iptables

- iptables das klassische Firewall-Werkzeug unter Linux
- Block IPs
- Block IPs /Docker
- iptables Regeln dauerhaft speichern
- Regeln gegen PortScans

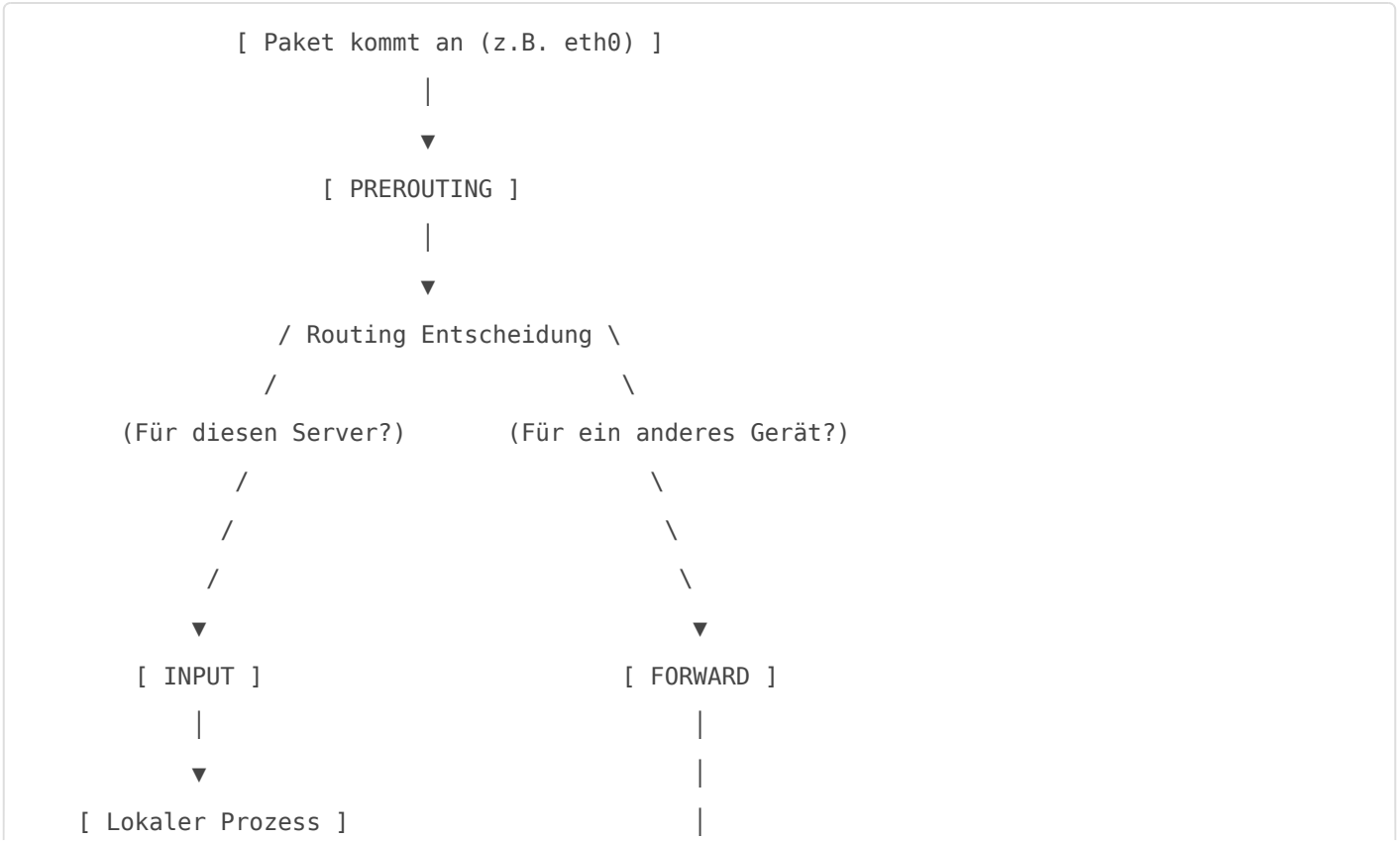
iptables das klassische Firewall-Werkzeug unter Linux

Auch wenn modernere Linux-Distributionen standardmäßig oft auf `nftables` setzen, bleibt das logische Prinzip von `iptables` die absolute Grundlage für das Verständnis von Netzwerk-Firewalls unter Linux.

