



by Bernard Perrot

<bernard.perrot(at)univ-rennes1.fr>

About the author:

Bernard war seit 1982 als System- und Netzwerk-Techniker für das CNRS (French National Scientific Research Center) tätig. Er war verantwortlich für Aufgaben, die die Sicherheit von Computersystemen betreffen beim "Institut National de Physique Nucléaire et de Physique des Particules" (In2p3). Nun arbeitet er für das "Institute of Mathematical Research (IRMAR)" in der Abteilung "Physical and Mathematical Sciences (SPM)".

Translated to English by:

Guy Passemar

<g.passemar(at)free.fr>

Sichern Sie Ihre Verbindungen mit SSH



Abstract:

Dieser Artikel wurde zuerst in einer Spezialausgabe des französischen Linux-Magazins publiziert, deren Schwerpunkt auf Sicherheit lag. Der Editor, die Autoren und die Übersetzer haben LinuxFocus freundlicherweise gestattet, jeden Artikel aus dieser Spezialausgabe zu publizieren. Dementsprechend wird LinuxFocus diese Artikel herausbringen, sobald sie ins Englische übersetzt sind. Dank an alle Menschen, die an dieser Arbeit beteiligt sind. Dieser Abstrakt wird für jeden Artikel reproduziert, der aus der gleichen Quelle stammt.

Die Ambition dieses Artikels ist ein guter Ausblick auf SSH, warum es benutzt wird. Dies ist kein Tutorial oder ein Installations-Handbuch, eher eine Einführung in das Vokabular und die Eigenschaften von SSH. Die Verweise und Dokumentation in diesem Artikel geben Ihnen alle Einzelheiten der Implementation.

Wofür wird SSH benutzt?

Vor allem (und historisch) ist SSH (der Befehl *ssh*) eine sichere Version der Befehle *rsh* (und *rlogin*). SSH bedeutet "*Secure SHell*" wie *rsh* für "*Remote SHell*" steht. Wenn Ihnen *rsh* leichten Shell-Zugang zu einem entfernten Rechner geben kann, aber ohne Mechanismen für die Benutzerauthentifizierung, bietet Ihnen *ssh* den gleichen Dienst, jedoch mit hoher Sicherheit auf mehreren Ebenen.

Wenn wir kurz an einen Benutzer denken, der nicht mehr wissen (oder tun) möchte, könnten wir hier aufhören und sagen, dass der Administrator seinen Job getan und die Software installiert hat (was heute sehr leicht ist), dann brauchen Sie nur noch den Befehl *ssh* als Ersatz für *telnet*, *rsh* oder *rlogin*, und alles funktioniert wie

vorher, aber mit viel mehr Sicherheit.

Wenn Sie bisher

```
% rlogin serveur.org (oder telnet serveur.org)
```

benutzt haben, geben Sie jetzt ein:

```
% ssh serveur.org
```

und es ist schon viel besser!

Um diese kurze Einführung zu beenden, will ich nur feststellen, dass bis heute alle Sicherheits-Zwischenfälle, die durch die einfache Benutzung von SSH anstelle von *rsh (rlogin,telnet)* hätten vermieden werden können, nur eine Konsequenz der Nachlässigkeit des Opfers sind.

Was sind die Bedürfnisse?

Um mehr auf Einzelheiten einzugehen, hier sind einige der kritischsten und schwachen Aspekte interaktiver Verbindungen, die man gelöst haben möchte:

- Vor allem sollte man vermeiden, dass Passwörter übers Netz übertragen werden;
- Benutzung starker Authentifizierung in verbundenen Systemen, nicht nur basierend auf dem Namen oder der IP-Adresse, die leicht gefälscht werden können;
- Ausführung entfernter Befehle in vollständiger Sicherheit;
- Schutz von Datei-Übertragungen;
- sichere X11-Sitzungen, die sehr angreifbar sind.

Als Antwort auf diese Bedürfnisse gibt es einige Lösungen, die nicht wirklich zufriedenstellen:

- Die "*R-Befehle*": diese Befehle übertragen das Passwort nicht im Klartext, aber der benutzte "rhosts"-Mechanismus besitzt zahlreiche Sicherheitsprobleme;
- Die "*Einmal-Passwörter (One Time Password),(OTP)*" schützen nur die Authentifizierung, nicht aber die folgenden Daten. Obwohl dieses System aus Sicherheitssicht sehr attraktiv ist, unterliegt es Beschränkungen, die eine Akzeptanz durch die Benutzer schwierig machen. Konsequenterweise wird es meist in Umgebungen benutzt, die keinen ergonomischen Erwägungen unterliegen;
- telnet mit Verschlüsselung: diese Lösung ist nur anwendbar auf ... telnet. Außerdem umfasst dies nicht das X11-Protokoll, was eine oft benötigte Ergänzung ist.

Und dann gibt es SSH, eine gute Lösung für:

- Ersatz der *R-Befehle*: *ssh* anstelle von *rsh* und *rlogin*, *scp* mit *rcp*, *sftp* mit *ftp*;
- benutzt starke Authentifizierung, basiert auf Verschlüsselungs-Algorithmen unter Benutzung von öffentlichen Schlüsseln (sowohl für Systeme wie für Benutzer);
- ermöglicht die Umleitung des TCP-Datenstroms in einem Sitzungs-"Tunnel" und in X11-Sitzungen, hier kann dies automatisch erfolgen;
- Verschlüsselt den Tunnel und komprimiert ihn, wenn dies nötig ist und angefordert wird.

SSH Version 1 und SSH Version 2

Da nichts in dieser Welt perfekt ist, gibt es zwei inkompatible Versionen des SSH-Protokolls: die 1.x-Version (1.3 und 1.5) und die 2.0-Version. Der Wechsel von einer zur anderen Version ist für die Benutzerin nicht schmerzhaft, wenn sie die richtigen Clients und Server zur Verfügung hat, die mit der Version kompatibel sind.

Das SSH Version 1-Protokoll ist integriert, während SSH Version 2 das vorherige Protokoll in drei "Schichten" redefiniert hat.

