Ein Gateway für SSH

Mirko Dziadzka
http://mirko.dziadzka.net

15. April 2005

Inhaltsverzeichnis

1 Einleitung **3**

2 Das Problem **3**

3 Eigenschaften von OpenSSH **4**

3.1 Forced Commands 4

3.2 Client-seitige Konfiguration und ProxyCommand 6

4 Implementation des SSH Gateway **8**

5 Was jetzt noch fehlt **11**

5.1 Implementation der Autorisierung 11

5.2 Sicherheit 11

5.3 Hängende Sessions 12

6 Gelerntes **12**

6.1 TCP Forwarder 12

6.2 Nutzerfreundlichkeit 12

1 Einleitung

Seit nunmehr knapp 10 Jahren ist SSH das Standard Tool für den Remote Zugriff in der Unix Welt. Es stellt Vertraulichkeit, Integrität und Authentizität einer Terminalsession sicher.

Im folgenden soll etwas auf die weiteren Möglichkeiten von SSH, genauer der OpenSSH Implementation, eingegangen und diese an einem praktischen Beispiel demonstriert werden.

Der Autor hat die hier beschriebene SSH Gateway Lösung vor zwei Jahren in einer sicherheitskritischen Umgebung umgesetzt und seitdem begleitet.

2 Das Problem

Am Anfang stand die Aufgabe: „Zu implementieren ist ein SSH Gateway als Teil einer Firewallinfrastruktur.“

Im konkreten Fall ging es um die Einschränkung und zentrale Kontrolle von Zugriffen, die die Unix Admins einer Firma auf diverse, strikt getrennte Kundenprojekte haben (Abbildung 1).

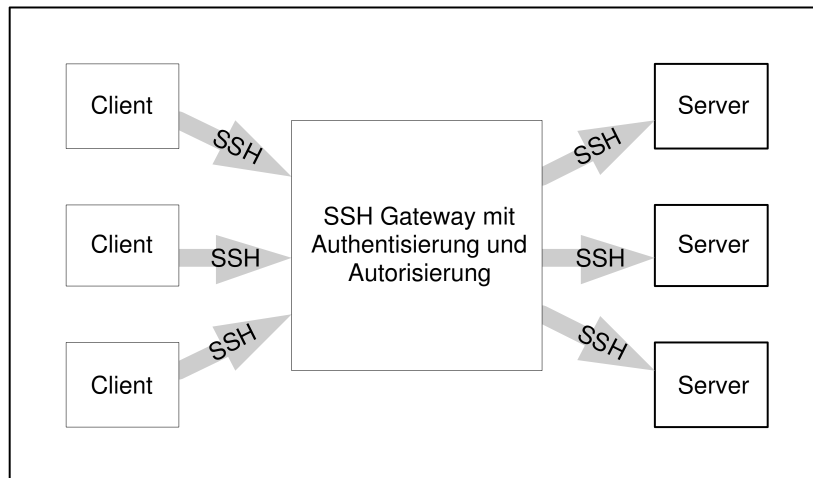

Abbildung 1: Idee des SSH Gateway

So ein SSH Gateway kann aber genausogut für den Remote Zugriff aus dem Internet auf interne Systeme benutzt werden.

Technisch gesehen gibt es folgende 5 Anforderungen:

Authentisierung *Das Gateway authentisiert jeden Benutzer.* Aus sicherheitstechnischen Überlegungen sind Passwörter hier nicht empfehlenswert. Aus praktischen Überlegungen (wenn mehrere SSH Gateways kaskadiert werden sollen) sind SecurID oder ähnliche Verfahren auch nicht akzeptabel.

Übrig bleibt also nur das Public Key Verfahren, optional auch mit Chipkarte.

Autorisierung *Das Gateway autorisiert den Zugriff des Benutzers auf den Server.* Das Gateway entscheidet, welcher Benutzer welchen Server ansprechen darf. Die Server selber führen zwar auch eine eigene Authentisierung durch, das Gateway entscheidet jedoch, ob diese überhaupt angesprochen werden.

