LinuxFocus article number 273
http://linuxfocus.org

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

Over de auteur:
Bernard is System en Netwerk Enigneer voor het CNRS (Frans Nationaal wetenschappelijk onderzoek centrum) sinds 1982. Hij heeft de leiding gehad over projecten met betrekking tot computer systeem beveiliging bij het "Institut National de Physique Nucléaire et de Physique des Particules" (In2p3). Tegenwoordig is hij werkzaam bij het Instute Of Mathematical Research (IRMAR) in het Natuurkundig en mathematische wetenschappen departement (SPM).

Vertaald naar het Nederlands door:
GH Snijders
<ghs(at)linuxfocus.org>

Beveilig je connecties met SSH

Kort:

Dit artikel werd voor het eerst gepubliceerd in een speciale editie van Linux Magazin Frankrijk, gefocust op beveiliging. De redacteur en de autheurs waren zo vriendelijk om LinuxFocus toe te staan om ieder artikel uit deze speciale uitgave te publiceren. Aansluitend, zal LinuxFocus deze presenteren zodra de vertaling naar het Engels gereed is. Met dank aan alle mensen die betrokken zijn bij dit werk. Deze abstract zal gereproduceerd worden voor artikelen van dezelfde oorsprong.

In dit artikel zullen we een goede blik werpen op SSH, en op het gebruik ervan. Dit is geen gids of installatie handleiding, maar eerder een introductie in het vocabulair en de mogelijkheden van SSH. De links en documentatie bij dit artikel zullen alle implementatie details bezorgen.

Waar wordt SSH voor gebruikt?

Allereerst (en historisch), SSH (het commando *ssh* is een veilige variant van de *rsh* en *rlogin*

commando’s. SSH staat voor "*Secure SHell*" zoals *rsh* staat voor "*Remote SHell*". Dus, zoals *rsh* je eenvoudig toegang kan geven tot shell op een andere machine, zonder mechanisme voor gebruikers authenticatie, levert *SSH* dezelfde mogelijkheid maar dan met een hoog niveau van beveiliging.

Als we erg kort door bocht zouden willen zijn voor een gebruiker die niet meer wil weten (of doen), zouden we hier kunnen stoppen en zeggen dat de beheerder z’n werk heeft gedaan en de software heeft geïnstalleerd (dit is tegenwoordig simpel), en dat alles wat je nodig hebt, het *SSH* commando is, om het *telnet*, *rsh* of *rlogin* commando’s te vervangen en dat alles hetzelfde werkt met betere beveiliging.

Dus, als je gebruik maakte van:

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

gebruik je nu:

```
% ssh serveur.org
```

en het is nu al een stuk beter!

Ter afsluiting van deze korte samenvatting, wil ik stellen dat vanaf vandaag, alle beveiligings incidenten die voorkomen konden worden door simpelweg SSH te gebruiken in plaats van *rsh* (*rlogin*, *telnet*) voornamelijk een consequentie zijn van nalatigheid van het slachtoffer.

Benodigdheden

Om meer informatie te geven, volgen hier een paar van de meest cruciale en fragiele aspecten van interactieve connecties die men graag opgelost ziet:

- Ten eerste, voorkom dat wachtwoorden over het net worden gestuurd;
- gebruik sterke authenticatie op verbonden systemen, niet alleen gebaseerd op de naam of IP adres, welke gespoofed kunnen worden;
- voer remote commando’s uit in volledige beveiliging.
- bescherm het transport van bestanden;
- beveilig X11 sessies, welke erg kwetsbaar zijn

Als reactie op deze eisen, bestaan er oplossingen die niet echt bevredigen:

- De "*R-commando’s*": deze commando’s versturen het password niet in platte tekst, maar gebruiken het "rhosts" mechanisme, welke onderwerp is van vele beveiligings problemen;
- Het "*One Time Password*, (OTP, eenmalig wachtwoord)": beschermt alleen de authenticatie, niet de data. Ondanks dat dit systeem uit het oogpunt van veiligheid heel aantrekkelijk is, heeft het eigenschappen die voor gebruikers lastig te accepteren zijn. Het wordt voornamelijk gebruikt in omgevingen waar ergonomie van onderschikt belang is;
- The "*One Time Password*
- telnet met encryptie: deze oplossing is alleen toepasbaar op... telnet. Buiten dat, het omvat niet het X11 protocol, vaak een vereiste toevoeging.

En er is SSH, een goede oplossing voor:

- het vervangen van de *R-commands*:*ssh* in plaats van *rsh* en *rlogin*, *scp* met *rcp*, *sftp* met *ftp*;
- Gebruikt sterke encryptie, gebaseerd op encryptie algoritmes voor publieke sleutels (PKI), zowel voor systemen als voor gebruikers;
- maakt het mogelijk om de TCP data stroom in een sessie te "tunnelen" en in X11 sessies, dit kan geautomatiseerd worden;
- de tunnel versleutelen, en, indien gewenst, te comprimeren.

SSH versie 1 en SSH versie 2

Niets is perfect in deze wereld, daarom bestaan er twee niet-compatibele versies van het SSH protocol: de 1.x versie (1.3 en 1.5) en de 2.0 versies. Om te switchen van de een naar de ander is geen probleem voor de gebruiker, mits met de juiste client en de juiste server bij de hand.