1. SSH Transport Layer Protocol (SSH-TRANS)
2. SSH Authentication Protocol (SSH-AUTH)
3. SSH Connection Protocol (SSH-CONN)

Jede Protokollschicht ist speziell in einem Dokument (Draft) definiert, das von der IETF normiert wurde, gefolgt von einem vierten Dokument, das die Architektur beschreibt (SSH Protocol Architecture, SSH-ARCH). Man findet alle Details unter: <http://www.ietf.org/html.charters/secsh-charter.html>

Ohne zu sehr in die Einzelheiten zu gehen, dies finden Sie in SSHv2:

- die Transport-Ebene bietet Integrität, Verschlüsselung und Komprimierung, die Authentifizierung von Systemen
- die Authentifizierungs-Ebene bietet ... Authentifizierung (Passwort, Host-basiert, öffentlicher Schlüssel)
- die Verbindungs-Ebene, die den Tunnel verwaltet (Shell, SSH-agent, Port-Weiterleitung, Flußsteuerung).

Die wichtigsten technischen Unterschiede zwischen SSH Version 1 und 2 sind:

SSH Version 1	SSH Version 2
monolithisches (integriertes) Design	Trennung der Authentifizierungs-, Verbindungs- und Transport-Funktionen in Ebenen
Integrität mittels CRC32 (nicht sicher)	Integrität mittels HMAC (hash-Verschlüsselung)
ein und nur ein Kanal per Sitzung	unbeschränkte Kanal-Anzahl per Sitzung
Aushandlung nur mit einem symmetrischen Chiffre im Kanal, Sitzungs-Identifikation mit eindeutigem Schlüssel auf beiden Seiten	detailliertere Aushandlung (symmetrischer Chiffre, öffentliche Schlüssel, Komprimierung, ...) und ein separater Sitzungsschlüssel, Komprimierung und Integrität auf beiden Seiten
benutzt nur RSA als Algorithmus für den öffentlichen Schlüssel	RSA und DSA als Algorithmus für den öffentlichen Schlüssel
Sitzungs-Schlüssel vom Client übermittelt	Sitzungs-Schlüssel wird über das Diffie-Hellman-Protokoll ausgehandelt
Sitzungs-Schlüssel für die ganze Sitzung gültig	erneuerbare Sitzungs-Schlüssel

Handhabung eines Schlüsselbundes

Lassen Sie mich eben die SSH-Schlüsseltypen definieren:

- **Benutzer–Schlüssel:** Ein Schlüsselpaar, das aus einem öffentlichen und einem privaten Schlüssel besteht (beide asymmetrisch), Benutzer–definiert und permanent (auf Platte gespeichert). Dieser Schlüssel erlaubt Benutzer–Authentifizierung, wenn die Authentifizierungsmethode öffentlicher Schlüssel genutzt wird (weiter unten beschrieben).
- **Rechner–Schlüssel:** auch ein aus öffentlichem und privatem Schlüssel bestehendes Schlüsselpaar (beide asymmetrisch), aber zur Installations–/Konfigurations–Zeit vom Systemadministrator definiert und permanent (auf Platte gespeichert). Dieser Schlüssel erlaubt Authentifizierung zwischen Systemen.
- **Server–Schlüssel:** erneut ein aus öffentlichem und privatem Schlüssel bestehendes Schlüsselpaar (beide asymmetrisch), aber von einem Dämon beim Systemstart generiert und regelmäßig erneuert. Dieser Schlüssel verbleibt im Hauptspeicher, um den Austausch des Sitzungsschlüssels in SSHv1 zu sichern (In SSHv2 gibt es keinen Server–Schlüssel, da hier der Austausch über das Diffie–Hellman–Protokoll gesichert wird).
- **Sitzungs–Schlüssel:** dies ist ein geheimer Schlüssel, der vom symmetrischen Verschlüsselungs–Algorithmus zur Verschlüsselung des Kommunikations–Kanals genutzt wird. Wie in modernen kryptografischen Produkten üblich, ist dieser Schlüssel zufällig und vergänglich. SSHv1 benutzt einen Schlüssel per Sitzung auf beiden Seiten. SSHv2 benutzt zwei regenerierte Sitzungsschlüssel, einen auf jeder Seite.

Die Benutzerin fügt einen Passwort–Satz hinzu, der den privaten Schlüssel der erwähnten Schlüssel–Paare schützt. Dieser Schutz wird erreicht durch die Verschlüsselung der Datei mit dem privaten Schlüssel durch einen symmetrischen Algorithmus. Der zur Verschlüsselung der Datei genutzte geheime Schlüssel wird aus dem Passwort–Satz abgeleitet.

Authentifizierungs–Methoden

Es gibt verschiedene Benutzer–Identifikations–Methoden, die Wahl wird in Abhängigkeit von den Anforderungen der Sicherheits–Regeln getroffen. Die Authentifizierungsmethoden werden in der Konfigurationsdatei des Servers aktiviert (oder auch nicht). Hier sind die Hauptkategorien:

- **"telnet ähnlich":**

Dies ist die "traditionelle" Passwort–Methode: Beim Verbinden wird der Benutzer, nachdem er sich angemeldet hat, nach einem Passwort gefragt, das an den Server weitergeleitet wird, der es mit dem Passwort vergleicht, das dem Benutzer zugeordnet ist. Das verbleibende Problem (das eine astronomische Anzahl von Piratereien im Internet verursacht) ist, dass dieses Passwort *im Klartext* im Netz umläuft, und es daher von jedermann abgehört werden kann, der über einen einfachen "sniffer" verfügt. Hier bietet SSH die gleiche Oberfläche, (es ist eine einfache Methode für Anfänger, um von telnet zu SSH zu migrieren, weil man nichts Neues lernen muss ...), nichtsdestoweniger hat das SSH–Protokoll den Kanal verschlüsselt und das klar lesbare Passwort ist eingekapselt.

Eine noch sicherere Variante, konfigurierbar, wenn man die erforderlichen Programme auf dem Server hat, ist ein "One time password" (S/Key zum Beispiel): es ist sicherlich besser, offensichtlich sicherer, aber die ergonomischen Beschränkungen lassen eine Benutzung nur bei bestimmten Rechnern zu. Das System funktioniert wie folgt: Nachdem man seine Identität eingegeben hat, wird vom Server eine sog. "challenge (Herausforderung)" gesendet (anstatt nach einem statischen Passwort zu fragen), auf die die Benutzerin antworten muss. Da diese Herausforderung sich ständig ändert, muß die Antwort sich auch ändern. Konsequenterweise ist das Abhören der Antwort nicht wichtig, da sie nicht nochmals verwendet werden kann. Die erwähnte Beschränkung liegt in der Kodierung der Antwort, die berechnet werden muss (durch ein Token, Software auf dem Client usw), und die Eingabe ist ziemlich "kabalistisch" (im besten Fall sechs englische Einzel–Silben).

