version des neuen Formats (siehe Appendix A).

Dieses Unterkapitel erklärt, was wir für die Behandlung der neuen Protokolle, wie sie in Kapitel 2 eingeführt wurden, für Information benötigen.

Generell braucht es natürlich erst einmal die protokoll-unspezifischen Angaben wie ID, Empfangs-Erwartung, Zeit[7], Protokoll-Typ, sowie die Start- und Ziel-IP.

Eine Übersicht über die fünf protokoll-spezifischen Wert-Belegungen sehen wir in Tabelle 4 (Erläuterungen dazu in Kapitel 3.4 und 3.5).

Protokoll-Typ	*Wert 1*	*Wert 2*	*Wert 3*	*Wert 4*	*Wert 5*
TCP	srcprt	dstport	flags	seqnr	acknr
UDP	srcprt	dstport			
TCPUDP	srcprt	dstport	flags	seqnr	acknr
ICMPecho	type	idnr	seqnr		
ICMPtstamp	idnr	seqnr	otime	rtime	ttime
ICMPunreach	code	origid			
ICMPredir	code	gwip	origid		
ICMPtexc	code	origid			

Tabelle 4: Nötige Werte aller Protokoll-Typen

Speziell ist das Pseudo-Protokoll 'TCPUDP': Häufig will man gewisse Tests für TCP *und* UDP durchführen. Damit man solche Testpakete aber nicht für beide Protokolle definieren muss, lässt sich die Protokoll-Verwendung von solchen 'TCPUDP'-Tests durch eine Option beim Programmstart von *fwtest* festlegen. Standardmässig werden sie als TCP-Pakete verwertet; mit der Option `-u` werden sie als UDP-Pakete interpretiert, dabei werden die TCP-spezifischen Werte (*flags*, *seqnr* und *acknr*) einfach ignoriert.

3.4. UDP-Pakete

UDP ist ein sehr einfaches und schmales, aber vielleicht umso nützlicheres Protokoll. Deshalb gibt es auch nur wenig anzugeben: nämlich nur die Portnummer beim Ursprungs- (*srcprt*) und beim Ziel-Rechner (*dstprt*). Eine Darstellung eines UDP-Pakets findet sich in Kapitel 2.1.

Da wir bei unseren Testpaketen generell keine Daten mitsenden, bzw. dem Benutzer keine Möglichkeit dazu geben, ist die Gesamtlänge eines UDP-Datagramms immer 8 Bytes.

Schliesslich bleibt nur noch ein Wert: die Prüfsumme, welche aber automatisch berechnet wird.

[7]Die Zeit kann in *fwtest* zukünftig nur noch relativ angegeben werden, siehe [Sch06].

3.5. ICMP-Pakete

Eine vollständige Implementierung von ICMP macht weder Sinn, noch wäre sie jemals fertig zu stellen, da laufend neue Typ-Spezifikationen dazukommen.

Uns ist es einerseits wichtig, überhaupt irgendeinen ICMP-Typen implementiert zu haben, damit wir die Firewall auf ICMP generell testen können. Andererseits möchten wir natürlich auch einige spezifische Anwendungen von ICMP testen. Dafür haben wir uns ein paar gängige Typen ausgesucht: *Echo Request/Reply*, *Destination unreachable* und *Time exceeded*.

Des Weiteren haben wir auch noch *Redirection* und *Timestamp (reply)* ausgewählt, da diese von der *Libnet*-Bibliothek auch direkt unterstützt werden und somit keinen grossen zusätzlichen Aufwand bei der Implementierung darstellen.

Interessant wären auch noch *Source Quench* (eine Art Flusssteuerung), *Parameter Problem* (Fehlermeldung bei ungültigen Header-Daten) und *Information request/reply* (Herausfinden der Netzwerk-Nummer). Da diese ICMP-Typen von *Libnet* aber (noch) nicht unterstützt werden, belassen wir es vorerst bei den erstgenannten fünf Typen. Diese sind in Kapitel 2.2 näher erläutert.

ICMP Echo / Echo Reply Mit dem Wert *type* wählt man aus, ob eine Echo-Anfrage (8) oder eine Echo-Antwort (0) verschickt werden soll. Der ICMP-Code bleibt immer 0.

idnr und *seqnr* dienen zur eindeutigen Identifikation. Es können Nummern bis max. 65535 (16bit) angegeben werden.

Theoretisch wäre es möglich mit diesem ICMP-Typen Daten mitzusenden. Darauf verzichten wir aber vorerst, wie auch schon bei TCP und UDP. Einer späteren Anpassung steht nichts im Wege.

ICMP Timestamp / Timestamp reply Hier dienen erneut *idnr* und *seqnr* zur besseren Identifikation (16bit-Nummern). *otime*, *rtime* und *ttime* sind 32bit-Zeitstempel (Anzahl Millisekunden seit Mitternacht UTC). Der ICMP-Code ist auf 0 festgelegt.

Eine Timestamp-Anfrage wird erstellt, wenn bei *rtime* und *ttime* 0 angegeben wird, sonst wird eine Timestamp-Antwort erstellt.

Da ICMP-Fehlermeldungen (siehe folgende Typen) immer auf ein empfangenes Paket reagieren, müssen sie zur Identifikation beim Ursprungsrechner den vollständigen IP-Header plus mindestens 64bit IP-Daten des Original-Paketes enthalten. Somit kann dieser seine Anfrage korrigieren und erneut senden.

Damit wir aber nicht pro ICMP-Fehlermeldung das ganze Original-Paket spezifizieren müssen, referenzieren wir einfach vorhandene Pakete in unseren Testfällen. Das ist auch sinnvoll, da wir in unseren Tests doch möglichst reale Bedingungen schaffen möchten: Pakete hängen voneinander ab und bilden (zumindest bei TCP) zustandsbehaftete Verbindungen.

Somit muss man bei allen Protokoll-Typen, die zur Klasse der Fehlermeldungen gehören, die ID des Original-Paketes angeben (*origid*)[8]. Gemäss [Bra89] ist es nicht erlaubt, auf andere ICMP-Fehlermeldungen zu verweisen. Das Original-Paket darf jedoch in einem anderen Testfall liegen und kann (unsinnigerweise) zeitlich auch erst später[9] erscheinen.

ICMP Destination unreachable Die Art dieser Meldung gibt man mit *code* an. Dieser Wert zwischen 0 und 15 entspricht genau dem ICMP code, dh. eine 1 steht zum Beispiel für einen 'Host unreachable error' (siehe Tabelle 1).

ICMP Redirect Dieser Paket-Typ benötigt den Wert *code* (zwischen 0 und 3, siehe Tabelle 2), sowie die IP des zu kontaktierenden Gateways (*gwip*).

ICMP Time Exceeded Hier sollte der *code* entweder 0 (TTL exceeded) oder 1 (fragment reassembly time exceeded) sein.

3.6. Paket-ID-Übertragung

Damit wir wissen, welche Pakete die Firewall durchlässt und welche nicht, ist es wichtig zu wissen, welches empfangene Paket welcher Paket-Spezifikation in der TP-Datei entspricht.

In der Version 0.5 wurden einfach alle gesandten Werte wie IP, Port, Sequenznummer, etc. auf Gleichheit überprüft. Da ein Paket unterwegs aber verändert werden kann, suchen wir jetzt eine bessere Lösung. Ein Beispiel zu solch einer Änderung stellt NAT[10] dar, das IPs und Ports maskiert.

Deshalb möchten wir nun eine Paket-Identifikations-Nummer im Paket mitsenden und hoffen, dass die Firewall oder sonst ein Router diese ID nicht verändert. Dazu gibt es einige Möglichkeiten:

Payload Die einfachste Möglichkeit ist wohl einfach die ID (bestehend aus Testfall- und Paket-ID) im Datenblock (sog. *payload*) mitzusenden. Damit könnte man auch fast beliebig grosse Nummern gebrauchen. Vor allem wäre dies auch sehr einfach zu implementieren.

Das grosse Problem bei dieser Lösung ist, dass nicht alle Protokolle reine Datenblöcke enthalten können. Natürlich funktioniert es bei den grossen Hauptprotokollen TCP und UDP – sogar auch bei ICMP Echo. Aber bei ICMP haben wir generell keine Möglichkeit eigene Daten mitzusenden, zumindest dann nicht, wenn wir RFC-konform bleiben wollen.

[8] Im endgültigen TP-Datei-Format sieht die *origid* zum Beispiel so aus: `21.8` (für das Paket 8 im Testfall 21).

[9] Wir nehmen immer die Daten aus der Testpaket-Datei, da wir nicht auf ankommende Pakete zählen können. Weil somit die Daten immer zur Verfügung stehen, ist es uns gleichgültig, zu welcher Zeit das Original-Paket verschickt wird.

[10] Network Address Translation, zum Beispiel zur Einbindung privater Netzwerke ins öffentliche Netz. NAT soll in *fwtest*-1.0 neu unterstützt werden, siehe [Sch06].

Bei ICMP-Paketen würde man deshalb die ID-Übertragung unterlassen und die Identifizierung auf die traditionelle Art durchführen (Vergleichung aller Header-Felder). Dies wäre aber nicht sehr transparent:

Mit eingeschaltetem NAT würden ICMP-Nachrichten in der Statistik als 'false positive' erscheinen, weil die IPs nicht mehr übereinstimmen und somit das Paket (durch Header-Vergleich) nicht identifiziert werden kann. Der Benutzer könnte das falsch interpretieren, nämlich dass die Firewall ICMP-Pakete blockt.