Het SSH versie 1 protocol is geïntegreerd, waar SSH versie 2 het vorige protocol heeft geherdefiniëerd in drie "lagen" (layers):

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

iedere laag is specifiek gedefiniëerd in een document (draft) genormaliseerd door de IETF, gevolgd door een vierde draft, welke de architectuur (SSH Protocol Architecture, SSH_ARCH) beschrijft. De details zijn te vinden op: http://www.ietf.org/html.charters/secsh-charter.html

Zonder al teveel op details in te gaan, hier is wat je kunt vinden in SSHv2:

- de transport verzorgt integriteit, encryptie, compressie en de authenticatie van systemen
- de authenticatie laag levert ... authenticatie (password, host-gebaseerd, public key)
- de connectie laag welke de tunnel beheerd (shell, SSH-agent, port forwarding, flow control).

Belangrijke technische verschillen tussen SSH versie 1 en 2 zijn:

SSH versie 1	SSH versie 2
monolitisch (geïntegreerd) ontwerp	scheiding van de authenticatie, connectie en transport functies in lagen
integriteit via CRC32 (niet erg veilig)	integriteit via HMAC (hash encryptie)
een, maar dan ook echt een kanaal per sessie	onbeperkt aantal kanalen per sessie
onderhandeling via symmetrische codering in het kanaal, sessie identificatie met een unieke sleutel aan beide kanten	gedetailleerde onderhandeling (symmetrische codering, publieke sleutels, compressie, ...), en een aparte sessie sleutel, compressie en integriteit aan beide kanten
alleen RSA voor het publieke-sleutel algorithme	RSA en DSA voor publieke-sleutel algorithme
sessie sleutel verstuurd door client	sessie sleutel overeengekomen door Diffie-Hellman protocol
sessie sleutel geldig gedurende de sessie	vernieuwbare sessie sleutel

Een aantal sleutels onderhouden

SSH sleutel typen, laat me ze kort definiëren:

- User key: een sleutelpaar samengesteld uit een publieke en een prive sleutel (beide assymetrisch), gebruiker gedefinieerd en permanent (opgeslagen op schijf). Deze sleutel staat de authenticatie van de gebruiker toe, als de public key authenticatie methode wordt gebruikt (verderop meer hierover).
- Host key: ook een sleutelpaar dat samengesteld is uit een publieke en een prive sleutel (beide assymetrisch), maar gegenereerd tijdens installatie/configuratie door de beheerder van de server, en permanent (opgeslagen op schijf). Deze sleutel staat de authenticatie tussen systemen toe
- Server key: weer een sleutelpaar dat is samengesteld uit een publieke en een prive sleutel (assymetrisch), maar gegenereerd door een daemon tijdens het starten en regelmatig vernieuwd. Deze sleutel blijft in het geheugen staan om de de uitwisseling van de sessie sleutel te beveiligen in SSHv1 (met SSHv2 is er geen server key, daar de uitwisseling wordt beveiligd met het Diffie-Hellman protocol).
- Session key:dit is een geheime sleutel die gebruikt wordt door het encryptie algorithme om het communicatie kanaal te versleutelen. Als altijd in moderne cryptografische producten, is deze sleutel random en vergankelijk. SSHv1 kent een sleutel per sessie, aan beide kanten. SSHv2 kent 2 gegenereerde sessie sleutels, een aan iedere kant.

De user voegt een wachtzin (pass phrase) toe, welke de prive sleutel beveiligt, als onderdeel van bovengenoemde sleutels. Deze beveiliging wordt gegarandeerd door het bestand met de prive sleutel te versleutelen met een symmetrisch algortihme. De geheime sleutel die gebruikt wordt om het bestand te versleutelen, komt voort uit de pass phrase.

Authenticatie methodes

Er bestaan verschillende methodes voor de identificatie van gebruikers. De keus wordt gemaakt aan de hand van de eisen, zoals beschreven in de beveilgings policies. De geauthoriseerde methoden worden al dan niet geactiveerd in het configuratie bestand van de server. Hier zijn de principe categorieën:

- **"telnet-achtig":**

  Dit is de "traditionele" wachtwoord methode: als er verbinding wordt gemaakt,wordt, na het vastellen van de identiteit, de gebruiker uitgenodigd het wachtwoord op te geven dat bij de desbetreffende gebruiker hoort, welke naar de server wordt verstuurd. Het resterende probleem (dat voor een astronomisch aantal problemen zorgt op het Internet) is dat het wachtwoord in *klare tekst* wordt rondgestuurd over het net, zodat het door iedereen onderschept kan worden met een eenvoudige "*sniffer*". Hier heeft SSH hetzelfde voorkomen (het is een eenvoudige methode voor de gebruiker om over te stappen van telnet naar SSH, daar er niets nieuws geleerd hoeft te worden ...), maar heeft het SSH protocol het kanaal versleuteld en wordt het wachtwoord-in-platte tekst hierin opgenomen.

  Een variant die zelfs nog veiliger is, en configureerbaar mits men de juiste spullen op de server heeft is de "*One Time Password*" methode (S/Key bijvoorbeeld): het is zeker beter, duidelijk veiliger, maar de ergonomische nadelen maken het alleen geschikt voor bepaalde sites. Dit systeem werkt als volgt: nadat de identiteit is opgegeven, wordt de gebruiker niet om een wachtwoord gevraagd, maar stuurd de server een "*challenge*" waar de gebruiker op dient te reageren. De challenge is altijd anders, en dus moet het antwoord mee veranderen. Dit heeft als gevolg dat het onderscheppen van het antwoord niet uitmaakt, daar het niet opnieuw gebruikt kan worden. De beperking, zoals genoemd, zit in het coderen van het antwoord, welke uitgerekend dient te worden (door een token, software op de client, etc.), het idee is op zich is "intrigerend".