- **"rhosts ähnlich" (host–basierend):**

In diesem Fall ist die Identifikation ähnlich zu den R-Befehlen mit den Dateien wie `/etc/hosts` oder `~/.hosts`, die die Client-Rechner "zertifizieren". SSH trägt hier nur zu einer verbesserten Rechnererkennung bei mittels Benutzung privater "*shosts*"-Dateien. Von einem Sicherheits-Standpunkt her ist dies nicht ausreichend und ich rate davon ab, diese Methode allein zu verwenden.

- **Durch öffentliche Schlüssel**

Hier basiert das Authentifizierungssystem auf asymmetrischer Verschlüsselung (s. [den Artikel über Verschlüsselung](#) wegen weiterer Einzelheiten; kurz: SSHv1 benutzt RSA und SSHv2 führte DSA ein). Der öffentliche Schlüssel des Benutzers wird zuvor auf dem Server registriert und der private Schlüssel ist auf dem Client gespeichert. Mit diesem Authentifizierungssystem laufen keine Geheimnisse über das Netz und werden niemals zum Server gesendet.

Dies ist ein ausgezeichnetes System, aber seine Sicherheit ist immer noch begrenzt (in meiner Sicht), weil sie fast ausschließlich von der "Ernsthaftigkeit" des Benutzers abhängt (dieses Problem ist nicht SSH-spezifisch, aber ich glaube, dass dies DAS Hauptproblem von Systemen mit öffentlichen Schlüsseln ist, wie z. B. den heutzutage populären PKI-Systemen): um die Komprimierung des öffentlichen Schlüssels auf dem Client zu vermeiden, ist der Schlüssel normalerweise durch ein Passwort geschützt (zumeist als *Passwort-Satz* bezeichnet, um die Notwendigkeit zu betonen, mehr als ein *Wort* zu benutzen). Wenn die Benutzerin ihren privaten Schlüssel nicht sorgfältig (oder überhaupt nicht) schützt, kann jemand sehr leicht Zugriff auf alle ihre Ressourcen erhalten. Daher sage ich, dass die Sicherheit von der Ernsthaftigkeit des Benutzers und seinem Vertrauen abhängt, weil in diesem System der Server-Verwalter *keine* Möglichkeit hat, zu erfahren, ob der private Schlüssel geschützt ist oder nicht. Z. Z. verwaltet SSH noch keine Widerruflisten (wie viele nicht, selbst PKI nicht ...). Wenn z.B. ein privater Schlüssel ohne einen *Passwort-Satz* auf einem Heim-Computer gespeichert wird (es gibt keine schlechten Menschen zuhause, warum soll man sich da mit einem *Passwort-Satz* belasten ...?) und eines Tages muß dieses Gerät zur Reparatur-Abteilung eines wichtigen Händlers (lachen Sie nicht, das wird passieren, wenn elektronische Signaturen selbstverständlich werden), wird der Mechaniker (sein Sohn, seine Freunde) in der Lage sein, an die privaten Schlüssel auf jedem Computer zu gelangen, der über seinen Tisch läuft.

Die Konfiguration des SSH Authentifizierungs-Mechanismus für den Benutzer ist leicht unterschiedlich, je nachdem, ob man SSHv1, SSHv2 oder OpenSSH nutzt oder einen MacOStm- oder Windowstm- Client. Die grundlegenden Prinzipien und Schritte sind:

- ◆ wie man ein "asymmetrisches Schlüsselpaar" generiert (d.h. ein privates/öffentliches RSA/DSA-Schlüsselpaar), zumeist auf dem Client-System, (wenn mehrere Client-Systeme verbunden sind, generieren wir das Schlüsselpaar auf einem der Clients und replizieren die Schlüssel auf den anderen Clients). Einige der Windowstm- und MacOStm-Clients verfügen nicht über Dienstprogramme zur Schlüsselgenerierung; hier müssen Sie die Schlüsselgenerierung auf einem Unix-Rechner vornehmen und dann duplizieren. Das Schlüsselpaar wird im Benutzer-Verzeichnis `~/.ssh` gespeichert.
- ◆ Kopieren des öffentlichen Schlüssels auf einen Server, der für die Authentifizierung benutzt wird. Dies geschieht, indem einer Datei auf dem Server eine Zeile hinzugefügt wird, die mit dem generierten Schlüsselpaar aus dem `~/.ssh`-Verzeichnis der Benutzerin korrespondiert. (Der Name der Datei ist abhängig von der SSH-Version, entweder `authorized_keys` oder `authorization`).
- ◆ Und das ist alles ... wenn die Konfiguration weitere Authentifizierung auf der Serverebene verlangt, wird der Client zur Verbindungszeit nach einem "*Passwort-Satz*" fragen.

Außerdem ist es nützlich, zumindest zwei weitere Elemente zu kennen, die die Authentifizierung betreffen:

- **ssh-agent**

Einer der Gründe, warum man seinen privaten Schlüssel nicht schützt, ist die damit verbundene Belästigung, dass man den Schlüssel bei jeder interaktiven Verbindung eingeben muss und man den Schlüssel nicht in Hintergrund-Skripten benutzen kann. Es existiert eine Abhilfe, der *SSH agent*. Es ist ein Dienstprogramm (*ssh-agent*), das Ihnen nach der Aktivierung hilft, eine dreifache Identifizierung (Benutzername/Rechnername/Passwort-Satz) zu speichern und diese Identifizierungsmerkmale an Ihrer Stelle abzusetzen, wenn dies beim Verbindungsaufbau erforderlich ist. Auf der Clientseite wird nur einmal nach dem Passwort gefragt, so dass man von SSO (*Single Sign On*) sprechen könnte.

Nun sind Sie informiert, niemand zwingt Sie, Ihren privaten Schlüssel zu schützen, aber wenn Sie das nicht tun, dann ist das Fahrlässigkeit und Sie sind verantwortlich für die Konsequenzen.