IP-Identifikation, aufgeteilt Eine andere Lösung wäre die Benutzung des IP-Identifikation-Feldes im IP-Header. Da alle Protokolle auf IP basieren, funktioniert diese Lösung für alle Protokolle.

Dieses Feld ist 16bit lang. Wenn man es aufteilt in 9bit für die Testfälle und 7bit für deren Pakete, könnten wir maximal 512 Testfälle und 128 Pakete pro Testfall zulassen. Damit setzen wir jedoch eine relativ niedrige Limite. Beim Testen von Firewalls kann es schnell sehr viele Pakete geben.

IP-Identifikation, Mapping Da wohl meistens nicht alle 128 Pakete pro Testfall spezifiziert werden, sind einige Paket-Nummern durch die fixe Aufteilung nicht benutzbar. Deshalb wäre eine dritte Möglichkeit im IP-ID-Feld ein Mapping zu gebrauchen, so dass wir fortlaufend alle Nummern von 0 bis 65535 vergeben können. Diese Anzahl wird in den meisten Fällen genügen.

Mix Theoretisch könnte man nun die beiden Haupt-Ansätze mischen und je nach Protokoll eine andere Art der Übertragung definieren. Auch diese Lösung ist für den Benutzer nicht transparent:

Wenn eine Firewall zum Beispiel nur die Daten blockt, dann würde *fwtest* nur ICMP-Pakete erkennen, da bei ihnen die ID im IP-Header steht. Sämtliche TCP- und UDP-Pakete hingegen würden als 'false positive' markiert, weil die Datenblöcke mit der ID nicht ankommen. Wiederum könnte der Benutzer auf eine Blockierung von bestimmten Protokollen (TCP/UDP) schliessen anstatt auf die tatsächliche Blockierung von Daten.

Natürlich möchten wir dem Benutzer die volle Kontrolle geben, aber standardmässig eine saubere Lösung ausführen. Deshalb haben Adrian Schüpbach[11] und ich uns für folgende Idee entschieden:

Normalerweise wird die ID gemappt im IP-Identifikations-Feld übertragen. Mit der *fwtest*-Option `-p` kann die Übertragung im Datenblock zugeschaltet werden (somit sind wir nicht mehr auf 16bit eingeschränkt). Dass dabei ICMP benachteiligt wird, sollte in der Benutzer-Dokumentation (README, Usage) erwähnt werden.

In der Implementierung in Kapitel 4.2.2 wird genauer auf dieses Mapping eingegangen. Die Idee ist simpel: Beim Parsen der Spezifikationen werden die Pakete fortlaufend durchnummeriert.

[11] Autor der Semesterarbeit über die *fwtest*-Umstrukturierung für NAT, siehe [Sch06].

4. Implementierung

Dieses Kapitel zeigt auf, wie die geplante Arbeit in *fwtest* ausgeführt wurde.

Kapitel 4.1 beschreibt, wie *fwtest* angepasst wurde, so dass mehrere Protokolle pro Programmlauf gebraucht werden können.

Kapitel 4.2 befasst sich mit dem Parser für die neuen Paket-Spezifikationen und der aufgebauten Datenstruktur.

Die Kapitel 4.3 und 4.4 zeigen auf, wie UDP und ICMP in *fwtest* implementiert wurden. Dies umfasst zwei Kernaspekte:

- Pakete generieren
- Pakete abfangen

4.1. Mehrere Protokolle gleichzeitig

Die im Kapitel 3.2 erwähnten Design-Änderungen führten zu einer Erweiterung der Ereignis-Struktur in *fwtest*: Anstatt ein globales Feld für den Protokoll-Typen zu verwenden, spezifizieren wir diesen nun für jedes Ereignis. Das führte zu vielen kleinen Änderungen im Code; diese konnten aber ohne Probleme ausgeführt werden, weil überall, wo die Protokoll-Information benötigt wurde, auch die Ereignis-Daten oder die Paket-Informationen verfügbar waren.

Eine Ausnahme bildete das Abfangen von ankommenden Paketen. Hier wissen wir eben nicht, was wir für ein Paket erhalten haben und müssen es erst analysieren. Dafür müssen wir aber genügend viele Daten des Paketes abfangen. Wir setzten daher die Länge *snaplen* auf ein Maximum aller möglichen, erwarteten Paket-Längen[12]. In unserer Konstellation brauchen die ICMP-Fehler-Nachrichten am meisten Platz: 36 Bytes (plus Ethernet- und IP-Kopfdaten). Für eine genauere Analyse siehe Kapitel 2.2.

Dieser Wert wird in `lpcap_init(char*)` gebraucht beim Starten der *pcap*-Bibliothek. Falls in Zukunft noch weitere Protokolle mit noch längeren Kopfdaten implementiert werden, kann in `lpcap.h` einfach die Konstante `MAX_THDR_LEN` verändert werden[13].

4.2. Parsen der Paket-Spezifikationen

Da das neue Format für die Spezifizierung der Test-Pakete einiges komplexer geworden ist (siehe Appendix A), wurde der Parser nicht mehr von Hand geschrieben, sondern anhand einer Grammatik zur Kompilierungszeit mit *Bison*[14] generiert.

[12] *snaplen* spezifiziert die maximal zu erfassende Anzahl Bytes.

[13] Ethernet- und IP-Kopfdaten (34 Bytes) werden in `lpcap_init(char*)` noch dazu addiert.

[14] *GNU Bison* ist ein Mehrzweck-Parser-Generator, der eine Grammatik-Beschreibung (im *yacc*-Format) für eine LALR kontext-freie Grammatik in ein C-Programm verwandelt, welches diese Grammatik parsen kann [CS05].

Die Grammatik (`tpparser.y`) basiert auf Symbolen, die ein Lexer zur Verfügung stellt. Dieser ist ein Teil der Arbeit von Adrian Schüpbach [Sch06].

4.2.1. Datenstruktur

Beim Parsen wird die benötigte Datenstruktur (siehe `tpparser.h`) für die Testpakete direkt aufgebaut: Die Pakete der verschiedenen Testfälle werden als Sende-/Empfangs-Ereignis im richtigen Zeitfenster eingefügt.

Die Ereignis-Datenstruktur besteht aus einer doppelt verketteten Liste mit allen Zeitfenstern (`tnode`). Pro Zeitfenster gibt es eine einfach verkettete Liste mit allen Testfällen (Ereignisse `event`), die in diesem Zeitfenster gesandt und empfangen werden sollen. Jedes Ereignis zeigt auf eine Sende- und eine Empfangs-Paketinformations-Struktur (`packetinfo`). Diese enthalten die IP-Adressen und die protokollspezifischen Werte (siehe auch Tabelle 4 auf Seite 16). Die Abbildung 10 zeigt die Struktur genauer auf.

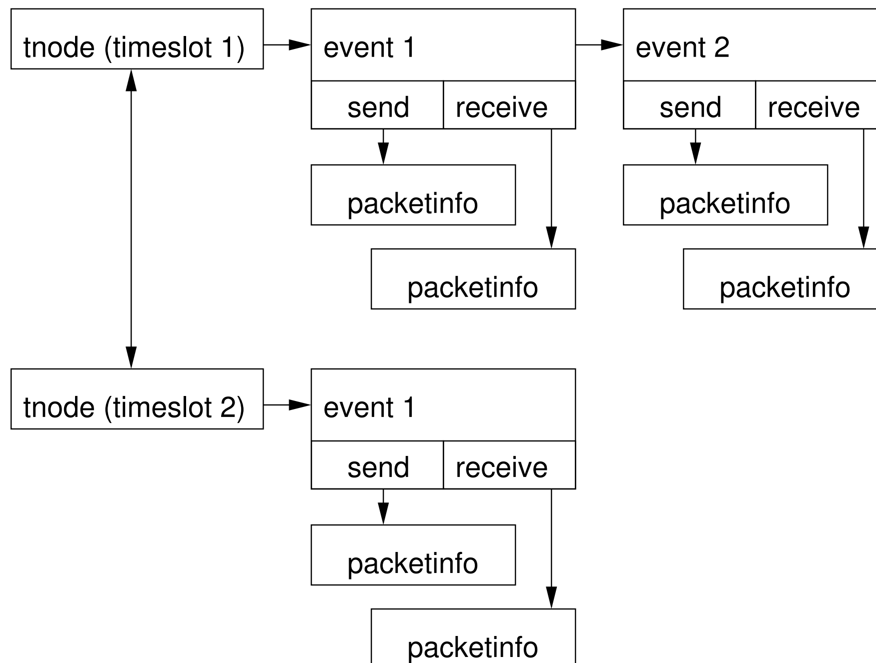


Abbildung 10: Aufgebaute Ereignis-Datenstruktur

4.2.2. Eigenschaften

Wie in Kapitel 3.6 beschrieben, verwenden wir zur Übermittlung der Paket-ID nicht die Testfall- und Paket-Nummer, sondern eine fortlaufende Zahl, um die zur Verfügung stehenden 16bit voll ausnutzen zu können. Die Pakete werden also gleich beim Parsen durchnummeriert; die Zahl wird beim entsprechenden Ereignis im Feld `pid` gespeichert. Falls die Option `-p` nicht aktiviert ist, wird die maximale Anzahl Pakete

auf $2^{16} - 1$ (65'535) eingeschränkt, siehe dazu auch den Design-Entscheid im eben erwähnten Kapitel.