Vertraulichkeit *Der Inhalt der Kommunikation zwischen Client und Server bleibt auch gegenüber dem Gateway gewahrt.* In unserem Szenario war dies notwendig. Es kann andere Fälle geben, in denen für Audits auch der Inhalt der Kommunikation protokolliert werden muss.

Auditing *Das Gateway protokolliert, wer, wann, auf welches System zugegriffen hat.* Gerade in sicherheitskritischen Bereichen kann dies für die Revision wichtig sein.

Transparenz *Der Zugriff soll möglichst transparent für die Benutzer erfolgen.* Unter anderem sollen auch auf SSH aufsetzende Dienste wie SCP und SFTP wie gewohnt funktionieren.

Ganz besonders möchte ich hier auf die Forderungen „Autorisierung“, „Vertraulichkeit“ und „Transparenz“ hinweisen. Diese sind bei naiven Implementationen in der Regel nicht erfüllt.

3 Eigenschaften von OpenSSH

Zur Realisierung des oben beschriebenen Gateways brauchen wir zwei weniger bekannte Funktionen von OpenSSH. Diese werden im folgenden erläutert.

3.1 Forced Commands

Normalerweise bekommt ein Benutzer nach dem Anmelden via SSH eine Login Shell zugewiesen und kann mit dieser alle Kommandos auf dem System ausführen. In bestimmten Fällen möchte man dies jedoch einschränken. Klassische Beispiele hierfür sind ein Backup über das Netzwerk oder ein remote angestossenes Intrusion Detection System. In beiden Fällen muss der Prozess auf dem Server als *root* laufen, hat also auf einem klassischen Unix alle Rechte auf dem System. Andererseits sind dies aber Aufgaben, die ohne Benutzerinteraktion automatisch via Cronjob ausgeführt werden sollen. Eine starke Authentisierung beim Zugriff auf das System ist also nur bedingt gegeben. Entweder ist das Passwort auf dem Client in einem Script hinterlegt oder der private Schlüssel des Benutzers ist nicht mit einer Passphrase geschützt.

Die Lösung des Dilemmas besteht darin, den Zugriff per SSH mit einem ungeschützten privaten Schlüssel zu erlauben, aber die durch ein Login mit diesem Schlüssel auf dem Server ausführbaren Aktionen auf das unbedingt notwendige zu beschränken.

Dazu dienen bei SSH die *Forced Commands*. Zu jedem Public Key auf dem Server kann in der Datei `~/.ssh/authorized_keys` der Name eines Unix Befehls hinterlegt werden, der bei einem Login mit diesem Public Key anstelle einer Shell ausgeführt wird.

Das sieht dann wie folgt aus:

```
# cat /root/.ssh/authorized_keys
command="/root/remote-ssh-command" ssh-dss AAAAB3NzaC1kc3MAAAEB
JAa8lzTplaMKm5MrWuuoTmqqhNT7Y0/tMokye9fAGJ6aLNdwC4Ybdi75f7xX8U0
..... F5Z2SeMAdQKXaBgigz9A== Automatic Network Administration
```

Bei jedem Login mit diesem Key wird das Shell-Script `/root/remote-ssh-command` aufgerufen. Damit nicht für jeden möglichen Befehl ein eigener Key generiert werden muss, erlaubt SSH den Zugriff auf das ursprünglich aufgerufene Kommando. Dieses wird in der Umgebungsvariablen `SSH_ORIGINAL_COMMAND` hinterlegt. Ein mögliches `remote-ssh-command` Script sieht wie folgt aus:

```
#!/bin/ksh -p

PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin
export PATH

LOGFILE=/var/log/$(basename $0)

log()
{
    echo "$(date +%Y%m%d%H%M%S)" $@ >> "${LOGFILE}"
}

log command "${SSH_ORIGINAL_COMMAND}"

case "${SSH_ORIGINAL_COMMAND}" in
    "test")
        echo test ok
        ;;
    "mrtg-interface-hme0")
        /usr/local/mrtg/bin/mrtg-interface hme0
        ;;
    *)
        log ERROR: invalid command "${SSH_ORIGINAL_COMMAND}"
        exit 1
        ;;
esac
exit 0
```

Wenn jetzt auf dem Client

```
$ ssh server mrtg-interface-hme0
```

aufgerufen wird, dann wird auf dem Server das Kommando

```
/usr/local/mrtg/bin/mrtg-interface hme0
```

abgearbeitet und die Ausgabe an den Client geschickt.