- **"verbose"-Modus**

Es kann sein, dass die Verbindung aus Gründen nicht möglich ist, die der Benutzerin unbekannt sind: fügen Sie einfach die Option "-v" (steht für "verbose", geschwätzig) dem *ssh*-Befehl hinzu. Damit erhalten Sie während der Verbindung zahlreiche detaillierte Nachrichten auf dem Bildschirm, die Ihnen genug Informationen geben, um den Grund für die Verbindungsverweigerung zu bestimmen.

Die Verschlüsselungs-Algorithmen

Hier muss man unterscheiden zwischen denen für die Verschlüsselung von Kommunikations-Kanälen (Verschlüsselung mit geheimen Schlüsseln) und jenen, die für die Authentifizierung benutzt werden (Verschlüsselung mit öffentlichen Schlüsseln).

Für die Authentifizierung können wir in der Version 2 des Protokolls zwischen RSA und DSA wählen, für Version 1 steht nur RSA zur Verfügung (daher keine Wahlmöglichkeit ...). DSA wurde aus historischen Gründen gewählt, weil RSA in einigen Ländern *patentiert* war. Seit dem Ende des Sommers 2000 ist RSA frei von Rechten, so dass diese Beschränkung entfallen ist. Ich habe wirklich keine Vorliebe der guten oder schlechten Wahl (nur das DSA ein "reines" NSA-Produkt ist).

Für die symmetrische Verschlüsselung gibt es schon fast zu viel Auswahl ... Das Protokoll schreibt einen gemeinsamen Algorithmus vor, der in allen Implementationen vorhanden sein muss: *triple-DES mit drei Schlüsseln*. Daher wird er benutzt, wenn die Verhandlung zwischen Client und Server hinsichtlich der anderen Algorithmen scheitert. Wenn Sie können, versuchen Sie einen der anderen Algorithmen auszuhandeln, da 3DES nun mit die geringste Performanz aufweist. Nichtsdestoweniger werden wir die exotischen oder alten (arc4, DES, RC4, ...) beiseite lassen und uns auf folgende beschränken:

- IDEA: bessere Performanz als 3DES, aber unter bestimmten Konditionen nicht vollständig lizenzfrei (es war oft der Standard-Algorithmus in Unix-Versionen);
- Blowfish: sehr schnell, wahrscheinlich sicher, aber der Algorithmus muss sich im Lauf der Zeit noch beweisen;
- AES: der neue Standard (ersetzt DES), wenn er auf beiden Seiten verfügbar ist, nutzen Sie ihn, dafür wurde er geschaffen.

Persönlich frage ich mich nach dem Interesse, so viele Algorithmen vorzuschlagen; obwohl das Protokoll die Möglichkeit erlaubt, einen "privaten" auszuhandeln (z. B. für eine bestimmte Benutzergruppe), dies scheint für mich wesentlich, aber für den normalen Gebrauch denke ich, dass AES im Laufe der Zeit zum Standard

werden wird. Falls AES komprimiert wird, werden die Sicherheitsprobleme größer sein als die von SSH verursachten ...

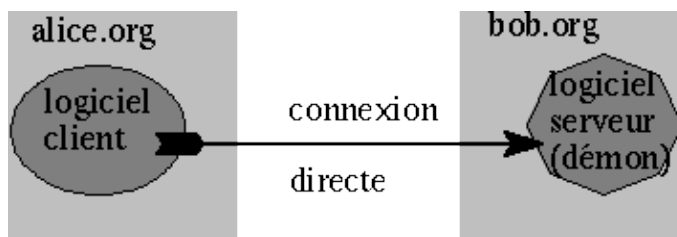
Port-Weiterleitung, Tunnelung

SSH ermöglicht Redirektion (*Weiterleitung*) eines jeden TCP-Datenstroms durch einen "Tunnel" in einer SSH-Sitzung. Das bedeutet, dass der Datenfluss einer Anwendung, statt direkt über die Client- und Serverports zu gehen, "eingeschlossen" wird in einem "Tunnel", der zur Verbindungs-/Sitzungszeit erstellt wird (*schauen Sie sich das folgende Diagramm an*).

Für das X11-Protokoll geschieht dies ohne spezielle Eingriffe (des Benutzers), mit transparenter Behandlung von *Displays* und erlaubt kontinuierliche Weiterleitung, falls es über mehrere Stationen geht.

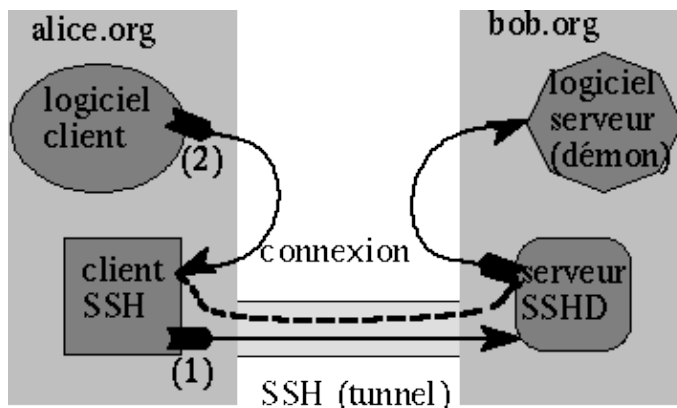
Für andere Datenströme gibt es Befehlszeilen-Optionen für jede Seite:

- Direkte Verbindung zwischen Client und Server



(Beispiel: `user@alice% telnet bob.org`)

- Lokale Port-Weiterleitung (Client) zu einem entfernten Port (Server)

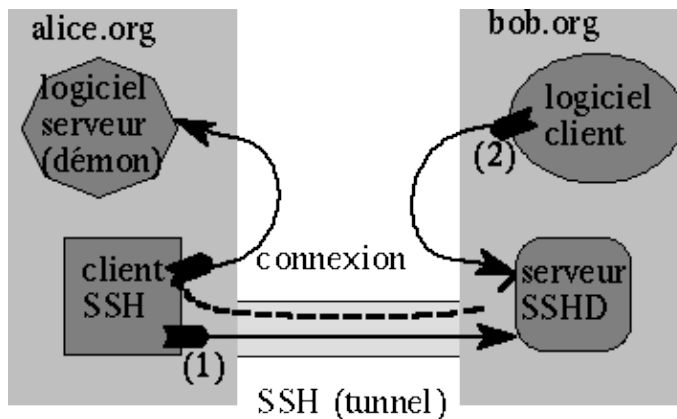


example: `user@alice% ssh -L 1234:bob.org:143 bob.org`

Dieses System ermöglicht Zugriff von "alice.org" auf den imap-Server bei "bob.org", Verbindungen von außerhalb auf das lokale Netzwerk werden abgewiesen. Sie werden nur über die *localhost*-Adresse zugelassen, Port 1234, von dem imap-Client, der auf "alice.org" ausgeführt wird.