Bei der Angabe von Ports kann man Nummern oder Namen verwenden, so wie sie in `/etc/services` beschrieben sind, Domain-Namen werden hingegen nicht in IP-Adressen aufgelöst.

Sämtliche Eingabe-Werte für die protokollspezifischen Optionen werden auf Korrektheit überprüft (ungültige Werte für Typen/Codes erzeugen einen fatalen Parser-Fehler). Zudem werden wie bis anhin Pakete ausgefiltert, welche nicht in das festgelegte Netzwerk des Sende-/ oder Empfangs-Rechners gehören (siehe Methode `involved`).

Um das Parsen zu initiieren, muss nur eine Funktion (`parse_file(char *)`) ausgeführt werden. Sie erhält als einziges Argument den Pfad der TP-Datei und stellt die erzeugte Datenstruktur bei Erfolg unter `pv.timeslots` global zur Verfügung.

4.2.3. Präprozessor M4

Ein Präprozessor kann den Benutzer beim Schreiben der Paket-Spezifikationen unterstützen, indem er die Datei vor dem Parser verarbeitet und dabei gewisse Textteile ersetzt. Anstatt zum Beispiel eine häufig gebrauchte IP in vielen Paketen anzugeben, verwendet man besser eine Konstante 'SRCIP' und definiert diese am Anfang der Datei mit folgendem Makro: `define(SRCIP,192.168.1.10)`. Der Präprozessor ersetzt dann die Zeichenfolge 'SRCIP' in der ganzen Datei mit der angegebenen IP.

Wir haben uns für *GNU m4* [Sei05] als Präprozessor entschieden, weil dieser heutzutage wohl auf jeder Linux-Distribution standardmässig installiert ist.

In *fwtest* stellen wir eine Standard-Makro-Datei zur Verfügung (siehe Appendix B). Sie enthält Makros für die verschiedenen ICMP-Typen und -Codes, damit der Benutzer nicht mit den Nummern hantieren muss.

Diese Datei kann man in der Umgebungsvariable `FWTEST_MACROS` angeben; dann liest *fwtest* diese immer vor den Paket-Spezifikationen ein. Generell lassen sich Dateien immer via dem *m4*-Makro `include` einbinden – man kann sich vorstellen so seine Tests in verschiedene Teile aufzusplitten, um eine bessere Übersicht zu haben.

Wer den Präprozessor ganz abschalten will, setzt die Umgebungsvariable `FWTEST_USEM4` einfach auf `no`.

4.3. UDP

Bei der Implementierung von UDP in *fwtest* gab es keine Probleme. Wir können UDP als einen kleinen Teil von TCP anschauen, weil es ja nur einen Teil der Kopfdaten von TCP braucht (nur die Port-Nummern, die Prüfsumme und die Datenpaket-Länge[15]). Natürlich ist UDP nicht so zuverlässig wie TCP und bietet auch keine Flusssteuerung, aber wir sind ja nur am Versenden und Empfangen von einzelnen Paketen interessiert.

[15]Eigentlich ist die Paket-Länge nicht in den TCP-Kopfdaten enthalten, aber um ein TCP-Paket mit `libnet_build_tcp` zu bauen, muss man sie trotzdem angeben.

4.3.1. Pakete generieren

Um ein UDP-Paket zu generieren, kopierten wir einfach die entsprechende Funktion für TCP (`lnet_build_tcp` in `lnet.c`) und passten sie für UDP an.

Das heisst wir änderten 'tcp' im Aufruf `libnet_build_tcp` auf 'udp' und löschten TCP-spezifische, unbenötigte Argumente. Wie auch bei TCP stellt *Libnet* für Felder mit mehreren Bytes die Netzwerk-Byte-Reihenfolge sicher und rechnet die Prüfsumme automatisch aus.

Des Weiteren trugen wir die neue Funktion in `lnet_build_pkts` ein, damit UDP-Pakete auch wirklich als solche behandelt werden.

Normalerweise werden die untersten 16bit der Paket-ID als IP-Identifikation mitgegeben. Wenn immer die Option `-p` beim Starten von *fwtest* angegeben wird, sendet *fwtest* die volle Paket-ID zusätzlich im Datenblock mit.

4.3.2. Pakete abfangen

`lpcap_capture`[16] passten wir an den in *fwtest*-v0.5 vorbereiteten Stellen so an, dass ein UDP-Paket nicht mehr abgewiesen oder ignoriert wird. Somit werden die empfangenen Daten in unsere Datenstruktur hinein kopiert.

Die Hauptsache ist die Analyse des eingelesenen UDP-Paketes. Dazu erstellten wir eine neue Funktion `cmp_udp_pkts`, welche ein empfangenes Paket mit einem Ereignis aus unserer Datenstruktur auf Gleichheit überprüft (auch fast dasselbe wie bei TCP). Diese muss von `analyze` und `mark_event` aufgerufen werden.

4.4. ICMP

Auch der Einbau von ICMP-Echo und ICMP-Timestamp ging ohne Probleme vonstatten. Bei der Implementierung der ICMP-Fehlermeldungen mussten wir speziell auf die Original-Pakete achten.

4.4.1. Pakete generieren

Für die Generierung aller ICMP-Pakete verwenden wir folgende *Libnet*-Funktionen:

- `libnet_build_icmpv4_echo`
- `libnet_build_icmpv4_unreach`[17]
- `libnet_build_icmpv4_redirect`
- `libnet_build_icmpv4_timeexceed`
- `libnet_build_icmpv4_timestamp`

[16] Siehe eine kurze Beschreibung dazu in Kapitel 3.1.
[17] Leider wird das MTU-Feld von *Libnet* nicht unterstützt.

Dazu prüfen wir vor dem Aufruf immer noch auf gültige ICMP-Typen und -Codes.

Die Paket-ID wird immer nur in den IP-Kopfdaten mitgesandt. Der Gebrauch eines Datenblocks bei ICMP wäre generell nicht RFC-konform, ausser bei ICMP-Echo. Wir verzichten aber auch da auf den Datenblock, da eine Echo-Antwort die gleichen Daten wie in der Echo-Anfrage enthalten muss und der Datenblock somit nicht völlig frei wählbar ist.

Original-Pakete Da wir bei ICMP-Fehlermeldungen zumindest Teile des Original-Pakets zurück senden müssen (siehe Kapitel 3.5), brauchen wir die Spezifikation davon. Der Parser gibt uns aber nicht direkt eine Speicher-Referenz auf das Paket, sondern nur den Testfall und die Paket-Nummer[18]. Deshalb suchen wir mit `find_event` nach dem Paket am entsprechenden Ort in der Ereignis-Datenstruktur.

Die Methode `lnet_build_origpkt` erledigt die Generierung des Original-Pakets; sie ermittelt das Paket und ruft je nach Protokoll die entsprechende Funktion auf, die es generieren kann. Dabei wird darauf geachtet, dass ICMP-Fehlermeldungen nicht als Original-Pakete dienen dürfen.

Somit stellen wir nicht nur wie verlangt die IP-Kopfdaten plus 64bit Daten bereit, sondern das gesamte Original-Paket, was gemäss [Bra89] durchaus erlaubt ist (solange die Original-Daten nicht verändert werden). Da wir die Daten nicht von einem angekommenen Paket nehmen, sondern aus unseren Paket-Spezifikationen, muss das ganze Paket darin vollständig beschrieben werden. Es ist also nicht mehr erlaubt, beim Generieren für gewisse Felder Zufallswerte einzubauen (wie es früher zum Beispiel bei der TCP-Fenstergrösse und auch der IP-Identifikation der Fall war). Daher haben wir für die TCP-Fenstergrösse einfach einen festen Wert 16'384 (2^{14}) gewählt.

4.4.2. Pakete abfangen

Beim Abfangen von Paketen werden sämtliche ICMP-Nachrichten als *ein* Protokoll angesehen; schliesslich steht in den IP-Kopfdaten im Protokoll-Feld nur ICMP und nicht der genaue ICMP-Typ. Wir kopieren einfach das gesamte Paket von *pcap*, schauen den ICMP-Typen dann nach und überprüfen die Länge des eingegangenen Pakets.

Wie auch bei UDP mussten neue Funktionen für die Paket-Analyse erstellt werden – pro ICMP-Typen eine. Somit ist *fwtest* nun in der Lage, ICMP-Pakete mit den Spezifikationen zu vergleichen.

[18] Da der Parser nur einmal über die Eingabe läuft und man auf Pakete verweisen kann, die erst weiter unten in der Eingabe spezifiziert werden, kann er selber keine Verweise in Speicher-Adressen auflösen.

5. Ergebnisse

Um unsere Implementierung zu prüfen, haben wir einige Tests durchgeführt. Dabei schauten wir in erster Linie nicht auf die Funktionsweise der Firewall, sondern auf die unseres Programmes *fwtest*.