Teil 1: Allgemeiner Überblick & Paketfluss (Routing)

Wenn ein Netzwerkpaket (z. B. ein TCP-Paket) eine Netzwerkkarte des Linux-Servers erreicht, durchläuft es den Linux-Kernel. Die Firewall `iptables` ist direkt in diesen Kernel-Prozess (über das Framework *Netfilter*) integriert.

Hier ist der vereinfachte Weg eines Pakets durch das System:



```
 (z.B. Webserver)                          |
        |                                  |
        ▼                                  |
[ Lokaler Prozess                          |
  sendet Paket ]                           |
        |                                  |
        ▼                                  |
   [ OUTPUT ]                              |
        |                                  |
        ▼                                  ▼
        \ Routing-Entscheidung /————————————┘
                  |
                  ▼
             [ POSTROUTING ]
                  |
                  ▼
          [ Paket verlässt Server ]
```

Die drei Haupt-Szenarien für Pakete:

1. **Eingehender Traffic für den Server selbst (z. B. SSH-Anfrage auf den Server):**
   - Das Paket kommt an der Netzwerkkarte an.
   - Es durchläuft die Kette **PREROUTING**.
   - Das System entscheidet über das Routing: "Das Paket ist für mich bestimmt."
   - Das Paket durchläuft die Kette **INPUT**.
   - Der lokale Dienst (z. B. der SSH-Daemon) empfängt das Paket.
2. **Ausgehender Traffic vom Server selbst (z. B. der Server macht ein System-Update):**
   - Ein lokaler Prozess erzeugt das Paket.
   - Das Paket durchläuft die Kette **OUTPUT**.
   - Eine Routing-Entscheidung bestimmt die ausgehende Schnittstelle.
   - Das Paket durchläuft die Kette **POSTROUTING**.
   - Das Paket verlässt den Server.
3. **Durchlaufender Traffic (Der Server agiert als Router/Gateway):**
   - Das Paket kommt an der Netzwerkkarte an.
   - Es durchläuft **PREROUTING**.
   - Routing-Entscheidung: "Das Paket ist für eine andere IP-Adresse im Netz bestimmt."
   - Das Paket durchläuft die Kette **FORWARD** (es wird durchgereicht).
   - Das Paket durchläuft **POSTROUTING**.
   - Das Paket verlässt den Server über eine andere (oder dieselbe) Netzwerkkarte.

Teil 2: Das iptables-Konzept (Tables, Chains, Rules, Targets)

`iptables` ist hierarchisch aufgebaut. Die Struktur lässt sich wie folgt zusammenfassen:

- Ein **Table** (Tabelle) bestimmt den *Zweck* der Paketverarbeitung (z.B. Filtern vs. Adressübersetzung).
- Ein Table enthält mehrere **Chains** (Ketten), die dem *Zeitpunkt* der Verarbeitung im Paketfluss entsprechen.
- Eine Chain enthält eine Liste von **Rules** (Regeln), die nacheinander von oben nach unten abgearbeitet werden.
- Wenn eine Rule zutrifft, wird ein **Target** (Ziel/Aktion) ausgeführt (z.B. Erlauben oder Blockieren).

1. Tables (Die Tabellen)

Es gibt vier Haupt-Tabellen in `iptables`. Wenn Sie beim Befehl keine Tabelle angeben, wird automatisch die Standardtabelle `filter` verwendet.