Besonders interessant ist hierbei, dass der Server nicht das vom Client mitgeschickte Kommando ausführen muss, sondern selber anhand der Eingaben ein eigenes Kommando generieren kann. Dies wird für die Implementation des Gateways von Nutzen sein.

3.2 Client-seitige Konfiguration und ProxyCommand

Eine der Stärken von OpenSSH gegenüber anderen SSH Implementationen sind seine äusserst flexiblen Konfigurationsmöglichkeiten auf Client Seite. Der Benutzer kann und sollte alle Optionen in der Datei `.ssh/config` hinterlegen.

Besonders hervorzuheben ist hierbei die Option `ProxyCommand`. Normalerweise stellt der SSH Client beim Aufruf

```
$ ssh hostname
```

eine TCP Verbindung zum Rechner `hostname` auf Port 22 her. Über diese TCP Verbindung wird dann das SSH Protokoll abgewickelt.

Im Gegensatz zu vielen anderen Netzwerk Kommandos der Unix Welt haben die OpenSSH Entwickler erkannt, dass es manchmal sinvoll sein kann, die Verbindung zum Server nicht über eine direkte TCP Verbindung herzustellen. Ein viel genutzter Anwendungsfall ist das Tunneln einer SSH Verbindung über einen HTTP Proxy, auch wenn dies oftmals ein Verstoss gegen die lokale Securitypolicy darstellt.

OpenSSH realisiert dies mit der Option `ProxyCommand`. Wenn diese gesetzt ist, wird von OpenSSH keine TCP Verbindung aufgebaut, sondern es wird das angegebene Kommando ausgeführt. Dieses ist dafür verantwortlich, eine Verbindung zu einem SSH Server aufzubauen und diese mittels Standardeingabe und Standardausgabe dem SSH Client zur Verfügung zu stellen. Ein triviales Beispiel:

```
$ cat .ssh/config
...
Host    gateway
ProxyCommand    nc gateway 22
....
```

Hier wird bei einem Aufruf von

```
$ ssh gateway
```

das Kommando

```
nc gateway 22
```

aufgerufen und darüber dann das SSH Protokoll abgewickelt. Das Programm `nc` [1] verbindet hierbei transparent seine Standard Ein-/Ausgabe mit dem Port tcp/22 auf dem Gateway. In diesem Beispiel bewirkt das Kommando

```
nc gateway 22
```

genau dass, was OpenSSH ohne die Option `ProxyCommand` getan hätte, es wird eine TCP Verbindung zum Server auf Port 22 aufgebaut. Allerdings kann man damit auch interessantere Dinge machen. Angenommen, wir können einen Rechner `monitoring` nicht direkt per TCP erreichen, sondern nur über ein Gateway in einer DMZ.

Wir könnten dies zwar mit

```
$ ssh gateway ssh monitoring
```

realisieren, haben aber hier den Nachteil, dass auf dem Gateway die Kommunikation im Klartext vorliegt (Abbildung 2). Auch ist dies nicht transparent für die Benutzung von Diensten wie SCP und SFTP.