5.1. Testumgebung

Für diese Aufgabe haben wir uns die Testumgebung in Abbildung 11 zurecht gelegt:

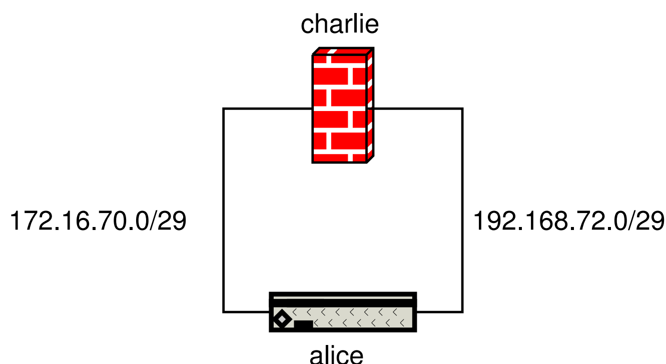


Abbildung 11: Testumgebung

Unser Testrechner *alice* hat zwei Netzwerkschnittstellen, die beide mit einer Firewall *charlie* verbunden sind. Diese Netzwerke stellen damit beide Seiten der Firewall dar, nämlich 172.16.70.0/29 und 192.168.72.0/29. Die Firewall leitet Pakete unter diesen Netzwerken weiter oder blockt sie ab – je nach konfigurierten Regeln.

Für die beiden Maschinen verwenden wir einfachheitshalber VMWare [vmw04], so dass wir auf richtige Hardware verzichten können. Auf den virtuellen Maschinen läuft ein älteres Linux (Redhat 7.3); für die Firewall verwenden wir *iptables*.

Zur Kontrolle des Netzwerkverkehrs lassen wir *tcpdump* auf *alice* mithören. Damit erhalten wir einen genauen Report, welche Pakete auf den jeweiligen Netzen versandt werden.

5.2. Zwei Testfälle

Zur Illustration der Funktionalität von *fwtest* haben wir zwei Testfälle ausgewählt. Beim Testen betrachten wir die Ein- und Ausgaben von *fwtest*, sowie die Logs von *tcpdump*.

Abbildung 12 zeigt, wie unsere Testfälle gemäss unserem neuen Format (siehe Appendix A) spezifiziert werden.

Es handelt sich dabei um zwei Testfälle: Beim Fall 1 wird zuerst ein UDP-Paket von *ip1* Port 12300 an den Port 25 (smtp) von *ip2* gesandt. Dies quittiert der Zielrechner *ip2* mit einer ICMP-Meldung vom Typ *Destination (Port) unreachable*. Darauf

```
define(ip1, 172.16.70.4)
define(ip2, 192.168.72.6)
testcase 1 {
  packet 1 { UDP send { ip1 ip2 12300 smtp }
             receive { ip1 ip2 12300 smtp } }
  packet 2 { ICMPunreach send { ip2 ip1 PORT_UNREACHABLE 2.1 }
                     receive { ip2 ip1 PORT_UNREACHABLE 2.1 } }
  packet 3 { UDP send { ip1 ip2 12300 24 } receive {} }
}
testcase 2 {
  packet 1 { ICMPecho send { ip2 ip1 ECHO_REQUEST 23 1 }
                  receive { ip2 ip1 ECHO_REQUEST 23 1 } }
  packet 2 { ICMPecho send { ip1 ip2 ECHO_REPLY 23 1 }
                  receive { ip1 ip2 ECHO_REPLY 23 1 } }
}
```

Abbildung 12: `test.tp`: Spezifikation unserer Testfälle im neuen Format

versucht es der Sender *ip1* erneut, diesmal beim Port 24. Von diesem Paket erwarten wir, dass es bereits von der Firewall geblockt wird.

Beim zweiten Fall wird eine ICMP-Echo-Anfrage vom Rechner *ip2* zu *ip1* versandt. Darauf sollte eine Echo-Antwort zurückkommen.

```
#!/bin/bash
ip1=172.16.70.4
ip2=192.168.72.6

# initialize firewall: drop all packets
iptables -F
iptables -X
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -P FORWARD DROP

# udp: ip1:* -> ip2:25
iptables -A FORWARD -p udp -s $ip1 -d $ip2 --dport smtp -j ACCEPT

# icmp: ip2 -> ip1
iptables -A FORWARD -p icmp -s $ip2 -d $ip1 -j ACCEPT
```

Abbildung 13: Firewall-Regeln

Wir konfigurieren nun die Firewall, so dass sie sämtliche UDP-Pakete von *ip1* zu *ip2* an den Port 25 zulässt. Damit sollte der erste Testfall erwartungsgemäss ablaufen. Für den zweiten Testfall lassen wir ICMP-Pakete aber nur in die eine Richtung zu, nämlich von *ip2* nach *ip1*. Damit wird das Paket `2.2` fälschlicherweise geblockt und

wir können feststellen, wie *fwtest* mit dieser Situation umgeht und wie es dieses Ereignis aufzeichnet.

Die Regeln der Firewall aktivieren wir mit dem Shell-Skript in Abbildung 13.

Soweit haben wir für den Test alles vorbereitet. Wir starten nun *fwtest* mit dem folgenden Befehl:

```
./run_fwtest.sh test.tp test.log 172.16.70.0/29 eth0 192.168.72.0/29 eth1
```

Nach wenigen Sekunden ist der Test vollendet und *fwtest* zeigt eine Statistik an, wie in Abbildung 14 zu sehen ist. Es sind fünf Pakete versandt worden; drei davon sind richtigerweise angekommen (*true negatives*) und eines wurde erfolgreich (*true positive*), ein anderes aber fälschlicherweise geblockt (*false positive*).

```
Packets captured:    5
True positives:      1
True negatives:      3
False positives:     1
False negatives:     0
False changed:       0
```

Abbildung 14: Ausgabe der Statistik von *fwtest*-1.0

Wenn wir das Logfile (Abbildung 15) betrachten, sehen wir das Paket `2.2` aufgelistet. Es handelt sich genau um diese ICMP-Echo-Antwort, welches die Firewall gemäss unserer Testfall-Spezifikation hätte weiterleiten müssen. Indem wir die Firewall aber absichtlich anders konfigurierten, haben wir aufgezeigt, dass *fwtest* wirklich in der Lage ist, eine solche “Regelwidrigkeit” aufzuspüren.

```
# Fwtest v1.0 [Firewall Testing Tool]
# 02/15/2006 13:34:12.611088
# Timeout between time slots: 1 second(s)
# [time]          [type]      [id]     [packet]
#
13:34:14.641088 false_pos   2.2      --
```

Abbildung 15: Logdatei von *fwtest*-1.0

Nun sehen wir uns noch an, welche Pakete im Netzwerk tatsächlich versandt wurden. Die Abbildungen 16 und 17 zeigen deutlich, welche Pakete die Firewall geblockt und welche sie weitergeleitet hat.

Dieser simple Test hat veranschaulicht, dass *fwtest* auch mit unserer Erweiterung der Protokolle zuverlässig funktioniert.

Ausserdem wurden zusätzliche Tests durchgeführt. Diese prüften einerseits das korrekte Generieren und Versenden aller verschiedenen Paket-Typen. Dazu wurde *ethereal* [Com05] hinzugezogen, das dank einer grafischen Oberfläche über abgefangene Netzwerk-Pakete eine übersichtlichere Anzeige als *tcpdump* bietet.

Andererseits wurde geprüft, ob *fwtest* eine Meldung macht, wenn auch nur kleinste Details in einem Paket falsch übertragen werden. Besonderen Wert haben wir eben-

```
13:34:12.646795 arp who-has ip1 tell 172.16.70.1
13:34:12.647471 ip1.12300 > ip2.smtp: [udp sum ok] udp 0 (ttl 64, id 0, len 28)
13:34:12.648248 arp reply ip1 is-at 0:0:ac:10:46:4
13:34:12.648407 ip2 > ip1: icmp: echo request (ttl 63, id 3, len 28)
13:34:13.641300 ip1 > ip2: icmp: echo reply (ttl 64, id 4, len 28)
13:34:13.642261 ip2 > ip1: icmp: ip2 udp port smtp unreachable (ttl 63, id 1, len 56)
13:34:14.641547 ip1.12300 > ip2.24: [udp sum ok] udp 0 (ttl 64, id 2, len 28)
```

Abbildung 16: *TCPdump*-Ausgabe aus dem Netzwerk 172.16.70.0/29

```
13:34:12.640330 ip2 > ip1: icmp: echo request (ttl 64, id 3, len 28)
13:34:12.647990 arp who-has ip2 tell charlie
13:34:12.649703 arp reply ip2 is-at 0:0:c0:a8:48:6
13:34:12.649964 ip1.12300 > ip2.smtp: [udp sum ok] udp 0 (ttl 63, id 0, len 28)
13:34:13.641453 ip2 > ip1: icmp: ip2 udp port smtp unreachable (ttl 64, id 1, len 56)
```

Abbildung 17: *TCPdump*-Ausgabe aus dem Netzwerk 192.168.72.0/29

falls darauf gelegt, dass die Paket-ID in den IP-Kopfdaten sowie auch im eigentlichen Datenblock versendet werden kann.

Alle diese Tests hat die neue Version von *fwtest* erfolgreich gemeistert.

6. Abschluss

6.1. Zusammenfassung

Die Umsetzung einer Sicherheits-Policy durch eine Firewall sollte möglichst umfassend geprüft werden. Mit der *fwtest*-Erweiterung um UDP und ICMP haben wir einen wichtigen Schritt in diese Richtung getan: Testfälle können jetzt TCP-, UDP- und auch ICMP-Pakete abdecken.