- `filter` **(Standard-Tabelle):**
  - **Zweck:** Die eigentliche Firewall. Hier wird entschieden, ob ein Paket durchgelassen oder blockiert wird.
  - **Chains:** `INPUT`, `FORWARD`, `OUTPUT`.
- `nat` **(Network Address Translation):**
  - **Zweck:** IP-Adressen oder Ports umschreiben. Wird für Router, Port-Forwarding (DNAT) oder das Teilen einer Internetverbindung (Masquerading/SNAT) genutzt.
  - **Chains:** `PREROUTING`, `OUTPUT`, `POSTROUTING`.
- `mangle` **(Paketmanipulation):**
  - **Zweck:** Spezielle Modifikationen am IP-Header (z.B. Ändern der "Time to Live" (TTL) oder Markieren von Paketen für spezielles Routing).
  - **Chains:** Alle 5 Chains (`PREROUTING`, `INPUT`, `FORWARD`, `OUTPUT`, `POSTROUTING`).
- `raw` **(Rohdaten-Verarbeitung):**
  - **Zweck:** Pakete von der Verbindungsverfolgung (Connection Tracking / `conntrack`) ausschließen. Sehr selten benötigt, meist zur Performance-Optimierung bei DDoS-Angriffen.
  - **Chains:** `PREROUTING`, `OUTPUT`.

---

2. Chains (Die Ketten)

Chains entsprechen den Stationen im Netzwerkfluss (siehe Diagramm oben). Es gibt vordefinierte (eingebaute) Chains und Sie können eigene, benutzerdefinierte Chains erstellen.

Die 5 Standard-Chains sind:

1. **`PREROUTING`:** Greift, sobald ein Paket die Netzwerkkarte betritt – *bevor* das System entscheidet, wohin das Paket geroutet wird.
2. **`INPUT`:** Greift für Pakete, die direkt an Ihren Server gerichtet sind.
3. **`FORWARD`:** Greift für Pakete, die den Server nur passieren (Routing/Gateway).
4. **`OUTPUT`:** Greift für Pakete, die von Ihrem Server selbst generiert wurden.
5. **`POSTROUTING`:** Greift kurz vor dem Verlassen der Netzwerkkarte – *nachdem* das Routing bereits stattgefunden hat.

---

3. Rules (Die Regeln)

Eine Regel besteht aus zwei Teilen: den **Bedingungen** (Matches) und der **Aktion** (Target).

Wenn ein Paket eine Kette durchläuft, prüft `iptables` die Regeln von oben nach unten. Sobald eine Regel komplett auf das Paket zutrifft, wird die zugehörige Aktion ausgeführt und die Suche in dieser Kette ist in der Regel beendet.

Wichtige Parameter zum Erstellen von Bedingungen (Matches):

- `-p` (Protocol): Welches Protokoll? (z.B. `tcp`, `udp`, `icmp`).
- `-s` (Source): Woher kommt das Paket? (IP-Adresse oder Subnetz, z.B. `192.168.1.50` oder `192.168.1.0/24`).
- `-d` (Destination): Wohin geht das Paket? (z.B. IP des Servers).
- `--sport` / `--dport` (Source-/Destination-Port): Welcher Port? (z.B. `--dport 22` für SSH oder `--dport 80` für HTTP). *Hinweis: Erfordert immer die vorherige Angabe des Protokolls mit* `-p`.
- `-i` (Input-Interface): Über welche Netzwerkkarte kommt das Paket rein? (z.B. `-i eth0`).
- `-o` (Output-Interface): Über welche Netzwerkkarte geht das Paket raus? (z.B. `-o wlan0`).
- `-m state` / `-m conntrack` (Verbindungszustand): Ist das Paket Teil einer bekannten Verbindung? (z.B. `--state ESTABLISHED,RELATED`).

---

4. Targets (Die Ziele / Aktionen)

Das Target bestimmt, was mit dem Paket passiert, wenn die Bedingungen einer Regel erfüllt sind. Man gibt es mit dem Parameter `-j` (Jump) an.

Die wichtigsten **beendenden** Targets (das Paket verlässt die Kette):

- **`ACCEPT`:** Das Paket wird durchgelassen. Die Verarbeitung in dieser Kette stoppt.
- **`DROP`:** Das Paket wird wortlos verworfen. Der Absender erhält keine Rückmeldung und läuft in einen Timeout (sicherste Methode für das Internet).
- **`REJECT`:** Das Paket wird abgewiesen, aber der Absender erhält eine Fehlermeldung (z.B. eine ICMP "Port Unreachable"-Nachricht). Sinnvoll in internen Netzen zur schnellen Fehlersuche.