- ◆ (1) die Benutzerin auf "alice.org" öffnet (Verbindung) den SSH-Tunnel
- ◆ (2) die Benutzerin auf "alice.org" konfiguriert den lokalen imap-Client für Zugriff auf den lokalen imap-Server auf *localhost* Port 1234

- Weiterleitung eines entfernten Ports (Client) auf einen lokalen Port (Server)



Beispiel: `root@alice% ssh -R 1234:bob.org:143 bob.org`

Dies ist die gleiche Situation wie oben, aber der Port auf dem entfernten Rechner wird weitergeleitet. Nur root hat die Rechte, um diesen SSH-Befehl auszuführen, aber jeder Benutzer kann diesen weitergeleiteten Port/Tunnel nutzen.

Diese mächtige Fähigkeit führt dazu, das SSH manchmal als "ein Tunnel für die Armen" bezeichnet wird. Hier bezeichnet Armut diejenigen, die keine Administrator-Rechte auf dem Client haben. Nur in speziellen Fällen kann ein lokaler Port mit niedrigen Privilegien (Port > 1024) und ohne Superuser-Rechte ("root") weitergeleitet werden. Andererseits, wenn die Weiterleitung eines privilegierten lokalen Ports benötigt wird, muss dies entweder unter einer *root*-Kennung erfolgen oder der Client muss mit Superuser-Privilegien ("*suid*") installiert werden (tatsächlich erlaubt ein privilegierter lokaler Port die Redefinition eines Standard-Dienstes).

Wie mit IP, ist es recht einfach, etwas in etwas anderes einzupacken (und umgekehrt), es ist nicht nur möglich, althergebrachte TCP-Datenströme weiterzuleiten, sondern auch PPP-Verbindungen, so dass wir einen "realen" IP-Tunnel in IP erstellen können (der verschlüsselt und daher sicher ist). Die Methode überschreitet den Rahmen dieses Artikels, Sie können jedoch "[Linux VPN-HOWTO](#)" wegen Einzelheiten und Setup-Skripten lesen (Sie finden auch richtige VPN-Lösungen für Linux wie "*stunnel*", die Sie erwägen sollten, bevor Sie eine endgültige Entscheidung treffen).

Denken Sie daran, dass die erste Möglichkeit ist, *telnet*-Datenströme umzuleiten: Dies scheint total nutzlos, da SSH automatisch interaktive Verbindungen implementiert. Sie können jedoch beim Weiterleiten von *telnet*-Verbindungen Ihren bevorzugten Client anstelle des interaktiven SSH-Modus benutzen. Dies könnten Sie besonders schätzen in einer Windowstm- oder MacOStm-Umgebung, wo der SSH-Client nicht den Vorlieben des Benutzers entspricht. Zum Beispiel leidet der "terminal emulation"-Teil des "*Mindterm*"-Clients (Javas SSH-Client, verfügbar auf allen modernen Systemen) unter der mangelnden Performanz der Sprache JAVA: es kann vorteilhaft sein, den Client nur zum Eröffnen des SSH-Tunnels zu nutzen.

Auf die gleiche Weise können Sie einen entfernten Client wie "*xterm*" starten (zum Beispiel mittels automatischer X11-Weiterleitung in SSH), was uns die Benutzung von SSH auf X-Terminals erlaubt.

Beachten Sie, dass der Tunnel offen bleibt, solange Daten fließen, selbst wenn sie nicht vom Initiator kommen. Daher ist der "*sleep*"-Befehl sehr nützlich zum Öffnen eines neuen SSH-Tunnels, um eine neue TCP-Verbindung weiterzuleiten.


```
% ssh -n -f -L 2323:serveur.org:23 serveur.org sleep 60
```

```
% telnet localhost 2323
```

```
... welcome to serveur.org ...
```

Die erste Zeile öffnet den Tunnel, startet den Befehl "*sleep 60*" auf dem Server und leitet den lokalen Port 2323 auf den entfernten Port Nummer 23 (*telnet*) weiter. Die zweite startet einen *telnet*-Client auf dem lokalen Port 2323 und benutzt dann den (verschlüsselten) Tunnel, um sich mit dem *telnetd*-Dämon auf dem Server zu verbinden. Der "*sleep*"-Befehl wird nach einer Minute beendet (es bleibt nur eine Minute Zeit, um *telnet* zu starten), aber SSH wird den Tunnel erst schliessen, wenn der letzte Client beendet wurde.

Haupt-Distributionen: frei verfügbar

Wir müssen zwischen Clients und/oder Servern auf den verschiedenen Plattformen unterscheiden und Sie sollten wissen, dass SSH Version 1 und SSH Version 2 inkompatibel sind. Die Referenzen am Ende des Artikels helfen Ihnen, andere Implementationen zu finden, die nicht in der folgenden Tabelle enthalten sind, welche sich auf freie Produkte mit genügend stabilen Fähigkeiten beschränkt..

Produkt	Plattform	Protokoll	Verweis	Notizen
OpenSSH	Unix	Versionen 1 und 2	www.openssh.com	Einzelheiten unten
TTSSH	Windows tm	Version 1	www.zip.com.au/~roca/ttssh.html	
Putty	Windows tm	Version 1 und 2	www.chiark.greenend.org.uk/~sgtatham/putty	nur Beta
Tealnet	Windows tm	Version 1 und 2	telneat.lipetsk.ru	
SSH secure shell	Windows tm	Versionen 1 und 2	www.ssh.com	frei für nicht-kommerzielle Nutzung
NiftytelnetSSH	MacOS tm	Version 1	www.lysator.liu.se/~jonasw/freeware/niftyssh/	
MacSSH	MacOS tm	Version 2	www.macssh.com	
MindTerm	Java	Version 1	www.mindbright.se	V2 nun kommerziell

Beachten Sie, dass MindTerm sowohl eine unabhängige Java-Implementation (Sie benötigen nur eine Java-*runtime*-Umgebung) als auch ein *servlet* ist, das innerhalb eines kompatiblen und gut gestalteten Webbrowsers ausgeführt werden kann. Leider sind die letzten Versionen dieser exzellenten Distribution gerade kommerzielle Produkte geworden.