Mit der Hilfe von Adrian Schüpbach haben wir ein neues Format zur Spezifizierung von Testpaketen entworfen, das auch für weitere Protokolle leicht anzupassen ist (Appendix A). Dazu haben wir einen neuen, leistungsfähigen Parser entwickelt (Kapitel 4.2).

Des Weiteren haben wir *fwtest* so abgeändert, dass auch mit mehreren Protokollen gleichzeitig gearbeitet werden kann (Kapitel 4.1).

Bei ICMP haben wir gesehen, dass eine vollständige Implementierung schwer möglich ist, da das Protokoll viele verschiedene Untertypen hat und immer weitere dazu kommen (Kapitel 3.5). Wir haben deshalb nur die folgenden wohl wichtigsten Typen implementiert: *Echo Request/Reply*, *Destination unreachable*, *Redirect*, *Time Exceeded* und *Timestamp Request/Reply*.

Zum Schluss haben wir die Implementierung anhand selbst erstellter Testfälle geprüft. Dies hat uns nicht die Funktionsweise der benutzten Firewall, sondern eben die von *fwtest* selber aufzeigt. Dabei hat die neue Version von *fwtest* (v1.0) alle Tests erfolgreich bestanden.

6.2. Fazit

Im Rückblick auf diese Semesterarbeit erkenne ich, dass die Phasen der Implementierung jeweils sehr schnell abgeschlossen sind, Design-Prozesse aber länger dauern können und auch von grosser Wichtigkeit sind.

Eine gute, zeitliche Koordination zwischen den verschiedenen Mitarbeitern ist nötig, damit sich ein Projekt nicht in die Länge zieht.

Obwohl die Implementierung von UDP und ICMP nun ganz anders aussieht, als ursprünglich in *fwtest*-0.5 vorgesehen, konnten wir doch einen grossen Nutzen aus den damaligen Design-Entscheidungen ziehen. Es stellte sich heraus, dass sich die *Libnet*-Bibliothek für alle benötigten Protokolle eignet.

Generell gab ICMP bei der Implementierung wegen der verschiedenen Untertypen und besonderen Eigenheiten einiges an Mehr-Aufwand als UDP.

Persönlich habe ich einiges über Netzwerk-Programmierung gelernt – nicht nur beim “Bit-Fiddling” während des Programmierens in C, sondern auch beim Einarbeiten in den Programmcode von *fwtest*-0.5 und einigen Teilen der verwendeten Bibliotheken.

6.3. Ziele für die Zukunft

Die Entwicklung an *fwtest* ist längst nicht abgeschlossen.

Interessant wird es eigentlich erst, wenn Testpakete richtige Datenblöcke enthalten (momentan werden Pakete nur mit Kopfdaten verschickt, das heisst ohne Datenblöcke). Firewalls könnten durchaus anders reagieren, wenn Daten hinzukommen.

In Bezug auf ICMP lassen sich in *fwtest* weitere Typen implementieren, die wir hier noch ignoriert haben. Es sollte auch nach einer Lösung gesucht werden für das angesprochene Problem betreffend des fehlenden MTU-Feldes (siehe Kapitel 4.4).

Zudem könnte es sinnvoll sein, *fwtest* so anzupassen, dass ICMP-Pakete erwartet werden können, die nicht von *fwtest* selber, sondern von der Firewall (oder einem anderen Rechner im Netz) verschickt worden sind. Eine abblockende Firewall wirft ein Paket ja nicht unbedingt nur fort, sondern sendet unter Umständen auch eine ICMP-Fehlermeldung zurück – dies möchte man vielleicht zusätzlich prüfen.

Ein Update aller benötigten Bibliotheken sollte ausserdem bald in Betracht gezogen werden, um allfälligen Fehlern und Sicherheitslücken entgegen zu wirken.

In ferner Zukunft dürfte wohl durchaus IPv6 eine grosse Rolle spielen; eine Unterstützung davon könnte man deshalb auch allmählich ins Auge fassen.

6.4. Danksagung

An dieser Stelle möchte ich mich herzlich bei allen bedanken, welche zum Gelingen dieser Semesterarbeit beigetragen haben. Insbesondere gilt der Dank meiner Betreuerin Diana Senn für die regelmässigen Treffen und die gesamte Unterstützung während des Projektes.

Ich bedanke mich ebenfalls bei Adrian Schüpbach für die Motivation und die gute Zusammenarbeit in den überlappenden Teilarbeiten. Die Arbeit hat so definitiv mehr Spass gemacht.

A. Format Test-Pakete

```
Testcases       = Testcase { Testcase } ;
Testcase        = 'testcase' Int '{' Packet { Packet } '}' ;
Packet          = 'packet' Int '{' (Tcp | Udp | Tcpudp | Icmpecho
                  | Icmpunreach | Icmpredir | Icmptexc | Icmptstamp) '}' ;
Tcp             = 'TCP' 'send' Tcpsend 'receive' Tcprecv ;
Tcpudp          = 'TCPUDP' 'send' Tcpsend 'receive' Tcprecv ;
Tcpsend         = '{' Ip Ip Port Port Flag Optint Optint '}' ;
Tcprecv         = '{' Ip Ip Port Port Flag Optint Optint '}' | '{' '}' |
                  '?' | 'ok' ;
Udp             = 'UDP' 'send' Udpsend 'receive' Udprecv ;
Udpsend         = '{' Ip Ip Port Port '}' ;
Udprecv         = '{' Ip Ip Port Port '}' | '{' '}' | '?' | 'ok' ;
Icmpecho        = 'ICMPecho' 'send' Icmpechosend 'receive' Icmpechorecv ;
Icmpechosend    = '{' Ip Ip Int Optint Idseq '}' ;
Icmpechorecv    = '{' Ip Ip Int Optint Idseq '}' | '{' '}' | '?' | 'ok' ;
Icmpunreach     = 'ICMPunreach' 'send' Icmpunreachsend 'receive'
                  Icmpunreachrecv ;
Icmpunreachsend = '{' Ip Ip Int Origid '}' ;
Icmpunreachrecv = '{' Ip Ip Int Origid '}' | '{' '}' | '?' | 'ok' ;
Icmpredir       = 'ICMPredir' '{' Icmpredirsend 'receive' Icmpredirrecv ;
Icmpredirsend   = '{' Ip Ip Int Ip Origid '}' ;
Icmpredirrecv   = '{' Ip Ip Int Ip Origid '}' | '{' '}' | '?' | 'ok' ;
Icmptexc        = 'ICMPtexc' 'send' Icmptexcsend 'receive' Icmptexcrecv ;
Icmptexcsend    = '{' Ip Ip Int Origid '}' ;
Icmptexcrecv    = '{' Ip Ip Int Origid '}' | '{' '}' | '?' | 'ok' ;
Icmptstamp      = 'ICMPtstamp' 'send' Icmptstampsend 'receive'
                  Icmptstamprecv ;
Icmptstampsend  = '{' Ip Ip Optint Optint Optint Optint Optint '}' ;
Icmptstamprecv  = '{' Ip Ip Optint Optint Optint Optint Optint '}' | '{'
                  '}' | '?' | 'ok' ;
Ip              = Int '.' Int '.' Int '.' Int | Var ;
Port            = Int | Text | Var ;
Flag            = 'S'|'A'|'F'|'R'|'U'|'P' { 'S'|'A'|'F'|'R'|'U'|'P' } |
                  '-' ;
Optint          = Int | '-' ;
Origid          = Int '.' Int ;
Text            = Letter { Letter } ;
Var             = Capitalletter { Letter } ;
Int             = Digit { Digit } ;
Capitalletter   = 'A' | 'B' | ... | 'Z' ;
Letter          = Capitalletter | 'a' | 'b' | ... | 'z' ;
Digit           = '0' | '1' | ... | '9' ;
```

B. Standard-Makros

```
# icmpecho types
define(‘ECHO_REQUEST’, ‘8’)
define(‘ECHO_REPLY’,   ‘0’)

# icmpunreach codes
define(‘NET_UNREACHABLE’,                     ‘0’)
define(‘HOST_UNREACHABLE’,                    ‘1’)
define(‘PROTOCOL_UNREACHABLE’,                ‘2’)
define(‘PORT_UNREACHABLE’,                    ‘3’)
define(‘FRAGMENTATION_NEEDED’,                ‘4’)
define(‘SOURCE_ROUTE_FAILED’,                 ‘5’)
define(‘DEST_NET_UNKNOWN’,                    ‘6’)
define(‘DEST_HOST_UNKNOWN’,                   ‘7’)
define(‘COMM_WITH_DEST_NET_PROHIBITED’,       ‘9’)
define(‘COMM_WITH_DEST_HOST_PROHIBITED’,     ‘10’)
define(‘NET_UNREACHABLE_TOS’,                ‘11’)
define(‘HOST_UNREACHABLE_TOS’,               ‘12’)
define(‘COMM_ADMINISTRATIVELY_PROHIBITED’,   ‘13’)
define(‘HOST_PRECEDENCE_VIOLATION’,          ‘14’)
define(‘PRECEDENCE_CUTOFF’,                  ‘15’)

# icmp redir codes
define(‘NET_ERROR’,      ‘0’)
define(‘HOST_ERROR’,     ‘1’)
define(‘TOS_NET_ERROR’,  ‘2’)
define(‘TOS_HOST_ERROR’, ‘3’)

# icmp texc codes
define(‘TTL_EXCEEDED’,                 ‘0’)
define(‘FRAG_REASSEMBLY_TIME_EXCEEDED’, ‘1’)
```