- **"rhosts-achtig"** (*host gebaseerd*:

  In dit geval gebeurd de identificatie, net als met de R-commando's, met behulp van bestanden als /etc/rhosts of ~/rhosts, welke client sites "certificeren". SSH draagt alleen maar bij aan betere host-identificatie, en aan het gebruik van eigen "*shosts* bestanden. Vanuit het oogpunt van beveiliging is dit niet voldoende, en raad ik af om dit als enige methode te gebruiken.

- **Met publieke sleutels**

  Hier zal het authenticatie systeem gebaseerd zijn op assymetrische encryptie (zie ook het artikel over encryptie voor meer details; in het kort komt het er op neer dat SSHv1 RSA gebruikt en SSHv2 DSA mogelijk maakt). De publieke sleutel van de gebruiker wordt van te voren opgeslagen op de server en de prive sleutel wordt op de client opgeslagen. Met dit authenticatie schema reizen er geen geheimen over het Internet, en worden nooit naar de server verstuurd.

  Dit is een schitterend systeem, maar toch is de veiligheid ervan beperkt (vanuit mijn oogpunt) doordat het bijna exclusief afhankelijk isvan de serieusheid van de gebruiker (dit probleem is niet exclusief van SSH, ik geloof dat het HET probleem is met publieke sleutel systemen, zoals het vandaag de dag populaire PKI): dus, om compressie van de publieke sleutel op de client systemen te voorkomen, wordt de sleutel normaal gesproken beschermd door een wachtwoord (vaak ook een *pass phrase*, wachtzin genoemd om het belang van meer dan 1 *woord* aan te geven). Als de gebruiker zijn prive sleutel niet voorzichtig (of helemaal niet) beschermt kunnen anderen

gemakkelijk toegang krijgen tot al zijn bronnen. Dat is waarom ik beweer dat de veiligheid afhankelijk is van hoe serieus de gebruiker is, en zijn mate van vertrouwen, daar in dit geval de systeembeheerder *geen enkele* manier heeft om te achterhalen of de prive sleutel veilig is, of niet. Op het moment biedt SSH geen revocation (herroepings) certificaten (velen doen dit niet, zelfs PKI niet...). Bijvoorbeeld, als de prive sleutel wordt opgeslagen zonder passphrase op een thuis computer (er zijn geen 'slechten' thuis, dus waarom zou je jezelf lastig vallen met een pass phrase...?) en op een dag gaat hij ter reparatie naar een belangrijke dealer (lach niet, dit is wat er gebeurd als electronische signaturen algemeen gaan worden...), zal de reparateur (zijn zoon, zijn vrienden) in staan zijn op de prive sleutels van iedere computer die op zijn tafel verschijnt, af te halen.

De configuratie van het SSH authenticatie mechanisme voor de gebruiker is lichtelijk anders wanneer men SSHv1, SSHv2 of OpenSSH gebruikt, ook voor een MacOS[tm] of Windows [tm] client. De basis principes en stappen om te onthouden zijn:

- Hoe een "assymetrisch sleutelpaar" te genereren (het prive/publieke RSA of DSA sleutelpaar), meestal op het client systeem, (als er verschillende clients verbinding maken genereren we het sleutelpaar op een van de clients en repliceren we deze naar de andere). Sommige van de Windows [tm] en MacOS [tm] clients kennen geen service programma's om sleutelparen te genereren, het genereren dient dan op een Unix machine te gebeuren, alvorens de sleutels te repliceren. Het sleutelpaar wordt opgeslagen in de `~/.ssh`user directory.
- Het kopieëren van een publieke sleutel naar servers die gebruikt zullen worden voor de authenticatie. Dit kan door een regel toe te voegen, welke overeenkomt met het gegenereerde sleutelpaar in de gebruikers ~/.ssh directory, aan een server bestand in dezelfde directory. (De naam is afhankelijk van de SSH versie, ofwel`authorized_keys` of `authorization`).
- En dat is alles... als de configuratie verder authenticatie vereist op server niveau, zal de client gevraagd worden om een "*pass phrase*" tijdens het maken van de verbinding.

Het kan ook handig zijn om te weten dat er tenminste nog twee elementen bestaan met betrekking op authenticatie:

- **ssh-agent**

  Een van de redenen waarom iemand zijn prive sleutel niet beschermd is de irritatie van het opgeven van de sleutel voor iedere interactieve verbinding plus het feit dat de sleutel niet gebruikt kan worden als de verbinding wordt opgezet door een script dat in de achtergrond loopt. Er bestaat een remedie in de vorm van *SSH agent*. Dit is een service programma, dat, eens geactiveerd door jou, behulpzaam kan zijn door een drie-voudige identifier (gebruikersnaam/hostnaam/pass phrase) op te slaan, en deze tijdens het opzetten van de verbinding kan opgeven in jouw plaats. Aan de kant van de client wordt eenmalig om het wachtwoord gevraagd, en dus zou je het SSO (*Single Sign On* (eenmalige login) kunnen noemen.

  Nu je geïnformeerd bent, staat niets je in de weg om je prive sleutels te beveiligen, maar als je dat niet doet, is dat jouw verantwoording, en heb je de consequenties aan jezelf te danken.

- **"*verbose*" mode**

Het gebeurt dat de connectie faalt om redenen die onbekend zijn voor de gebruiker: voeg dan eenvoudig de "`-v` optie (staat voor "*verbose*") toe aan het *ssh* commando. Aansluitend zullen er een hoop gedetailleerde mededelingen verschijnen op je scherm gedurende de verbinding, dit zal je vaak voldoende informatie opleveren om de oorzaak van de fout te achterhalen.