Wichtige **nicht-beendende** oder spezielle Targets:

- **`LOG`:** Das Paket wird im Kernel-Log protokolliert (meist einsehbar via `dmesg` oder `/var/log/syslog`), danach wird die nächste Regel in der Kette geprüft.
- **`MASQUERADE`:** (Nur in der `nat`-Tabelle) Ersetzt die private Absender-IP durch die öffentliche IP der ausgehenden Schnittstelle (praktisch bei dynamischen IPs).
- **`SNAT` (Source NAT):** (Nur in der `nat`-Tabelle) Ändert die Quell-IP-Adresse des Pakets auf einen festen Wert.
- **`DNAT` (Destination NAT):** (Nur in der `nat`-Tabelle) Ändert die Ziel-IP-Adresse des Pakets (wichtig für Portweiterleitungen).

Praktisches Beispiel zum Verständnis

Nehmen wir an, Sie möchten Ihren SSH-Port (Port 22) für alle sperren, außer für Ihren administrativen PC mit der IP `192.168.1.100`.

Der Befehl dafür sieht so aus:

```
# 1. Erlaube SSH-Verbindungen von der Admin-IP
iptables -A INPUT -p tcp -s 192.168.1.100 --dport 22 -j ACCEPT

# 2. Verbiete SSH-Verbindungen für alle anderen IPs
iptables -A INPUT -p tcp --dport 22 -j DROP
```

Erklärung des ersten Befehls:

- `iptables`: Das Programm aufrufen.
- `-A INPUT`: Hänge die Regel an (**A**ppend) das Ende der Kette **INPUT** (in der Standardtabelle `filter`).
- `-p tcp`: Betrifft nur das Protokoll **TCP**.
- `-s 192.168.1.100`: Betrifft nur diese Quell-IP (**S**ource).
- `--dport 22`: Betrifft nur den Ziel-Port (**D**estination Port) 22.
- `-j ACCEPT`: Wenn all das zutrifft, springe (**J**ump) zur Aktion **ACCEPT** (erlauben).

Reihenfolge ist entscheidend: Würde man zuerst den `DROP`-Befehl für alle und danach erst den `ACCEPT`-Befehl für die Admin-IP eingeben, würde die Admin-IP ebenfalls blockiert. Da `iptables` die Regeln von oben nach unten abarbeitet, trifft beim Admin sofort die erste Regel ("Sperre Port 22 für alle") zu, und das Paket wird verworfen. Die zweite Regel würde nie erreicht werden.

Wie funktionieren Custom Chains?

Zusätzliche Ketten (sogenannte **Custom Chains** oder benutzerdefinierte Ketten) wie `DOCKER-USER`, `DOCKER-FORWARD` oder `DOCKER-ISOLATION` **werden nicht automatisch abgearbeitet**.

Der Linux-Kernel kennt von Natur aus nur die 5 eingebauten Standard-Ketten (`INPUT`, `OUTPUT`, `FORWARD`, `PREROUTING`, `POSTROUTING`). Ein Paket landet niemals "einfach so" in einer benutzerdefinierten Kette.

Eine benutzerdefinierte Kette funktioniert wie ein **Unterprogramm** (eine Funktion) in einer Programmiersprache. Sie muss explizit aus einer der Standard-Ketten heraus aufgerufen werden.

Dies geschieht über den Ihnen bereits bekannten Parameter **`-j` (Jump)**.

Das Prinzip des "Sprungs" (Jump & Return)

Wenn ein Paket die Kette `FORWARD` durchläuft, sieht die Auswertung so aus:

1. **`FORWARD` Regel 1:** Trifft Bedingung X zu? Wenn ja, springe (`-j`) in die Kette `DOCKER-USER`.
   - *Jetzt wird die Kette `DOCKER-USER` von oben nach unten abgearbeitet.*
   - Wenn dort eine Regel mit `ACCEPT` oder `DROP` greift, ist das Schicksal des Pakets besiegelt.
   - Wenn **keine** Regel zutrifft (oder eine Regel mit `-j RETURN` greift), springt das Paket **zurück** in die `FORWARD`-Kette zur Regel 2.
2. **`FORWARD` Regel 2:** Springe in die Kette `DOCKER-FORWARD`.
   - Das Paket durchläuft `DOCKER-FORWARD`.
   - Wenn kein Treffer: Zurück zu `FORWARD` Regel 3.
3. **`FORWARD` Regel 3:** ... und so weiter.