C. README

$Id: README 24662 2006-02-28 11:01:37Z beatstr $

==

```
    FWTEST v1.0 (February 2006)
    Author:       Gerry Zaugg <zauggge@gmail.com>
    Contributors: Adrian Schuepbach <scadrian@student.ethz.ch>
                  Beat Strasser <b8@student.ethz.ch>
    Maintainer:   Diana Senn <diana.senn@inf.ethz.ch>
```

==

0. Content

1. Description
2. Software Requirements
3. How to Perform a Test Run
   3.1 Test Environment
   3.2 Mandatory Files
   3.3 run_fwtest.sh
   3.4 Test Packets File
       3.4.1 General layout
       3.4.2 Protocol specific fields (PROT_FIELDS)
       3.4.3 Preprocessor
       3.4.4 Variables
       3.4.5 Parse errors
   3.5 Running Fwtest
   3.6 Example
4. Timeout
5. Source Files

1. Description

Fwtest v1.0 provides a simple, portable interface to perform firewall testing. It is executed on a so-called testing host that sends test packets through a firewall. Fwtest crafts and injects the test packets and captures them if they pass the firewall. When receiving packets, fwtest performs an exhaustive analysis revealing irregularities (i.e. packets that are accepted by the firewall although they were expected to be dropped or packets that are discarded/changed although they were expected to be passed). The irregularities are logged and serve as source of information to identify failures.

Fwtest v1.0 was a further development of fwtest v0.5 which evolved in the context of a Diploma Thesis by Gerry Zaugg. Besides an extension to UDP and ICMP, the new version has experienced some underlying restructuring in order to fit for NAT. For more information, read the corresponding documentation(s):

[1] Gerry Zaugg. Firewall Testing. January 2005.
    http://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg_05.pdf

[2] Adrian Schuepbach. Testen von Firewalls mit NAT. February 2006.
http://www.infsec.ethz.ch/people/dsenn/SA_AdrianSchuepbach_06.pdf
[3] Beat Strasser. Eine UDP-/ICMP-Erweiterung fuer fwtest. February 2006.
http://www.infsec.ethz.ch/people/dsenn/SA_BeatStrasser_06.pdf

2. Software Requirements

Fwtest v1.0 is known to compile and run under Linux Debian 3.0 with the 2.4.24 kernel, Linux Red Hat 7.3 with the 2.4.18-3 kernel and Gentoo Linux 2005.1 with the 2.6.14 kernel. Probably, it also runs under OpenBSD, FreeBSD and NetBSD.

You will need flex, bison, gcc, make, libc and libc-dev to build fwtest. We make use of m4 and the administration tools iptables or ipchains.

We need the following libraries:
* libpcap-0.7.2
* libnet-1.1.2.1
* libdnet-1.8
* libc

Libpcap, libnet and libdnet have to be installed from source since we use the header files (pcap.h, libnet.h, dnet.h).

NOTE: Run /sbin/ldconfig before using libdnet. ldconfig creates the necessary links and cache (for use by the run-time linker, ld.so) to the most recent shared libraries (for more information: man ldconfig).

3. How To Perform a Test Run

3.1 Test Environment

This section gives an overview on how a firewall test is performed with fwtest.

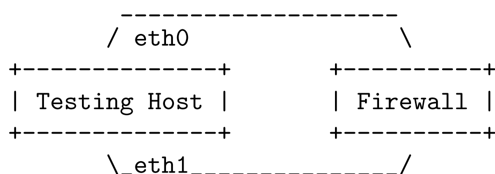


Figure 1: Sample test environment

We assume that you have set up a test environment similar to the one illustrated in figure 1: A testing host is connected to a firewall via two network devices. Fwtest will run on the testing host and craft, inject, capture and analyze packets as well as log irregularities.
You have to provide a test packet file holding the specifications of the test cases (consisting of several test packets) that fwtest will generate, inject and

```
capture. The layout of the this file is explained in section 3.4.


3.2 Mandatory Files

We now discuss how to run fwtest on a testing host.

Copy the following files into the test directory:

(1) Source files: Listed in the last chapter of this file.
(2) run_fwtest.sh: Fundamental shell script performing the test run.


3.3 run_fwtest.sh

run_fwtest.sh leads you through the process of firewall testing.
You have to be root when executing the script. Furthermore, you must provide
some arguments when starting the script. The test run will then be performed
automatically.

run_fwtest.sh requires at least six arguments:
(1) Test packets file
    File containing the test packets
    Read more about the file format below.
(2) Log file
    File where the irregularities are reported
(3) Network 1 and mask in CIDR notation
    Specifies the first network that the testing host represents.
    (e.g. 192.168.1.0/29)
(4) Network 1 interface
    The network interface to capture the packets from network 1 (e.g. eth0)
(5) Network 2 and mask in CIDR notation
    Specifies the second network that the testing host represents.
    (e.g. 172.16.70.0/29)
(6) Network 2 interface
    The network interface to capture the packets from network 2 (e.g. eth1)
(7) Option -p
    Transmit packet id in payload (TCP/UDP only*)
(8) Option -u
    Interpret TCPUDP packets as UDP instead of TCP

* ICMP packets usually don't allow to send a payload. The option -p covers only
TCP and UDP packets; the ICMP packet id is always sent in the IP header. Note
that the IP identification field is only 16 bit - if you deal with more than
2^16 packets including ICMP packets and together with the payload option, you
will certainly run into problems: statistics may be misinterpretable, so be
warned.

Before fwtest can be called, you must make sure that the gateway's MAC address
is contained in the local system's ARP cache. To achieve this, you may ping the
gateway on each device while the firewall/gateway is (still) accepting ICMP
packets:
```

```
ping -c 1 192.168.72.1
ping -c 1 172.16.70.1

Example how run_fwtest.sh may be called:

./run_fwtest test1 test1.log 192.168.72.0/29 eth0 172.16.70.0/29 eth1 -u

run_fwtest.sh executes the following steps:
(1) Check the arguments for validity.
    Exit if one of them is bad or not specified.
(2) Seek for the required programs, libraries and header files.
    Exit if one of them is missing.
(3) Compile fwtest if necessary.
(4) Set up local firewall rules to drop all incoming packets so that the testing
    host does not response to the packets it captures. This can only be done
    if either iptables or ipchains is installed on your system.
(5) Run fwtest.
    - Run a preprocessor on the packets definition file.
    - Testing is performed (i.e. packets are crafted, injected,
      captured, analyzed) and the irregularities are logged.
    - Fwtest will print some useful information, especially warnings and errors.

    To make this point clear: fwtest is the firewall testing tool that
    performs testing whereas run_fwtest.sh is only a shell script that
    prepares the testing host for the test run and compiles and runs fwtest
    for you. Read more about invoking fwtest in section 3.5.
(6) Remove the local firewall rules.
(7) Exit successfully.

The irregularities are stored in the log file. Evaluate them to identify
problems and abnormalities.


3.4 Test Packets File

3.4.1 General layout

The general layout of a file containing test packet specifications looks like
this:

testcase I {
  packet K { PROTOCOL send { PROT_FIELDS } receive { PROT_FIELDS } }
  packet L { PROTOCOL send { PROT_FIELDS } receive ok }
  packet M { PROTOCOL send { PROT_FIELDS } receive {} }
  packet N { PROTOCOL send { PROT_FIELDS } receive ? }
}
testcase J { ... }

I,J,K,L,M and N are integers which define the testcase/packet id. Every testcase
is run in parallel. The packet id stands for a global timeslot, so for example
the packet 1.2 (testcase 1, packet 2) is sent at the same time as packet 3.2.
```

```
For each packet you have to declare the protocol type (see below) as well as a
send and a receive clause where protocol specific fields are specified. In the
receive part you specify which values you expect the packet to have when the
packet arrives at the destination host. "receive ok" is a shortcut for a receive
block with exactly the same values as in the send clause. You may also have an
empty receive clause {} if you expect the packet to be dropped by the firewall.
A question mark means you're not interested whether the firewall drops or
forwards the packet.


3.4.2 Protocol specific fields (PROT_FIELDS)

Please read documentation [1] for further details on TCP and [3] for more
information on UDP and ICMP.

TCP (Transmission Control Protocol):
{ srcip dstip srcprt dstprt flags seqnr acknr }
    source ip, destination ip, source port, destination port, control flags (a
    combination of S/A/F/R/U/P), sequence number, acknowledgment number

UDP (User Datagram Protocol):
{ srcip dstip srcprt dstprt }
    source ip, destination ip, source port, destination port

ICMPecho (ICMP Echo request/reply):
{ srcip dstip type idnr seqnr }
    source ip, destination ip, type (0/8), identification number, sequence
    number

ICMPunreach (ICMP Destination unreachable):
{ srcip dstip code origid }
    source ip, destination ip, code (0-15), original packet id

ICMPredir (ICMP Redirect):
{ srcip dstip code gwip origid }
    source ip, destination ip, code (0-4), gateway ip, original packet id

ICMPtexc (ICMP Time exceeded):
{ srcip dstip code origid }
    source ip, destination ip, code (0-1), original packet id

ICMPtstamp (ICMP Timestamp request/reply):
{ srcip dstip idnr seqnr otime rtime ttime }
    source ip, destination ip, identification number, sequence number,
    originate timestamp, receive timestamp, transmit timestamp


3.4.3 Preprocessor

Fwtest v1.0 filters the given test packet file with a preprocessor (m4). So you
may define constants for often used IPs or ports. Fwtest includes by default the
```

file defined by the environment variable FWTEST_MACROS. run_fwtest.sh sets this to macros.m4 which contains constants for possible ICMP types and codes.
It's possible to disable the preprocessor by setting the environment variable FWTEST_USEM4 to 'no'.