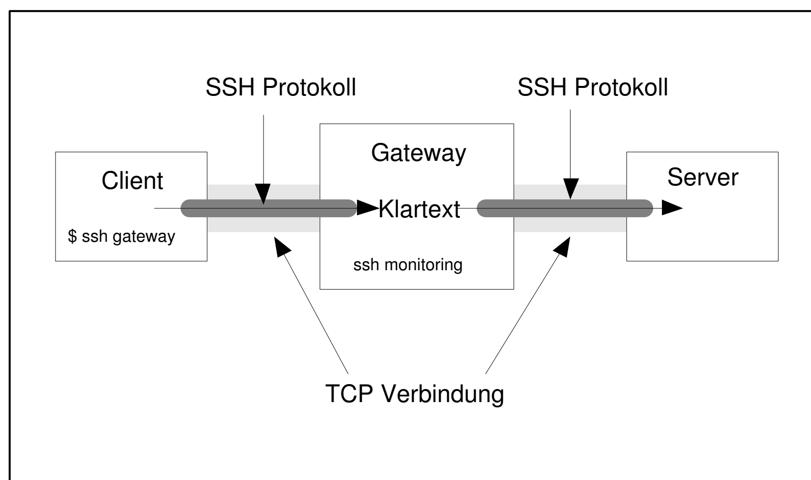

Abbildung 2: Ein naives Gateway

Um die Vertraulichkeit der Kommunikation und die Transparenz für auf SSH aufsetzende Dienste zu gewährleisten, muss ein End-to-End Tunnel aufgebaut werden. Dies kann wie folgt erreicht werden:

```
$ cat .ssh/config
...
Host  monitoring
ProxyCommand    ssh gateway nc monitoring 22
....
```

Hierbei wird bei einem Aufruf

```
$ ssh monitoring
```

zuerst eine SSH Verbindung zum Rechner `gateway` und auf diesem dann mittels

```
nc monitoring 22
```

die TCP Verbindung zum Rechner `monitoring` aufgebaut. Über diese Verbindung wird dann die eigentliche SSH Verbindung gefahren.

Auch hier gibt es zwei verschiedene TCP Verbindungen und zwei verschiedene SSH Tunnel. Die zweite SSH Verbindung besteht aber End-to-End zwischen Client und Server (Abbildung 3).

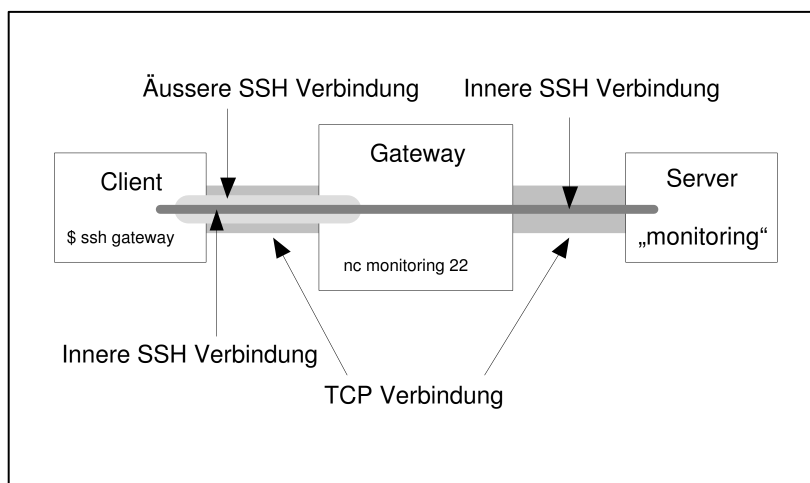

Abbildung 3: Gateway mit End-to-End Vertraulichkeit

4 Implementation des SSH Gateway

Nun können wir anfangen, unser SSH Gateway zu implementieren. Wir werden genau das im vorherigen Kapitel beschriebene Verfahren benutzen, dem Gateway aber die Entscheidung überlassen, ob und wohin die TCP Verbindung aufgebaut wird (Abbildung 4)

Da keiner der Benutzer direkt auf dem Gateway arbeiten soll, wäre es kontraproduktiv, ihnen Accounts auf dem lokalen System zu geben. Das gesamte SSH Gateway läuft unter einem speziellen Account, in diesem Fall `sshgw`. Unter diesem Account wird nie interaktiv gearbeitet.

Die Benutzer des SSH Gateways werden innerhalb dieses Accounts in der Datei `~/.ssh/authorized_keys` angelegt, für jeden Benutzer wird hier sein persönlicher Public Key eingetragen.

```
$ cat .ssh/authorized_keys
```