OpenSSH

Heute ist diese Distribution wahrscheinlich diejenige, die in einer Unix/Linux-Umgebung eingesetzt wird (kontinuierliche Unterstützung, gute Antwortzeiten, quell-offen und frei).

Die OpenSSH-Entwicklung begann mit der Original-Version (SSH 1.2.12) von Tatu Ylonen (die wirklich letzte freie) im OpenBSD2.6-Projekt (über OSSH). Nun wird OpenSSH von zwei Gruppen entwickelt, eine,

die nur für das OpenBSD-Projekt entwickelt, die andere passt den Code kontinuierlich an, um eine portable Version zu erstellen.

All dies hat einige Konsequenzen, insbesondere wurde der Code größer und größer, eine konstante Monster-Anpassung (ich merke, wie das "sendmail"-Syndrom am Horizont erscheint, und das ist nicht gut für eine Anwendung, die der Verschlüsselung gewidmet ist und welche besonders rigoros sein sollte).

Neben dem reinen und lesbaren Code stören mich zwei Punkte besonders:

- OpenSSH benutzt für seine Verschlüsselungsdienste die OpenSSL-Bibliothek, und diese Bibliothek wird generell dynamisch gelinkt. In unserem Fall der Implementation eines Verschlüsselungspaketes, das Eigenschaften wie einen guten Sicherheitsstandard und voll verlässliche Eigenschaften hat, lässt mir den vorhergehenden Ansatz als vollkommen falsch erscheinen. Natürlich entspricht eine Attacke auf die Bibliothek einer Attacke auf das Produkt. Anders als eine zerstörerische Attacke, die Verschlüsselungs-Charakteristik (Qualität) in OpenSSH ist die der Bibliothek, die unabhängig von OpenSSH existiert.
- OpenSSH benutzt für einige seiner empfindlichen Dienste (z. B. einen Zufallszahlen-Generator) die OpenBSD-Systemdienste. Nur in diesem Kontext mache ich die gleichen Bemerkungen zur externen Abhängigkeit wie in OpenSSL. Noch störender ist, dass die portable Version von OpenSSH, die auf anderen Plattformen arbeitet, die von OpenBSD benötigten Dienste an verschiedene Mechanismen auf den anderen Zielplattformen delegiert. Z. B., ob ein Zufallszahlen-Generator auf Ihrem System verfügbar ist oder nicht, werden wir diesen benutzen oder einen anderen internen. Konsequenterweise wird die effektive Entropie von OpenSSH abhängig von der Laufzeit-Plattform.

Nach meiner Meinung (und ich bin da nicht allein), sollte ein Mehr-Plattform-Verschlüsselungsprodukt ein bewiesenes, bestimmtes und konstantes Verhalten aufweisen, unabhängig von der Plattform und die speziellen Eigenschaften der Plattform und deren Entwicklung in Betracht ziehen (ausgleichen).

Nachdem wir dies gesagt haben, müssen wir zugeben, dass die Implementationen der Wettbewerber weder zahlreich noch besonders attraktiv sind. Ich glaube, dass es pragmatischer ist, zu erwägen, dass OpenSSH die schlimmste Implementation ist, wenn man alle anderen ausschließt ...! Es wäre ein nützliches Projekt für die Gemeinschaft, den Code neu zu gestalten und zu schreiben.

Schlechte Neuigkeiten ...

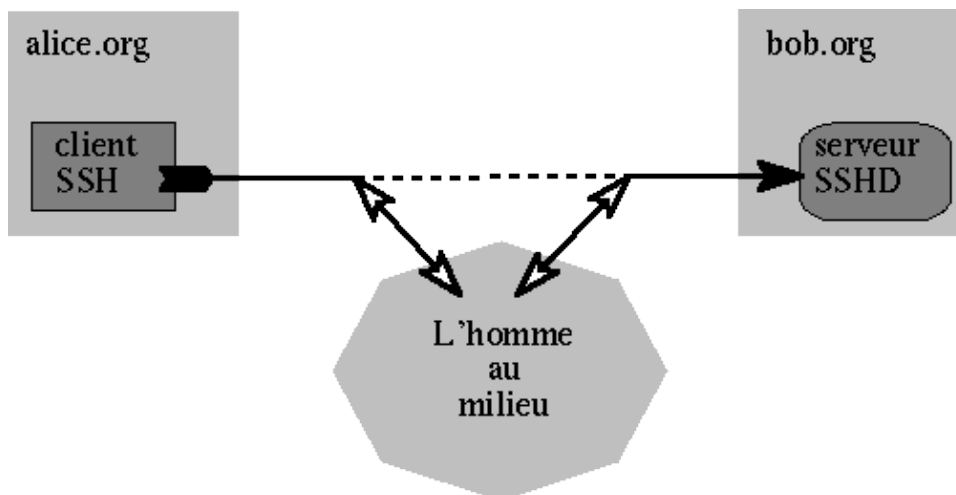
SSH bewirkt keine Wunder! Es tut das gut, wofür es geschaffen wurde, aber nach mehr dürfen Sie nicht fragen. Insbesondere wird es keine "berechtigten" Verbindungen verhindern: Wenn ein Konto kompromittiert ist, ist es für den Eindringling möglich, sich selbst mittels SSH mit Ihrem Computer zu verbinden, auch wenn es der einzige Weg ist, denn er kontrolliert die Authentifizierung. Sie können sich auf SSH nur dann vollständig verlassen, wenn es verbunden ist mit einer geeigneten und kohärenten Sicherheitspolitik: Wenn jemand nur ein Passwort benutzt und nicht überall SSH benutzt, ist das potentielle Risiko nur leicht vermindert. In dieser Situation kann SSH "fehlschlagen", da der Eindringling eine gesicherte verschlüsselte Verbindung mit Tunnelung nutzen kann, kann er wahrscheinlich alles tun, was er möchte, ohne die Möglichkeit einer effektiven Verfolgung durch Sie.