3.4.4 Variables

You may use variables for IP and port numbers in the packet specifications. As a firewall using NAT masks the source IP/port, you have to use such variables in the receive clause of a packet specification because the real values are not known before starting a test run. During the test, fwtest will automatically assign the variables with the gathered values of the received packets.

Variable names have to start with an uppercase letter and may not be longer than 15 characters. Numbers can be used in variable names except for the first character. Variables are only allowed for IP numbers and for port numbers. A variable is valid in the scope of a testcase. Within a testcase, a variable is assigned only once and all packets that use the same variable, have to have the same value for the field the variable is used. In different testcases the same variable name is NOT the same variable, because it is another scope.

3.4.5 Parse errors

Whenever you get parse errors on a test packet file, the line number will be displayed. Whereas the number is correct for syntax errors, Fwtest v1.0 errs mostly on a semantical error e.g. the discovery of a duplicate packet id. Because packet specifications are internalized only at the end of a testcase, such errors are not detected until the end of the actual testcase.

Since the order of the packets is not relevant (only the packet id specifies the time slot), Fwtest reverses the packets to simplify matters because of the internal data structure; so, errors may be detected in theo opposite order than they appear in the specification.

3.5 Running Fwtest

We briefly explain how to invoke fwtest without making use of run_fwtest.sh. fwtest expects the test packets file name as argument, and takes the following options:

```
        -l <log file>  Log file wherein the irregularities are stored
        -n <network>   Network 1 and mask in CIDR notation (e.g. 192.168.1.0/29)
        -i <interface> Network 1 interface to capture the packets.
        -m <network>   Network 2 and mask in CIDR notation (e.g. 172.16.70.0/29)
        -j <interface> Network 2 interface to capture the packets.
        -p             Transmit packet id in payload (neglecting ICMP)
        -u             Interpret TCPUDP packets as UDP instead of TCP
```

You have to specify the networks and the interfaces (-n, -i, -m and -j).

Fwtest may be called like this (make sure you are root):

```
./main -l test1.log -n 192.168.72.0/29 -i eth0 -m 172.16.70.0/29 -j eth1 test.tp
```

The test packets file (e.g. test.tp) has to be defined. You have no chance to perform a test without a test packets file. If the syntax of the file is invalid, fwtest will display an error message and quit instantly.

3.6 Example

Let’s just show at a simple test run in a possible environment:

Test host setup:

```
ifconfig eth0 down
ifconfig eth1 down
ifconfig eth0 172.16.70.2 netmask 255.255.255.0 up
ifconfig eth1 192.168.72.2 netmask 255.255.255.0 up
route add -net 172.16.70.0/24 gw 172.16.70.1
route add -net 192.168.72.0/24 gw 192.168.72.1
```

Firewall setup:

```
ifconfig eth0 down
ifconfig eth1 down
ifconfig eth0 172.16.70.1 netmask 255.255.255.0 up
ifconfig eth1 192.168.72.1 netmask 255.255.255.0 up
echo 1>  /proc/sys/net/ipv4/ip_forward
echo 1>  /proc/sys/net/ipv4/conf/default/proxy_arp
```

Ping the firewall from the test host (the firewall is still accepting all incoming packets):

```
ping -c 1 172.16.70.1
ping -c 1 192.168.72.1
```

Install the firewall rules on the firewall (e.g. drop all packets, but forward tcp packets to 192.168.72.3:25):

```
iptables -F
iptables -X
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -P FORWARD DROP
iptables -A FORWARD -p tcp -d 192.168.72.3 --dport 25 -j ACCEPT
```

Create a test file named ‘test.tp’ like this:

```
define(ipa, 172.16.70.3)
define(ipb, 192.168.72.3)
testcase 1 {
```

```
        packet 1 { TCP send { ipa ipb 4000 25 S 60 - } receive ok }
        packet 2 { ICMPunreach send { ipb ipa PORT_UNREACHABLE 1.1 }
                            receive {} }
    }
    testcase 2 {
        packet 1 { UDP send { ipa ipb 5005 domain } receive {} }
    }
```

Now fire off fwtest by calling run_fwtest.sh:

```
    ./run_fwtest.sh test.tp test.log 172.16.70.0/24 eth0 192.168.72.0/24 eth1
```

After the test run is complete, the statistics will be displayed which reports one packet as successfully forwarded (true_negative) and packet 1.2 and 2.1 as successfully rejected (true_positive).

4. Timeout

A timer is startet by fwtest after the packets for a given timeslot are sent out. When the timer expires after a specified timeout value, fwtest steps to the next timeout. The timeout value is stored in the header file main.h. After changeing the timeout value, fwtest has to be recompiled.

5. Source Files

```
Makefile          - Makefile
main.c            - main program
tpparser.l        - lexer grammar
tpparser.y        - parser grammar
lpcap.c           - packet capturing routines
lnet.c            - packet crafting and injection routines
log.c             - log routines
util.c            - helper routines
symboltable.c     - a symbol table for the test packets file variables
semanticchecker.c - a semantic-checker for the test packets file
main.h            - main definitions & TIMEOUT definition
tpparser.h        - parse definitions
lpcap.h           - packet capture definitions
lnet.h            - packet crafting definitions
log.h             - log definitions
symboltable.h     - symbol table definitions
semanticchecker.h - semantic-checker definitions
```

D. Aufgabenstellung

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information Security
www.infsec.ethz.ch

Semester thesis for Beat Strasser

Firewall Testing

An extension to fwtest to handle UDP and ICMP

Supervisor:	Diana Senn
Professor:	Prof. D. Basin
Issue Date:	August 2005
Submission Date:	February 2006

1 Introduction

We live in a world where all the company networks are connected to the Internet. Nobody can control the Internet, therefore a company has to protect their data from unauthorised access through the Internet. This is done by firewalls whose analogon in the physical world are locks. Everybody understands that doors need to be locked to prevent unauthorised access. It is the same in the digital world: unauthorised access to a companies network should be prevented, and this can be done by one or several firewalls.
Using the analogon of the door lock again, everybody understands that it is not enough to have a door lock. Only if the lock is locked properly and only authorised people have got a key to unlock it, we have what we want. It is the same in the digital world. It is not enough to have a firewall. We can only be satisfied if the firewall is doing what we expect from it. And to find out if a firewall satisfies our expectations (stated by a policy) we need to test it.

2 Motivation

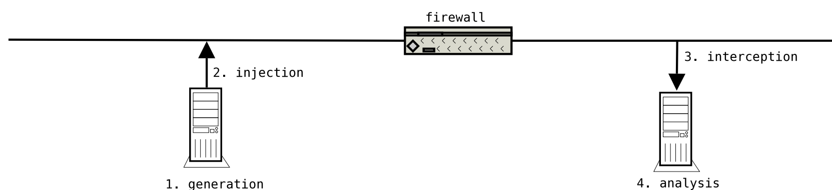


Figure 1: Flow of the Test Packets

Firewall Testing consists of two parts: the theoretical part of finding adequate test cases, and the practical part of running these test cases on the real system. The aim of this thesis is to cover the second part.

1

Running test cases on a real system consists of four steps as shown in figure 1. The first step is the generation of the network packets. According to a given test case the corresponding network packets have to be built. The second step consists of injecting the packets generated in step one into the network. The third and the fourth step are then to intercept the packets which were injected in step two behind the firewall, comparing them to the expectations, and logging the result.

3 Assignement

3.1 Objectives

During his diploma thesis [1], Gerhard Zaugg has written a simple tool – named fwtest – which can do the above for TCP packets in a bidirectional way. As there are also other interesting low-level protocols passing the firewall, the goal of this semester thesis is to extend fwtest to being able to generate UDP and ICMP packets as well.

3.2 Tasks

- Understanding fwtest
- Designing a format for UDP and ICMP tp-files
- Adding generation methods for UDP packets to fwtest
- Adding generation methods for ICMP packets to fwtest
- Conducting tests to show the correct behaviour of the implementation

As there will be another student (Adrian Schüpbach) working on fwtest in an overlapping time frame, some synchronisation between the two students is expected. This will mainly consist in determining a new format for the tp-files together and working on the same subversion repository.

3.3 Deliverables

- At the beginning of the semester thesis an agreement must be signed which allows the supervisor of this thesis, his project partners and ETH Zurich to use and distribute the software written during the thesis.
- At the end of the second week, a detailed time schedule of the semester thesis must be given and discussed with the supervisor.
- At the end of the semester thesis a presentation of 20 minutes must be given during an Infsec group seminar. It should give an overview as well as the most important details of the work.
- The final report may be written in English or German. It must contain an abstract written in both English and German, this assignement and the schedule. It should include an introduction, an analysis of related work, and a complete documentation of all used software tools. Three copies of the final report must be delivered to the supervisor.