De encryptie algorithmes

Er is een verschil tussen degene om communicatie kanalen te versleutelen (encryptie met geheime sleutels) en die voor authenticatie (encryptie met publieke sleutels).

Voor authenticatie kunnen we kiezen tussen RSA en DSA met versie 2 van het protocol, en alleen RSA voor versie 1 (geen keus dus...). Historisch werd DSA gekozen, daar RSA in sommige landen *gepatenteerd* was. Sinds het eind zomer van het jaar 2000 is RSA vrij van rechten, en dus is deze noodzaak verdwenen. Ik heb werkelijk geen voorkeur voor wat de goede of slechte keus is (alleen dat DSA een "puur" NSA produkt is).

Voor symmetrische encryptie is er bijna teveel om uit te kiezen... Het protocol bevat een algemeen algorithme dat aanwezig moet zijn in alle implementaties: *triple-DES met 3 sleutels*. Aansluitend zal deze gebruikt worden als de onderhandeling tussen client en server faalt over een ander algortithme. Als je kunt, probeer te onderhandelen met een ander algorithme dan 3DES, daar deze ondertussen een van de minst presenterende algorithmes is. Niettemin zullen we, tenzij noodzakelijk, de exotische of oudere aan kant zetten (arc4, DES, RC4, ...) en ons beperken tot:

- IDEA: presteerd beter dan 3DES, maar is niet helemaal vrij van licenties onder bepaalde omstandigheden (het was vaak het standaard algorithme in Unix versies);
- Blowfish: erg snel, waarschijnlijk veilig, maar het algorithme moet zich nog bewijzen met de tijd;
- AES: de nieuwe standaard (vervangt DES), als het aan beide zijden beschikbaar is, gebruik dan deze, hier is het voor gemaakt.

Persoonlijk vraag ik me af wat het nut is om zoveel algorithmes voor te stellen: zelfs als het protocol toestaat om een "prive algorithme" te gebruiken (voor een bepaalde groep gebruikers, bijvoorbeeld), lijkt me dit essentieel, maar voor normaal gebruik, zal denk ik, met de tijd, AES de standaard worden. Als AES wordt gekraakt zullen de beveiligingsproblemen groter zijn dan die, die door SSH veroorzaakt worden...

Port Forwarding, tunnelen

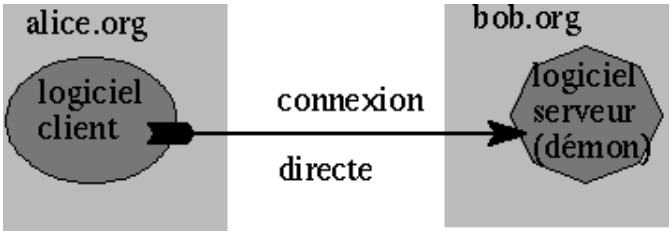
SSH staat toe om een willekeurige TCP data stroom om te leiden (*forwarding*) door een "tunnel" in een SSH sessie. Dit betekend dat de applicatie data-stroom, in plaats van direct door de client en de server poorten te worden beheerd, "ingepakt" wordt in een "tunnel", gecreëerd tijdens connectie (sessie) tijd. (*zie ook het volgende diagram*

Voor het X11 protocol gaat dit zonder byzondere handelingen door de gebruiker in z’n werk, met

transparante afhandeling van de *displays* (schermen) en staat continue aankondigingen toe, zelfs als ze worden gemaakt via meerdere hops.

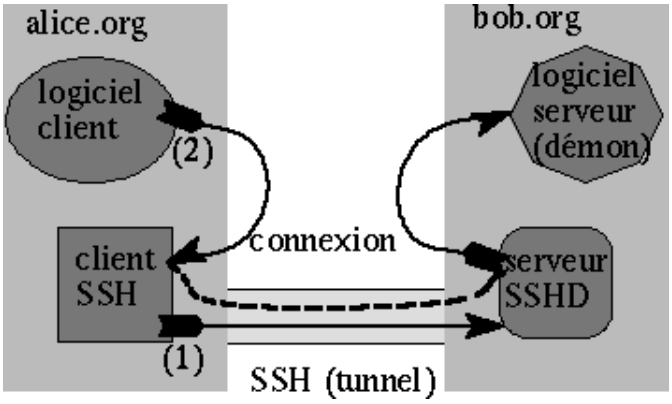
Voor andere stromen is er een commando-regel optie, voor iedere kant:

- Directe verbinding tussen client en server



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

- Lokale poort forwarding (client) naar poort op afstand (server)

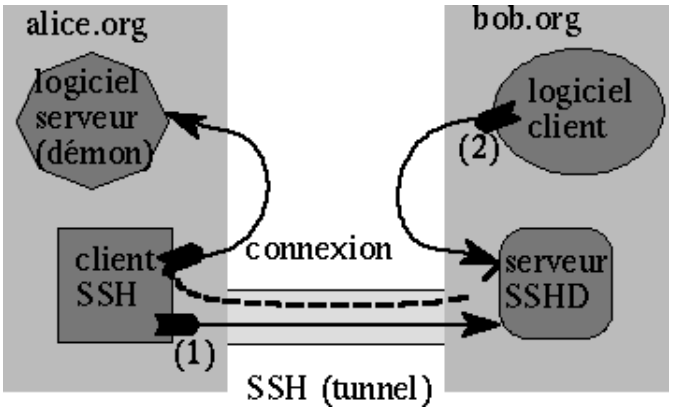


voorbeeld:`user@alice% ssh -L 1234:bob.org:143 bob.org`

dit systeem staat toegang toe vanaf "*alice.org*" naar de imap server van "*bob.org*, verbindingen van buitenaf zullen afgewezen worden door het lokale netwerk. Ze worden alleen beschikbaar gemaakt via het *localhost* adres, poort 1234, vanaf de imap client uitgevoerd op "*alice.org*".