Die konkrete Reihenfolge bei Docker

Docker nutzt dieses Prinzip intensiv, um den Netzwerkverkehr für Container zu regeln. Da Container-Traffic in der Regel durch den Host hindurchgeroutet wird, findet fast die gesamte Docker-Magie in der Standard-Kette **`FORWARD`** statt.

Wenn Sie Docker installieren, klinkt es sich in die `FORWARD`-Kette ein und baut folgende Reihenfolge auf:

```
Eingehendes Paket im FORWARD-Prozess
   │
   ▼
1. [ Sprung zu DOCKER-USER ] ──────────► Hier liegen IHRE benutzerdefinierten Regeln.
   │ (Falls kein Treffer, zurück)       (z. B. "Blockiere IP X, die auf Container zugreift")
   ▼
2. [ Sprung zu DOCKER-FORWARD ] ──────► Docker prüft erste Routing- und Basisregeln.
   │ (Falls kein Treffer, zurück)
   ▼
3. [ RELATED,ESTABLISHED ACCEPT ] ────► Ist das Paket Teil einer bereits erlaubten,
```

```
   |                                  bestehenden Verbindung? Wenn ja: ACCEPT (Erlauben).
   ▼
4. [ Sprung zu DOCKER ] ──────────► Hier liegen die Regeln für Port-Weiterleitungen.
   | (Falls kein Treffer, zurück)       (z.B. "Darf Traffic auf Port 80 des Containers?")
   ▼
5. [ Container ausgehend/intern ] ──► Regeln, die bestimmen, ob Container ins Internet
   |                                  dürfen oder untereinander kommunizieren dürfen.
   ▼
6. [ Default Policy: DROP ] ────────► Wenn bis hierhin kein "ACCEPT" kam, wird das Paket
                                      vom System weggeworfen.
```

(Hinweis: Je nach Docker-Version und Betriebssystem können die Ketten minimal anders heißen, z. B. `DOCKER-ISOLATION-STAGE-1` anstelle von `DOCKER-FORWARD`, das logische Prinzip bleibt aber exakt dasselbe.)

Warum macht Docker das so? (Das `DOCKER-USER`-Geheimnis)

Ein häufiges Problem bei Docker-Anfängern ist folgendes Szenario: Sie starten einen Container und geben einen Port frei (z. B. `-p 80:80`). Danach versuchen sie, diesen Port über die normale `INPUT`-Kette der Firewall zu sperren. **Das funktioniert nicht!** Da der Traffic für den Container bestimmt ist, läuft er nicht über `INPUT`, sondern über `FORWARD`.

Wenn Sie nun einfach eine Sperre ans Ende der `FORWARD`-Kette hängen würden, würde diese ebenfalls ignoriert. Warum? Weil Docker weiter oben in der Kette (im Schritt 4 bei `DOCKER`) das Paket bereits mit `ACCEPT` durchgelassen hat. Sobald ein Paket ein `ACCEPT` erhält, ist die Prüfung beendet und spätere Regeln werden ignoriert.

Die Lösung von Docker: Docker hat ganz oben als allererste Regel in der `FORWARD`-Kette den Sprung nach **`DOCKER-USER`** eingebaut.

- Docker selbst fasst diese Kette niemals an.
- Sie ist exakt dafür da, dass **Sie als Admin** dort Regeln eintragen können.
- Da diese Kette ganz am Anfang geprüft wird, können Sie hier unerwünschten Traffic blockieren (`DROP`), *bevor* die automatischen Erlaubnis-Regeln von Docker überhaupt greifen.