In dem gleichen Sinne muß man auch gut gemachte "rootkits" berücksichtigen, die generell einen SSH-Dämon enthalten, um eine diskrete Rückkehr in Ihr System zu ermöglichen, aber mit einigen Modifizierungen: er lauscht natürlich nicht auf Port 22, stoppt das Logging, ist nach einem gewöhnlichen Dämon benannt (z. B. *htpd*), und auch unsichtbar für einen "ps"-Befehl (der sicherlich auch durch das Rootkit verändert wurde).

Andererseits sollte man nicht zu sehr über die Gefahr besorgt sein, die ein SSH-Dämon darstellen kann, der es Eindringlingen ermöglicht, noch mehr verdeckt zu unternehmen: Sie wissen (so hoffe ich), dass es möglich ist, in IP alles in alles einzupacken, einschließlich "Fehldarstellung" der grundlegenden Protokolle über eine Firewall: HTML-Tunnelung, ICMP-Tunnelung, DNS-Tunnelung, Wenn Sie ein 100 % sicheres System haben möchten, müssen Sie Ihren Computer ausgeschaltet lassen ;-).g

SSH ist nicht frei von Sicherheits-"Schlupflöchern", die aus der Implementation abgeleitet wurden (viele wurden in der Vergangenheit korrigiert, da es kein perfektes Programm gibt), aber auch auf der Protokoll-Ebene. Diese "Schlupflöcher", obwohl sie als sehr alarmierend angekündigt wurden, betreffen häufig Schwachstellen, die schwierig auszunutzen sind, weil sie technisch sehr komplex sind: Man muß im Kopf behalten, dass die Sicherheitszwischenfälle, die durch die Benutzung von SSH vermieden werden, täglich auftreten, andererseits diejenigen, die auf Schwachstellen von SSH beruhen, oft theoretischer Natur sind. Es ist interessant, die Studie zu lesen, die sich mit "man in the middle"-Attacken befasst: <http://www.hsc.fr/ressources/presentations/mitm/index.html>. Nichtsdestoweniger ist es nötig, diese möglichen Verletzlichkeiten für "Hoch-Sicherheits"-Anwendungen {Finanzen, Militär, ...} in Rechnung zu stellen, wo die vom Cracker benutzten Mittel erheblich sein können, wenn er hochmotiviert wegen der Ziele und des Profits angreift.

- "Man in the middle"-Attache:



Der Angreifer fängt die Pakete von beiden Seiten ab und generiert seine Pakete, um jede Seite zu täuschen
(es sind verschiedene Szenarien möglich, bis dahin, die Verbindung zu einer Seite zu beenden, und mit der anderen Seite fortzufahren und vorzutäuschen, es handele sich um den normalen Partner).

Ich weise oft auf eine unverständliche Schwäche im Protokoll hin, die das Auffüllen betrifft (als verdeckter Kanal bekannt): In beiden Versionen 1 und 2 haben die Pakete eine Länge, die ein Mehrfaches von 64 Bit ist, und werden mit einer Zufallszahl aufgefüllt. Dies ist recht ungewöhnlich und bringt einen klassischen Fehler zum Vorschein, der in Verschlüsselungsprodukten sehr bekannt ist: ein "versteckter" oder ("unterbewusster") Kanal. Normalerweise füllt man mit einer verifizierten Sequenz auf, z. B. der Wert n für den Byterang n (*selbstbeschreibendes Auffüllen*). Da in SSH die Sequenz (per Definition) zufällig wird, kann sie nicht geprüft werden. Daher ist es möglich, dass eine der kommunizierenden Parteien die Kommunikation kompromittieren kann, die von einer dritten Partei abgehört wird. Man kann sich auch eine fehlerhafte Implementierung vorstellen, die beiden Parteien unbekannt ist (leicht zu realisieren bei Produkten, wo nur die Binärprogramme geliefert werden, wie es generell bei kommerziellen Produkten üblich ist). Dies kann leicht getan werden und in diesem Fall muss man nur Client oder Server "infizieren". Solch einen unglaublichen Fehler im Protokoll zu lassen, obwohl es universell bekannt ist, dass die Installation eines versteckten Kanals in einem

Verschlüsselungsprodukt DER klassische und einfachste Weg ist, die Kommunikation zu korrumpieren, scheint mir unglaublich. Es mag interessant sein, Bruce Schneiers Bemerkungen zu lesen, die die Implementation solcher Elemente in von Regierungs-Agenturen beeinflussten Produkten betreffen (<http://www.counterpane.com/crypto-gram-9902.html#backdoors>).

Ich beende diesen Punkt mit dem letzten Fehler, den ich während der Portierung von SSH nach SSF (französische Version von SSH) fand, der sich in der Kodierung der Unix-Versionen vor 1.2.25 findet. Die Konsequenz war, dass der Zufallsgenerator ... vorhersagbare ... Ergebnisse produzierte (diese Situation ist in einem kryptographischen Produkt bedauerlich; ich werde nicht in die Einzelheiten gehen, aber man kann die Kommunikation durch einfaches Abhören kompromittieren). Zu der Zeit hatte das SSH-Entwicklungsteam den Fehler korrigiert (es war nur eine Zeile zu ändern), allerdings ohne irgendeine Warnung zu versenden, noch nichtmals eine Erwähnung im "changelog" des Produktes ... Wenn man nicht will, dass dies bekannt wird, hätte man nicht anders gehandelt. Natürlich besteht keine Verbindung zu dem Link zum obigen Artikel.

Schlußfolgerung

Ich wiederhole, was ich in der Einführung geschrieben habe: SSH, kein Produkt bewirkt Wunder noch löst es alle Sicherheitsprobleme, aber es ermöglicht es, die schwächsten Aspekte der historischen interaktiven Verbindungsprogramme (telnet, rsh, ...) effizient zu behandeln.

Bibliographie, wesentliche Links

Die folgenden zwei Bücher behandeln SSH Version 1 und SSH Version 2:

- *SSH: the Secure Shell*
Daniel J. Barret & Richard E. Silverman
O'Reilly – ISBN 0-596-00011-1
- *Unix Secure Shell*
Anne Carasik
McGraw-Hill – ISBN 0-07-134933-2
- [openssh: http://www.openssh.com](http://www.openssh.com)

Und wenn Sie etwas Geld ausgeben möchten, können sie hier anfangen ...:

- <http://www.ssh.com>

Ausnutzung des verdeckten Kanals (verursacht durch das zufällige Auffüllen in SSHv1)

Hier ist eine Möglichkeit, den verdeckten (unterbewußten) Kanal auszunutzen, der durch das zufällige Auffüllen in SSHv1 (und v2) möglich ist. Ich lehne jede Verantwortung für Herzattacken ab, die die ganz Paranoiden treffen könnten.