  - (1) de gebruiker op "*alice.org*" opent (verbindt) de SSH tunnel
  - (2) de gebruiker op *"alice.org"* configureerd de clients lokale imap voor toegang tot de imap server die luisterd op *localhost* poort 1234

- Een verre poort (client) doorsturen naar een lokale poort (server)



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

Dit is hetzelfde als hierboven, maar de poort op de host op afstand wordt doorgestuurd. Alleen root heeft de benodigde privileges om dit SSH commando uit te voeren, maar iedere gebruiker kan vervolgens gebruik maken van deze doorgestuurde poort/tunnel.

Deze krachtige feature heeft wel eens voor gezorgt dat SSH de "tunnel voor de armen" werd genoemd. Men moet hier wel bedenken dat hier met de "armen" de gebruikers worden bedoeld die geen administratieve rechten hebben op de client zijde. Alleen in bepaalde gevallen kan een lokale poort worden geforward met lagere rechten (poort > 1024) en zonder super-user rechten ("*root*"). Aan de andere kant, wanneer een geprivileergde poort lokale poort doorgestuurd dient te worden, moet het ofwel met een *root* account gebeuren, of de client moet voorzien worden van super-user privileges ("*suid*") (in feite staat een geprivileergde lokale poort een herdefinitie van een standaard service toe).

Zoals alles met IP is het eenvoudig om iets in iets anders te plaatsen (en andersom), het is niet alleen mogelijk om standaard TCP stromen door te sturen, maar ook PPP connecties, welke ons toestaan een "echte" IP tunnel in IP te maken (versleuteld, dus veilig). De methode valt buiten het bereik van dit artikel, maar je kunt de "Linux VPN-HOWTO" lezen voor meer details en setup scripts (er bestaan ook ’native’ VPN oplossingen voor Linux, zoals "*stunnel"* welke je zou moeten bekijken alvorens een beslissing te nemen).

Onthoud dat de eerste mogelijkheid is om *telnet* stromen om te leiden: dit lijkt volledig nutteloos, daar SSH interactieve verbindingen standaard implementeerd. Echter, met behulp van het doorsturen van *telnet* verbindingen, kun je je favoriete client blijven gebruiken in plaats van de SSH interactieve modus. Vooral in een Windows^{tm} of MacOs^{tm} omgeving kan dit van pas komen, waar de SSH client de ergonomie van de gebruiker niet echt tegemoet komt. Bijvoorbeeld, het "terminal emulatie" deel van de "*Mindterm* client (Java’s SSH client, beschikbaar voor alle moderne systemen) lijdt aan het gebrek aan performance van de Java taal: het kan voordelig zijn om deze client alleen te gebruiken om de SSH tunnel te openen.

Op dezelfde manier, kun je ook een client op afstand starten, zoals een "*xterm*" (met behulp van de automatische X11 forwarding in SSH), welke ons toestaat SSH te gebruiken op X terminals.

Merk op dat de tunnel openblijft, zolang er data doorheen gaat, zelfs als dit niet van de client komt. Dus, het "*sleep*" commando wordt erg bruikbaar om een SSH te openen om een nieuwe TCP connectie door te sturen.