2

- Software and configuration scripts developed during the thesis must be delivered to the supervisor on a CD-ROM.

References

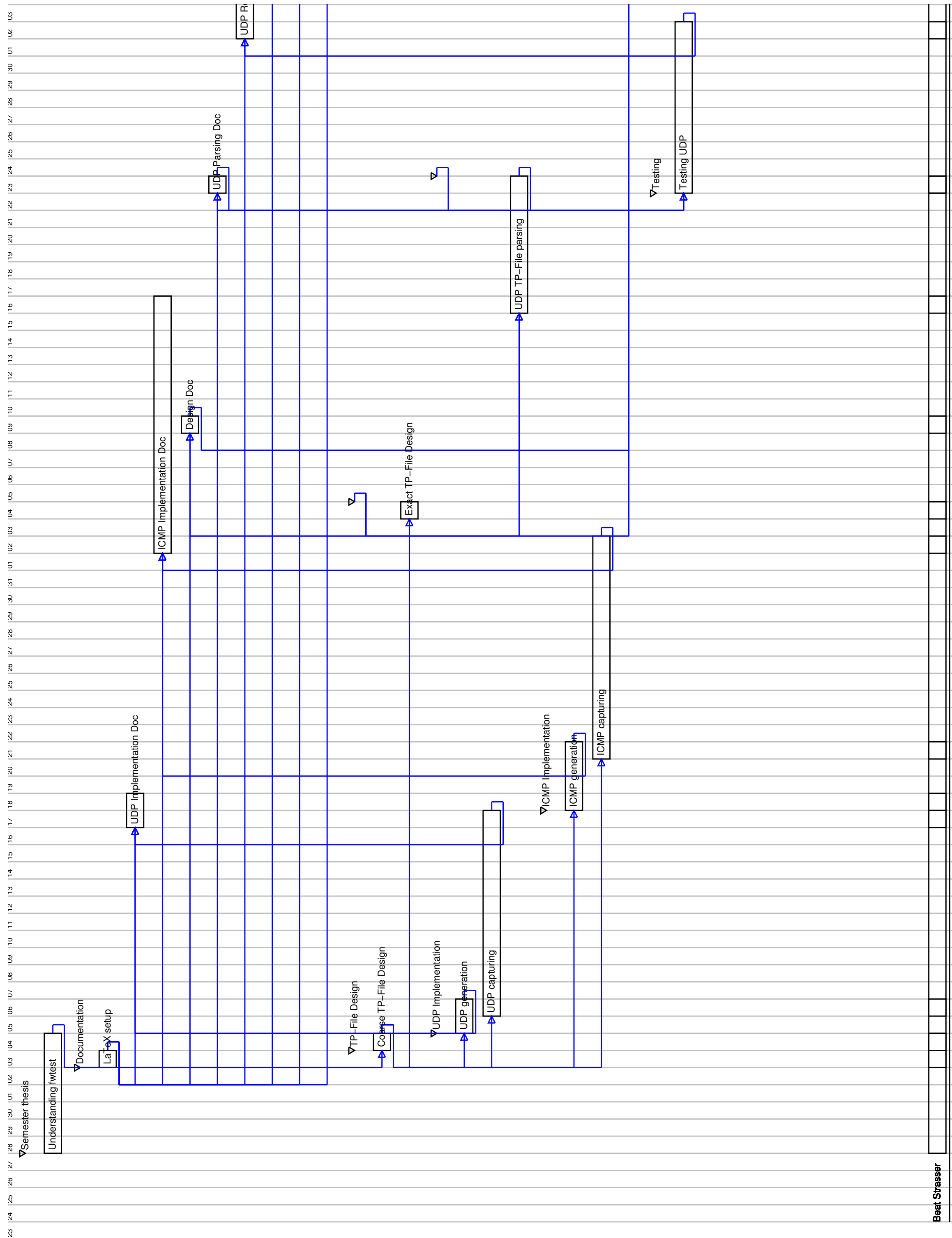
[1] Gerry Zaugg. Firewall testing. http://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg_05.pdf.

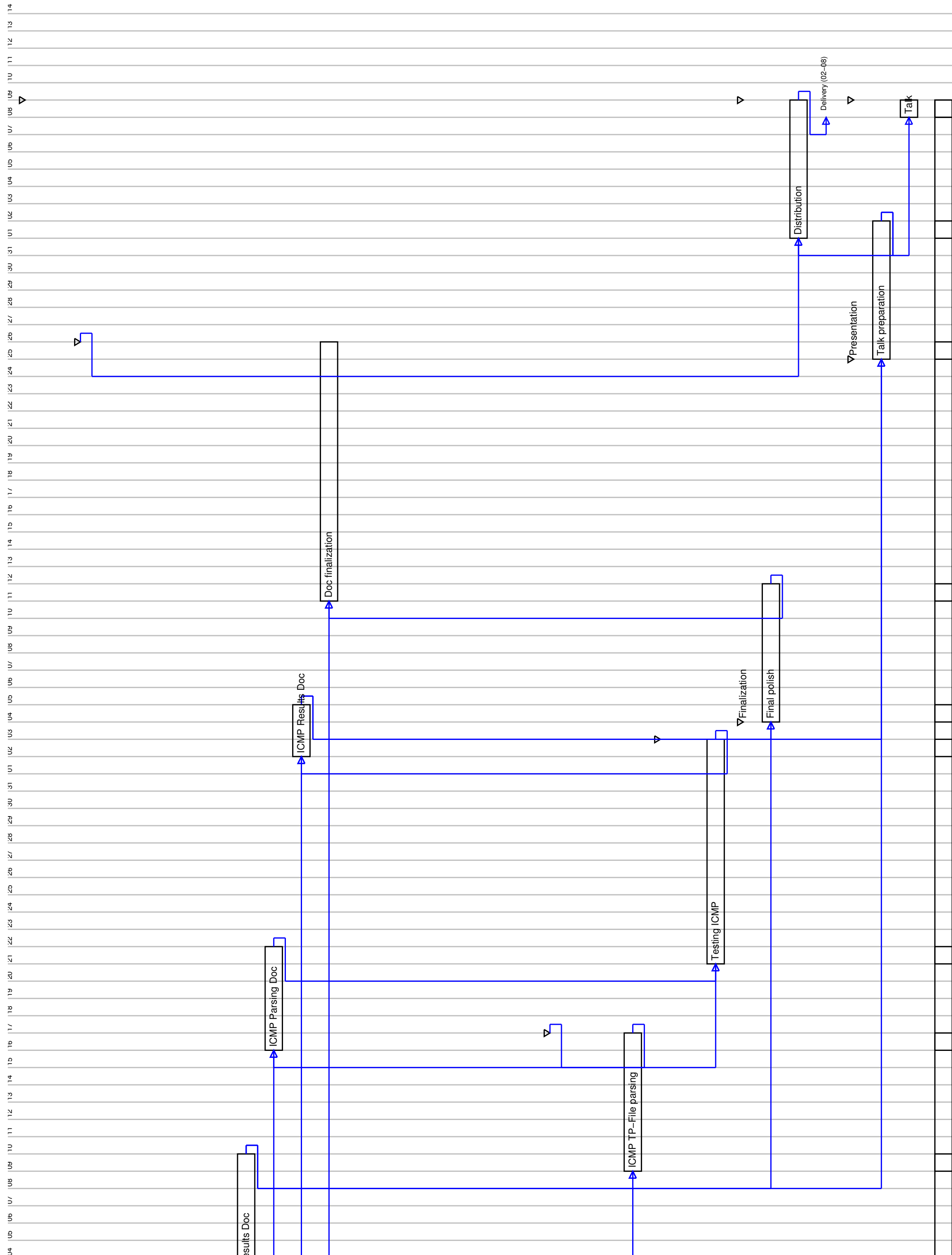
16th August 2005

Prof. D. Basin

3

E. Zeitplan




Delivery (02–08)
Talk
Distribution
Presentation
Talk preparation
Doc finalization
ICMP Results Doc
Finalization
Final polish
Testing ICMP
ICMP Parsing Doc
ICMP TP–File parsing

Literatur

[Bra89] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989. Updated by RFC 1349.

[Com05] Gerald Combs. Ethereal 0.10.12. http://www.ethereal.com/, August 2005.

[CS05] Robert Corbett and Richard Stallman. GNU Bison v2.1. http://www.gnu.org/software/bison/, September 2005.

[MD90] J.C. Mogul and S.E. Deering. Path MTU discovery. RFC 1191 (Draft Standard), November 1990.

[Mil92] D. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305 (Draft Standard), March 1992.

[Pos80] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[Pos81a] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. Updated by RFC 950.

[Pos81b] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[Sch06] Adrian Schüpbach. Testen von Firewalls mit NAT. http://www.infsec.ethz.ch/people/dsenn/SA_AdrianSchuepbach_06.pdf, February 2006.

[Sei05] Rene Seindal. GNU m4 v1.4.4. http://www.gnu.org/software/m4/, October 2005.

[vmw04] VMWare Workstation 4.5 documentation. http://www.vmware.com/support/ws45/doc/, April 2004.

[Zau05] Gerry Zaugg. Firewall Testing. http://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg_05.pdf, January 2005.