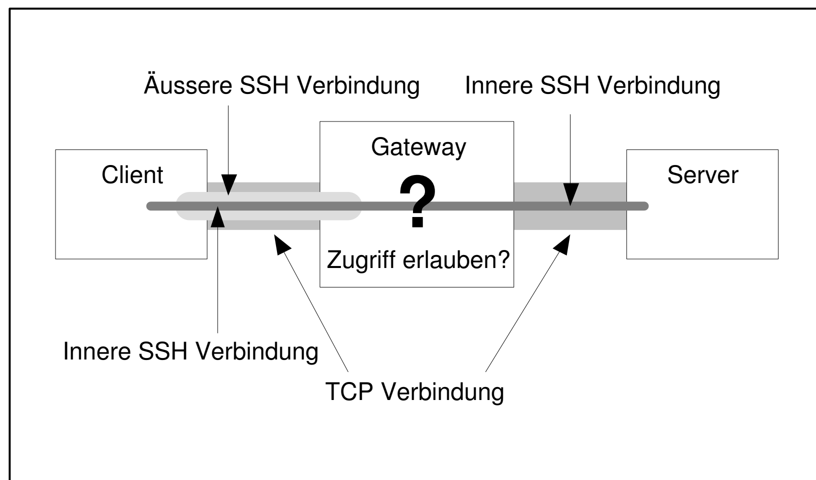

Abbildung 4: SSH Gateway

```
command="/opt/sshgw/bin/gw mirko.dziadzka" ssh-dss AAAAB3NzaCAAAE....
command="/opt/sshgw/bin/gw max.mustermann" ssh-dss AAAAB3NzaDAEK3....
...
```

Offensichtlich erhält jeder Public Key zwar dasselbe Kommando, diesem wird aber als Argument der reale Nutzername übergeben. Je nach benutztem Public Key wird das Script mit einem anderen Benutzernamen aufgerufen und kann diesen zur Autorisierung auswerten.

Wie erfahren wir nun, auf welchen Server sich der Benutzer eigentlich einloggen will? Ganz einfach, er muss es uns sagen. Um in unserem Beispiel zu bleiben, muss der Benutzer

```
$ ssh gateway monitoring
```

aufrufen, wenn er sich via `gateway` auf den Server `monitoring` einloggen will. Den Namen `monitoring` kann unser Script `gw` aus der Umgebungsvariable `SSH_ORIGINAL_COMMAND` erhalten. Damit haben wir alles zusammen, um eine erste Version des Gateway-Scriptes zu schreiben:

```
#!/bin/ksh

PATH=/bin:/usr/bin:/usr/local/bin
LOGFILE="/var/log/$(basename $0).log"

USER="$1"
SERVER="${SSH_ORIGINAL_COMMAND}"

#define authorisation check
access_ok()
```

```
{
        user="$1"
        server="$2"
        .... do database lookup here and return
        .... the result
}

# do sanity check for USER and SERVER
...

# check authorisation
if access_ok "$USER" "$SERVER" ; then
        # log this and connect to server
        log "START CONNECTION for $USER to $SERVER"
        nc "$SERVER" 22
        log "END CONNECTION for $USER to $SERVER"
else
        log "ACCESS DENIED for $USER on $SERVER"
        echo access denied
        exit 1
fi
exit 0
```

Dieses Script prüft, ob `USER` und `SERVER` gültige Werte haben, schaut dann mittels `access_ok` in einer Zugriffsdatenbank nach, ob der Benutzer `USER` sich auf dem Server `SERVER` einloggen darf. Wenn ja, wird vom Gateway eine TCP Verbindumg zum Server hergestellt, wenn nein, wird das Script beendet.

Wenn ein Benutzer auf Client Seite

```
$ ssh -l sshgw gateway monitoring
```

eingibt und `access_ok` diese Verbindung für diesen Benutzer freigibt, erhält der Benutzer eine Verbindung zum Port 22 auf dem Rechner `monitoring`

```
client$ ssh -l sshgw gateway monitoring
Enter passphrase for key:
SSH-2.0-OpenSSH_3.7.1p2
....
```

Diese Verbindung zum SSH Server kann nun in einer `ProxyCommand` Anweisung benutzt werden. Dafür tragen wir auf dem Client in der Datei `.ssh/config` folgendes ein:

```
$ cat .ssh/config

Host monitoring
ProxyCommand    ssh -l sshgw gateway monitoring
```

Jetzt führt ein

```
$ ssh monitoring
```

zum direkten Login auf dem Rechner `monitoring`. In diesem Fall werden zwei Authentisierungen verlangt, einmal am SSH Gateway und einmal am Server. Das zweimalige Eingeben der SSH Passphrase kann durch die Verwendung des SSH-Agent vermieden werden.