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

% telnet localhost 2323

... welkom bij serveur.org ...
```

De eerste regel opent de tunnel, start het commando "*sleep 60*" op de server, en stuurt de lokale poort 2323 naar de (*telnet*) poort nummer 23 op serveur.org. De tweede start een *telnet* client op de lokale poort 2323 en zal dan de (versleutelde) tunnel gebruiker met de *telnetd* daeon op de server verbinden. Het "*sleep*" commando stopt na een minuut (er is een wachttijd van een minuut om de telnet client te starten), maar SSH zal de tunnel pas afsluiten als de laatste client klaar is.

Belangrijkste distributies: vrij beschikbaar

We moeten onderscheid maken tussen clients en/of servers op de verschillende platformen en je zou moeten weten dat SSH versie 1 en SSH versie 2 niet compatibel zijn. Aan het eind van het artikel, zijn in de referenties andere implementaties met voldoende stabiele features te vinden, die niet in de volgende tabel zijn opgenomen.

produkt	platform	protocol	link	opmerking
OpenSSH	Unix	versies 1 en 2	www.openssh.com	details onder
TTSSH	Windows^{tm}	versie 1	www.zip.com.au/~roca/ttssh.html	
Putty	Windows^{tm}	versie 1 en 2	www.chiark.greenend.org.uk/~sgtatham/putty	alleen beta
Tealnet	Windows^{tm}	versie 1 en 2	telneat.lipetsk.ru	
SSH secure shell	Windows^{tm}	versies 1 en 2	www.ssh.com	vrij voor niet-commercieel gebruik
NiftytelnetSSH	MacOS^{tm}	versie 1	www.lysator.liu.se/~jonasw/freeware/niftyssh/	
MacSSH	MacOS^{tm}	versie 2	www.macssh.com	
MindTerm	Java	versie 1	www.mindbright.se	v2 nu commercieel

Merk op dat MindTerm zowel een onafhankelijke implementatie in Java is, (je hebt alleen een Java *runtime* nodig) als een *servlet* dat uitgevoerd kan worden in een goed ontworpen, compatible Web browser. Helaas zijn de laatste versies van deze geweldige distributie kort geleden commerciële producten geworden.

OpenSSH

Tegenwoordig is dit waarschijnlijk de distributie om te gebruiken in een Unix/Linux omgeving (continue ondersteuning, goede respons tijden, open source en vrij).

De ontwikkeling van OpenSSH begon met de originele versie (SSH 1.2.12) van Tatu Ylonen (de laatste echt vrije) in het OpenBSD 2.6 project (via OSSH). Tegenwoordig wordt OpenSSH ontwikkeld door twee groepen, een die alleen voor het OpenBSD project ontwikkeld, en een die de code continu aanpast om een portable ("draagbare") versie te maken.

Dit alles heeft z'n consequenties, vooral daar de code steeds meer en meer een monster aanpassing wordt (ik voel het "*sendmail*" syndroom aan de horizon verschijnen, dit is niet erg gunstig voor een applicatie is gericht op encryptie die extreem rigoreus zou moeten zijn).

Behalve schoon en leesbare coden, ergeren twee andere punten me:

- OpenSSH gebruikt voor z'n encryptie services de OpenSSL library, en meestal is deze dynamisch gelinkt. In ons geval, lijkt me deze implementatie van een encryptie utilitie pakket met karakteristieken als een hoog beveiligings niveau en zeer betrouwbare features, een slechte benadering. Natuurlijk, een aanval op de library is gelijk aan een aanval op het product. Anders dan een perverse aanval, zullen de encryptie karakteristieken (kwaliteit) in OpenSSH dezelfde zijn/worden als die van de library, welke z'n leven onafhankelijk van OpenSSH leeft.
- OpenSSH gebruikt voor sommige van z'n gevoelige services (zoals bijvoorbeeld een random-nummer generator), de OpenBSD system services. In deze context, zal ik dezelfde opmerkingen maken over externe afhankelijkheden als bij OpenSSL. Nog irritanter is, dat de portable versie van OpenSSH, die op meerdere platformen zou moeten werken, services die door OpenBSD benodigd zijn, delegeert naar andere mechanismen op de doel platformen. Of er bijvoorbeeld nu een random-nummer generator beschikbaar is of niet, voor jouw systeem, we zullen het gebruiken of anders een andere interne. Aansluitend wordt de effectiviteit van OpenSSH afhankelijk van het onderliggende platform, oftewel, je moet maar afwachten wat het wordt.

In mijn opinie (en ik ben hier niet alleen in), zou een multiplatform encryptie product een gedemonstreerd, vastgesteld en constant gedrag moeten vertonen, op welk platform het ook draait, als ook rekening houden met de specifieke karakteristieken en de evolutie van het platform en/of deze elimineren.

Dit gezegd te hebben, we moeten toegeven dat de concurrerende implementaties noch veel, noch aantrekkelijk zijn. Ik geloof dat het pragmatischer is om vandaag de dag OpenSSH als de beroerdste implementatie te zien, en de andere buiten spel te houden ...! Een bruikbaar project voor de gemeenschap zou zijn om de code te herontwerpen en te herschrijven.

Slecht nieuws ...

SSH is niet perfect! Waar het voor is gemaakt, doet het goed, maar je kunt niet vragen om meer. Met name zal het geen "ongeauthoriseerde" connecties voorkomen: als een account wordt gecompromitteerd,

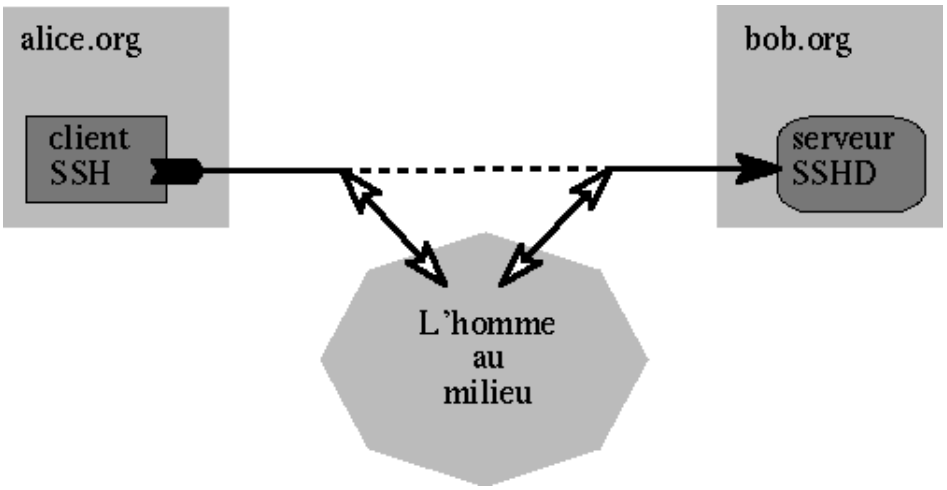