Beispiel: Eine IP in Docker-User sperren

Wenn Sie die bösartige IP `203.0.113.50` komplett von Ihren Docker-Containern aussperren wollen, schreiben Sie:

```
iptables -I DOCKER-USER -s 203.0.113.50 -j DROP
```

> *Hinweis: Das `-I` steht für **I**nsert. Es fügt die Regel ganz oben als Regel Nr. 1 in die `DOCKER-USER`-Kette ein.*

Block IPs

Eine einzelne IP blockieren

```
sudo iptables -A INPUT -s 192.168.1.100 -j DROP
```

- `-A INPUT` → Regel zur Eingangs-Kette hinzufügen
- `-s 192.168.1.100` → Quelle (die zu blockierende IP)
- `-j DROP` → Pakete einfach verwerfen (keine Antwort)

IP komplett sperren (ein- und ausgehend)

```
sudo iptables -A INPUT -s 192.168.1.100 -j DROP
sudo iptables -A OUTPUT -d 192.168.1.100 -j DROP
```

Statt DROP: aktiv ablehnen (REJECT)

```
sudo iptables -A INPUT -s 192.168.1.100 -j REJECT
```

- `DROP` → keine Antwort (wirkt wie "unsichtbar")
- `REJECT` → sendet Fehlermeldung zurück

Regel überprüfen

```
sudo iptables -L -n
```

Regeln dauerhaft speichern

Je nach System:

- **Debian/Ubuntu:**

```
sudo apt install iptables-persistent
sudo netfilter-persistent save
```

- **CentOS/RHEL:**

```
sudo service iptables save
```

Regel wieder entfernen

```
sudo iptables -D INPUT -s 192.168.1.100 -j DROP
```

Gezieltes Entfernen per Zeilennummer

```
iptables -L --line-numbers | grep <IP>
iptables -D <CHAIN> <Zeilennummer>
```

Block IPs /Docker

Docker umgeht teilweise die normalen iptables-Regeln, weil es eigene Regeln und Chains anlegt.

Docker setzt eigene Chains wie:

- `DOCKER`
- `DOCKER-USER`

□□ Traffic zu Containern wird oft **vor den normalen INPUT-Regeln verarbeitet**.

Deshalb greifen die Block-Regeln nicht wie erwartet.

Die Lösung: `DOCKER-USER` Chain nutzen

Docker hat extra eine Chain vorgesehen, damit man genau sowas machen kannst:

```
sudo iptables -I DOCKER-USER -s 192.168.1.100 -j DROP
```

□□ Wichtig:

- `-I` (insert) statt `-A`, damit die Regel **ganz oben** landet
- Diese Chain wird **immer vor Docker-Regeln ausgewertet**

---

iptables Regeln dauerhaft speichern

iptables-Regeln sind standardmäßig **temporär** und gehen nach einem Neustart des Servers verloren.

- **Unter Debian / Ubuntu:** Installiere das Paket `iptables-persistent` :

  ```
  sudo apt-get install iptables-persistent
  ```

  Nach dem Ändern von Regeln speicherst du sie mit:

  ```
  sudo netfilter-persistent save
  ```

- **Unter CentOS / RHEL / Rocky Linux:**

  ```
  sudo service iptables save
  ```

Regeln gegen PortScans

Diese vier `iptables`-Befehle dienen der Absicherung des Servers gegen unübliche Netzwerkpakete. Sie werden häufig eingesetzt, um **Port-Scans** (Sondierungen durch Angreifer) zu erkennen, zu protokollieren (loggen) und zu blockieren.

1. Null-Scan-Erkennung und Ratenbegrenzung

```
iptables -A INPUT -p tcp --tcp-flags ALL NONE -m limit --limit 1/h -j ACCEPT
```