Damit ist das Ziel erreicht, ein SSH Gateway mit den in Abschnitt 2 genannten Anforderungen zu implementieren.

5 Was jetzt noch fehlt

Das oben beschriebene Script ist nicht mehr als ein Proof-of-Concept. Für eine produktive Implementation muss noch etwas Arbeit hineingesteckt werden. Einige der Punkte sind im folgenden aufgeführt.

5.1 Implementation der Autorisierung

Ein Prototyp der Funktion `access_ok` inklusive der Implementation von Benutzergruppen zur einfachen Administration befindet sich auf der Website [2] zum Artikel. Dieser muss je nach Umgebung an die eigenen Bedürfnisse angepasst werden.

5.2 Sicherheit

Unsere Lösung hat bisher leider Sicherheitslücken. Ein regulärer Benutzer des SSH Gateways kann die Autorisierung umgehen. Wie geht das? OpenSSH unterstützt Portforwarding. Was wir für die Benutzung von X11 Applikationen unbedingt brauchen und was an vielen anderen Stellen nützlich ist, führt hier zum Problem.

Ein Angreifer, der auf dem Gateway das Recht hat, eine Verbindung zu `server_a`, nicht aber zu `server_b` aufzubauen, kann diese Beschränkung umgehen:

Zuerst baut er eine Verbindung zum SSH Gateway auf und lässt sich zu `server_a` weitervermitteln. Gleichzeitig aktiviert er für diese Verbindung Portforwarding vom SSH Gateway zu `server_b`

```
$ ssh -L 2222:server_b:22 gateway server_a
...
```

und in einem zweiten Fenster benutzt er dann diese durch Portforwarding entstandene Verbindung zum Login auf `server_b`.

```
$ ssh localhost:2222
```

Glücklicherweise kann man das Portforwarding auf dem Gateway abstellen, zum Beispiel mit der Option `no-port-forwarding` in der Datei `~/.ssh/authorized_keys` oder global mit der Option `AllowTcpForwarding` in der Datei `sshd_config`. Für Details sei auf das SSH Manual verwiesen.

5.3 Hängende Sessions

Das hier benutzte Programm `nc` zum Aufbau einer TCP Verbindung hat sich im Zusammenhang mit OpenSSH unter Solaris als nicht sehr brauchbar erwiesen. Nach einiger Zeit gab es auf dem Gateway hunderte hängende SSH Sessions. Deswegen wurde `nc` durch ein eigenes Programm ersetzt, dass nach einer gewissen Zeit der Inaktivität, spätestens jedoch nach 24 Stunden, die Verbindung beendet.

6 Gelerntes

Eine ähnliche Lösung wie hier beschrieben befindet sich nun seit 2 Jahren im Dauereinsatz und läuft im täglichen Betrieb stabil. Das geht soweit, dass selbst minütlich laufende Cronjobs und Monitoring Zugriffe über das SSH Gateway laufen.

6.1 TCP Forwarder

Eine Implementation eines eigenen TCP-Forwarders anstelle von `nc` ist wichtig, sonst tendieren SSH Verbindungen dazu, hängen zu bleiben.

Auch lassen sich hier die mit der Zeit aufkommende Wünsche nach Timeouts, Beschränkung des Zugriffs für bestimmte Benutzer nur zu bestimmten Zeiten oder eine Beschränkung der gleichzeitig offenen Sessions eines Benutzers implementieren.

6.2 Nutzerfreundlichkeit

Es muss den Benutzern eine fertige `.ssh/config` zur Verfügung gestellt werden, sonst wird eine solche Lösung nicht akzeptiert. Für den Admin verwunderlich, aber leider wahr.

Was noch mehr erstaunte: viele Benutzer geben lieber die Passphrase für ihren privaten Schlüssel zweimal ein als SSH-Agent zu benutzen.

Literatur

[1] Das Programm `nc` ist unter http://netcat.sourceforge.net/ erhältlich.

[2] Dieser Artikel, eventuelle Korrekturen und Ergänzungen sowie die hier beschriebenen Scripte sind unter http://mirko.dziadzka.net/papers/ssh-gateway verfügbar.