Die SSHv1-Pakete haben folgende Struktur:

Offset (Bytes)	Name	Länge (Bytes)	Beschreibung
0	Größe	4	

			Paketgröße, Feldgröße ohne Auffüllung, daher: Größe = Länge(Typ)+Länge(Daten)+Länge(CRC)
4	Auffüllung	p = 1 bis 8	zufälliges Auffüllen : Größe so angepasst, dass der verschlüsselte Teil ein Vielfaches von acht ist
4+p	Typ	1	Paket-Typ
5+p	Daten	n (variable ≥ 0)	
5+p+n	Prüfsumme	4	CRC32

Nur ein Feld ist nicht verschlüsselt: die "Größe". Die Größe des verschlüsselten Teils ist immer ein Vielfaches von acht, angepasst durch "Auffüllen". Das Auffüllen wird immer vorgenommen, wenn die Länge der drei letzten Felder bereits ein Vielfaches von acht ist, wird das Aufgefüllte 8 Bytes lang sein ($5+p+n \text{ Rest } 0 \text{ modulo } 8$). Man betrachte die Verschlüsselungsfunktion C , symmetrisch, im CBC-Modus genutzt und die Entschlüsselungsfunktion C^{-1} . Um diese Demonstration zu vereinfachen, nehmen wir nur die Pakete mit 8-Byte-Auffüllung. Wenn eines der Pakete ankommt, nehmen wir einen Wert von $C^{-1}(M)$, 8 Bytes in diesem Fall, anstatt mit einer Zufallszahl aufzufüllen. Dies bedeutet die Entschlüsselung einer Nachricht M mit der Funktion C , die zur Verschlüsselung des Kanals benutzt wurde (die Tatsache, dass M "entschlüsselt" wird, ohne vorher verschlüsselt worden zu sein, hat aus einer streng mathematischen Betrachtungsweise keine Wichtigkeit, ich gehe hier nicht auf die Einzelheiten der praktischen Implementation ein). Als nächstes führen wir die normale Verarbeitung des Paketes durch, d.h. die Verschlüsselung von Blöcken mit 8 Bytes.

Das wird das Ergebnis sein:

Offset	Inhalt	Hinweise
0	Länge	4 nicht verschlüsselte Bytes
4	8 Bytes Auffüllung (verschlüsselt)	daher $C(C^{-1}(M))$
12... Ende	Typ, Daten, CRC	

Was ist hier erstaunlich? Der erste verschlüsselte Block enthält $C(C^{-1}(M))$. Da C eine symmetrische Verschlüsselungsfunktion ist, gilt $C(C^{-1}(M)) = M$. Dieser erste Block wird entschlüsselt in einem verschlüsselten Datenstrom gesendet! Das bedeutet nur, dass jemand, der Kenntnis von dieser Liste hat und die Kommunikation belauscht, auch weiss, wie er diese Information ausnutzen kann. Natürlich kann man annehmen, dass die Nachricht M selbst verschlüsselt ist (z.B. gegen einen öffentlichen Schlüssel, womit vermieden wird, ein Geheimnis in den pervertierten Code einzufügen), was immer noch die Entschlüsselung durch jemanden verhindert, der nicht informiert ist.

So benötigt es z.B. drei Pakete diesen Typs, um den triple-DES(168 bit)-Sitzungsschlüssel durchzureichen, wonach der Datenfluss-Sniffer die gesamte Kommunikation entschlüsseln kann. Wenn der Schlüssel gesendet wird, ist es nicht länger nötig, das pervertierte Auffüllen "vorzuentschlüsseln", es ist möglich, Auffüllungen beliebiger Größe zu nehmen, wenn man noch mehr Informationen hinzufügen möchte.

Die Benutzung dieses verdeckten Kanals ist *absolut nicht zu erkennen!* Man muss vorsichtig sein, jedes Element der Nachricht wie eben erläutert zu verschlüsseln, damit die Entropie des Blocks nicht die List verrät. Unentdeckbar, da das Auffüllen zufällig geschieht, was mögliche Validierungs-Tests eliminiert. Zufälliges Auffüllen sollte *niemals* in kryptographischen Produkten angewandt werden.

Was diesen Kanal noch gefährlicher macht als andere im Protokoll, wird eingeführt durch Nachrichten wie SSH_MSG_IGNORE, wo Sie es nutzen können, *ohne* den Verschlüsselungs-Schlüssel zu kennen.

Um die perversen Effekte des zufälligen Auffüllens zu vermeiden, muss man im Protokoll nur die Benutzung des deterministischen Auffüllens definieren, allgemein als "*selbstbeschreibendes Auffüllen*" bezeichnet, was bedeutet, dass das Byte an Offset n auch n enthält. Zufälliges Auffüllen ist in SSH v2 noch verblieben, es ist eine Möglichkeit, so denken Sie daran ...

Zum Schluss möchte ich sagen, wenn ich den verdeckten Kanal kritisiere, dann deshalb, weil ich möchte, dass ein Produkt wie SSH, das von sich hohe Sicherheit behauptet, auch wirklich ein Maximum an Sicherheit bietet. Nun können Sie sich vorstellen, dass in vielen kommerziellen Produkten potentielle Möglichkeiten zur Manipulation existieren: nur quelloffene Produkte bieten die Lösung der primären Anforderung, die Möglichkeit zur Prüfung des Codes (selbst wenn die Prüfung oft noch durchgeführt werden muss).

<p><u>Webpages maintained by the LinuxFocus Editor</u> <u>team</u> © Bernard Perrot "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org</p>	<p>Translation information: fr --> -- : Bernard Perrot <bernard.perrot(at)univ-rennes1.fr> fr --> en: Guy Passemard <g.passemard(at)free.fr> en --> de: Hermann-Josef Beckers <beckerst/at/lst-online.de></p>
---	--

2005-01-11, generated by lfparsr_pdf version 2.51