is het mogelijk voor de inbreker om zichzelf via SSH met je computer te verbinden, zelfs al is het de enige manier, daar hij de authenticatie controleerd. SSH is alleen volledig betrouwbaar als het wordt gebruikt in combinatie met een praktisch en coherent beveiligings beleid: Als iemand overal hetzelfde wachtwoord gebruikt, en niet overal gebruik maakt van SSH, wordt het potentiële beveiligings risico slechts licht minder. Je kunt toegeven dat SSH in dit geval gas tegen kan gaan geven, daar de inbreker een beveiligde, versleutelde verbinding met tunneling kan gebruiken, en zo in staat is om bijna alles te doen wat hij wil, zonder de mogelijkheid om hem op te sporen.

In dezelfde stijl, zou men ook rekening moeten houden met goed ontworpen "rootkits" welke vaak een SSH daemon bevatten om een discrete toegang tot het systeem mogelijk te maken, maar met een paar modificaties: hij luistert niet op poort 22 natuurlijk, heeft de eigenschap om niet te loggen, doet zich voor als doodgewone daemon (bijvoorbeeld *httpd*) en is onzichtbaar voor een "ps" commando (ook aangepast door de rootkit).

Aan de andere kant moet men zich ook niet te druk maken over het gevaar dat een SSH daemon, die inbrekers toestaat nog verder verborgen te worden, oplevert: je weet (hopelijk) dat het mogelijk is om bijna alles in alles te stoppen in IP, inclusief "wangedrag" van de meest belangrijke protocollen en door een firewall heen: HTML tunneling, ICMP tunneling, DNS tunneling ... Schakel je systeem dus niet in als je een 100% veilig systeem wilt ;-).

SSH is geen uitzondering als het gaat om beveiligings "loopholes", overgenomen van de implementatie (vele zijn in het verleden gecorrigeerd maar perfecte programmatuur bestaat niet), maar ook op protocol niveau. Deze "loopholes" worden nogal alarmerend aangekondigd, maar behelsen meestal zwakheden die technisch zo complex zijn, dat ze lastig zijn te misbruiken: men moet in gedachten houden dat veel beveiligings incidenten die dagelijks voorkomen, vaak voorkomen hadden kunnen worden door het gebruik van SSH en dat diegenen die door SSH worden veroorzaakt door zwakke punten in SSH vaak nogal theoretisch zijn. Het kan interessant zijn om de volgende studie over "*man in the middle* aanvallen te lezen: http://www.hsc.fr/ressources/presentations/mitm/index.html. Niettemin is het nodig om rekening te houden met deze potentiële zwakke punten voor "high security" applicaties (bankieren, militaire toepassingen ...), waar de bedoelingen van de cracker zeer gemotiveerd zijn door de mogelijke beloning(en).

- "Man in the middle" aanval:



De agressor onderschept de pakketten van beide zijden, en genereert zijn pakketten om beiden te bedriegen
(afwijkende scenarios zijn mogelijk, bijvoorbeeld met het afsluiten van de verbinding aan de ene kant,
en aan de andere kant doorgaan, terwijl deze kant ervan overtuigd is dat het de normale partner is)

Ook mag ik graag wijzen op de onmiskenbare zwakeheid van het protocol voor wat betreft de "padding" (opvulling, ook bekend als bedekt kanaal): in zowel versie 1 als 2 hebben de pakketten een lengte die een veelvoud is van 64 bits, en worden opgevuld met een random nummer. Dit is nogal uitzonderlijk, en leidt daardoor tot een klassieke fout die goed bekend is in encryptie producten: een "verborgen" (of subliminaal") kanaal. Meestal vullen we op met een bekende volgorde, als voorbeeld, geef de waarde *n* aan de byte positie (*zelf beschrijvend opvullen*). In SSH, met de (per definitie) random volgorde, kan het niet worden gecontroleerd. Daardoor is het mogelijk dat een of meer partijen de communicatie misbruiken, bijvoorbeeld gebruikt door een afluisterende derde partij. Ook zou men zich een gecorrumpeerde implementatie kunnen voorstellen, onbekend voor beide partijen (dit kan eenvoudig voorkomen bij een product dat alleen als binaries wordt geleverd, zoals vaak het geval is bij commerciële producten). Dit kan simpel gebeuren, en in dit geval hoeft alleen de server of de client "geïnfecteerd te worden. Ondanks het feit dat zo'n bekende fout in het protocol is achtergelaten, (vooral daar het wijd en zijd bekend is dat de installatie van een "bedekt kanaal" in een encryptie product DE klassieke en eenvoudigste manier is om de communicatie te corrumperen) lijkt het me vrij onwaarschijnlijk dat hier veel misbruik van zou worden gemaakt. Het kan interressant zijn om Bruce Schneier's opmerkingen met betrekking tot zulke elementen in producten, onder toezicht van overheids instanties. (http://www.counterpane.com/crypto-gram-9902.html#backdoors).

Ik zal dit onderwerp afsluiten met de laatste bug die ik vond tijdens het porten van SSH naar SSF (Franse versie van SSH), in de code van de Unix versie voor 1.2.25. De consequentie was dat de random generator ... voorspelbare ... resultaten opleverde (deze situatie valt te betreuren in een cryptografisch produkt, ik zal de technische details achterwege laten, maar het is mogelijk om een communicatie te compromitteren met eenvoudige afluister praktijken). Tegen de tijd dat SSH's ontwikkel team het probleem had gecorrigeerd (slechts een regel om aan te passen) werd er vreemd genoeg geen waarschuwing rondgestuurd en werd het ook niet in de "changelog" genoemd... Als iemand niet wilde dat het bekend zou worden, had hij zich niet anders gedragen. Uiteraard is er geen relatie met de link naar het bovenstaande artikel.

Conclusie