- **`-A INPUT`**: Hängt die Regel an die `INPUT`-Kette an (gilt für alle eingehenden Pakete).
- **`-p tcp`**: Filtert nur den TCP-Netzwerkverkehr.
- **`--tcp-flags ALL NONE`**: Überprüfe alle TCP-Flags (SYN, ACK, FIN, RST, URG, PSH), aber keines davon darf gesetzt sein (`NONE`). Ein TCP-Paket ohne jegliche Flags ist laut Netzwerkstandard (RFC) ungültig. Angreifer nutzen dies für einen sogenannten **„Null-Scan“**, um offene Ports zu finden.
- **`-m limit --limit 1/h`**: Nutzt das Limit-Modul. Diese Regel greift nur maximal **1-mal pro Stunde** (`1/h`).
- **`-j ACCEPT`**: Lässt das Paket durch (erlaubt es).
- **Sinn dieser Regel:** Sie erlaubt maximal ein einziges solches "Null-Scan"-Paket pro Stunde (z. B. für legitime Messungen oder Toleranz bei Übertragungsfehlern). Alle weiteren Null-Scan-Pakete, die innerhalb dieser Stunde eintreffen, ignorieren diese Regel und fallen in der Firewall-Liste weiter nach unten, wo sie in der Regel durch eine allgemeine BLOCK-Regel blockiert werden.

---

2. Xmas-Scan-Erkennung und Ratenbegrenzung

```
iptables -A INPUT -p tcp --tcp-flags ALL ALL -m limit --limit 1/h -j ACCEPT
```

- **`--tcp-flags ALL ALL`**: Überprüfe alle TCP-Flags, und **alle** müssen gleichzeitig gesetzt sein (`ALL`). Ein Paket, bei dem alle Flags aktiv sind, ist im normalen Netzwerkverkehr unmöglich. Man nennt dies einen **„Xmas-Scan“** (Weihnachtsbaum-Scan), weil das Paket quasi „wie ein Weihnachtsbaum leuchtet“. Auch dies wird von Angreifern zur Erkennung des Betriebssystems oder offener Ports genutzt.
- **`-m limit --limit 1/h -j ACCEPT`**: Wie beim ersten Befehl wird auch hier maximal 1 solches Paket pro Stunde akzeptiert, um Fehlalarme oder minimale Tests zuzulassen. Der Rest

wird durch nachfolgende Regeln blockiert.

---

3. Ungültige Verbindungsaufbaue protokollieren (Loggen)

```
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j LOG --log-prefix "Stealth Scan"
```

- `! --syn`: Das Ausrufezeichen steht für eine Verneinung (NOT). Es bedeutet: Das TCP-Paket hat das `SYN`-Flag **nicht** gesetzt.
- `-m state --state NEW`: Das Paket versucht laut Verbindungsverfolgung (Connection Tracking) eine **neue** (`NEW`) Verbindung aufzubauen.
- **Sinn dieser Kombination:** Ein regulärer, legaler TCP-Verbindungsaufbau (Three-Way-Handshake) **muss** zwingend mit einem `SYN`-Paket beginnen. Wenn ein Paket eine neue Verbindung anfordert (`NEW`), aber kein `SYN`-Flag trägt (`! --syn`), ist das technisch unmöglich und deutet stark auf einen Port-Scan (z. B. FIN- oder ACK-Scan) oder einen Angriffsversuch hin.
- `-j LOG --log-prefix "Stealth Scan"`: Dieses Paket wird nicht blockiert, sondern im System-Log (meist `/var/log/messages` oder `dmesg`) mit dem Präfix „Stealth Scan“ protokolliert, damit der Administrator den Angriffsversuch nachvollziehen kann.

---

4. Ungültige Verbindungsaufbaue blockieren (Droppen)

```
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
```

- **Match-Kriterien (identisch zu Befehl 3):** Es sucht wieder nach TCP-Paketen, die eine neue Verbindung aufbauen wollen, ohne das `SYN`-Flag zu besitzen.
- `-j DROP`: Dieses Paket wird sofort und kommentarlos verworfen (gelöscht).
- **Zusammenspiel mit Befehl 3:** Diese beiden Befehle (3 und 4) werden in genau dieser Reihenfolge hintereinander in die Firewall eingepflegt. Dadurch wird das verdächtige Paket zuerst registriert und im Logbuch vermerkt (Befehl 3) und direkt im nächsten Schritt blockiert (Befehl 4).

Hier nochmal zusammengefaßt

```
iptables -A INPUT -p tcp --tcp-flags ALL NONE -m limit --limit 1/h -j ACCEPT
iptables -A INPUT -p tcp --tcp-flags ALL ALL -m limit --limit 1/h -j ACCEPT
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j LOG --log-prefix "Stealth Scan: "
iptables -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
```