Ik zal nog eens herhalen wat ik schreef in de introductie: SSH produceert geen wonderen, noch lost het alle beveiligings problemen op, maar maakt het wel mogelijk om efficiënt met de meest kwetsbare aspecten van de historische interactieve connectie programma's (telnet, rsh...) om te gaan.

Bibliografie, essentiële links

De volgende twee boeken gaan over SSH versie 1 en SSH versie 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

En als je wat geld wilt investeren, is hier de plek om te beginnen...:

- http://www.ssh.com

het bedekte kanaal exploiteren (mogelijk door de random opvulling in SSHv1)

Het is mogelijk om het bedekte (subliminale) kanaal te exploiteren, welke is ontstaan door het gebruik van de random opvulling in SSHv1 (en v2). Ik ontken alle verantwoordelijkheid voor hartaanvallen die bij de meest paranoïden kunnen voorkomen.

De SSHv1 pakketten hebben de volgende structuur:

offset (bytes)	**naam**	**lengte (bytes)**	**beschrijving**
0	grootte	4	pakket grootte, veld grootte zonder opvulling, dus: grootte = lengte(type) + lengte(data) + lengte (CRC)
4	opvulling	p =1 tot 8	**random** opvulling : grootte wordt zo aangepast dat het versleutelde deel een veelvoud van 8 wordt
4+p	type	1	pakket type
5+p	data	n (variable >= 0)	
5+p+n	controlegetal	4	CRC32

Slechts een veld is niet versleuteld: de "grootte". De lengte van het versleutelde deel is altijd een veelvoud van acht, met behulp van "opvulling" (padding). De padding wordt altijd uitgevoerd, als de lengte van de laatste drie velden reeds een veelvoud van 8 is, zal de padding 8 bytes lang zijn ($5+p+n$, blijft over 0 modulo 8). Neem de codeer functie C, symmetrisch, gebruikt in CBC mode, en de decodeer functie C^{-1}. Om deze demonstratie te vereenvoudigen, zullen we alleen pakketten gebruiken met 8 byte padding. Zodra een van de pakketten aankomt, zullen we, in plaats van deze te padden met een random nummer, een waarde $C(M)$ plaatsen, in dit geval 8 bytes groot. Dit betekend het decoderen van een bericht M met de functie C, gebruikt om het kanaal te versleutelen (het feit dat M ontcijferd wordt zonder eerst versleuteld te worden heeft geen belang vanuit een strict wiskundig oogpunt, ik zal hier niet de praktische implementatie in detail uitleggen). Vervolgens gaan we door naar het normale verwerken

van het pakket, oftewel het coderen van blokken van 8 bytes.

Het resultaat zal zijn:

offset	**bevat**	**opmerkingen**
0	grootte	4 bytes, niet gecodeerd
4	8 bytes opvulling (gecodeerd)	dus $C(C^{-1}(M))$
12... einde	type, data, CRC	

Wat is het verbazingwekkende hier? Het eerste gecodeerde blok bevat $C(C^{-1}(M))$. Dus, daar C een symmetrische encryptie functie is; $C(C^{-1}(M) = M$. Dit eerste blok wordt ongecodeerd verstuurd in een gecodeerde data stroom! Dit betekend alleen dat een ieder die de verbinding afluisterd en kennis heeft van de stratagem zal weten hoe de informatie geëxploiteerd kan worden. Uiteraard kan men aannemen dat het bericht gecodeerd is (met een publieke sleutel, bijvoorbeeld, welke een geheim bevat in de gecompromitteerde code), en dus beveiligd is tegen iemand die niet geïnformeerd is.

Bijvoorbeeld, er zijn drie pakketten van dit type nodig om de triple-DES (168 bit) sessie sleutel te sniffen, waarna de flux sniffer de gehele communicatie kan ontcijferen. Als de sleutel wordt verstuurd, is het net langer nodig om van te voren de padding te ontsleutelen, die vervolgens in het pakket wordt geïnjecteerd, het is mogelijk om paddings van elke grootte te gebruiken, als men nog meer informatie wil toevoegen.

Het gebruik van dit bedekte kanaal is *absoluut niet detecteerbaar*! (men moet erg voorzichtig zijn bij het coderen van ieder element van het bericht, zoals eerder uitgelegd, zodat de entropy van het blok niet de strategie onthuld. Niet detecteerbaar omdat de padding random is, en zo mogelijke validatie tests elimineerd. Random padding zou *nooit* worden toegepast in cryptografische produkten.

Wat dit kanaal nog gevaarlijker maakt dan andere in het protocol wordt veroorzaakt door berichten als SSH_MSG_IGNORE waardoor je er gebruik van kunt gebruiken *zonder* kennis van de gecodeerde sleutel.

Om de ’perverse’ effecten van random padding te voorkomen, zou men in het protocol deterministische padding moeten definiëren: vaak ook "*zelf beschrijvende padding*" genoemd. Dit houdt in dat de offset byte *n*, *n* bevat. Random padding komt voor in SSH v2, het is een keus, dus hou het in gedachten...

Ter afsluiting, zou ik nog willen zeggen dat als ik het bedekte kanaal becritiseer, het alleen is omdat ik graag wil zien dat een produkt als SSH, welke zich toelegt op sterke beveiliging, een maximum aan beveiliging levert. Nu ben je in staat om je voor te stellen dat er meer potentiële problemen liggen in commerciële produkten: alleen open source produkten kunnen een oplossing bieden voor de primaire behoefte, de mogelijkheid om de code te controleren (ook al dient het controleren van de code vaak gedaan te worden.

Site onderhouden door het LinuxFocus editors team © Bernard Perrot "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org	Vertaling info: fr --> -- : Bernard Perrot <bernard.perrot(at)univ-rennes1.fr> fr --> en: Guy Passemard <g.passemard(at)free.fr> en --> nl: GH Snijders <ghs(at)linuxfocus.org>

2005-01-14, generated by lfparser_pdf version 2.51