RFC 4253 : The Secure Shell (SSH) Transport Layer Protocol

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 2 août 2020

Date de publication du RFC : Janvier 2006

https://www.bortzmeyer.org/4253.html

Le protocole SSH de sécurité est composé de plusieurs parties, spécifiées dans des RFC différents. Ici, il s'agit de la couche basse de SSH, située juste au-dessus de TCP, et en dessous de protocoles qui permettent, par exemple, la connexion à distance. Cette couche basse fournit l'authentification (de la machine, pas de l'utilisateur) et la confidentialité, via de la cryptographie.

Le principe général de cette partie de SSH est classique, et proche de celui d'autres protocoles de cryptographie comme TLS (RFC 4251 [1] pour une vue générale de l'architecture de SSH). Le client établit une connexion TCP avec le serveur puis chaque partie envoie une liste des algorithmes utilisés pour l'authentification, l'échange de clés et le chiffrement ultérieur. Une fois un accord trouvé, le serveur est authentifié, typiquement par de la cryptographie asymétrique, comme ECDSA. Un mécanisme d'échange de clés comme Diffie-Hellman permet de choisir des clés partagées qui serviront pour les opérations de chiffrement symétrique (par exemple avec AES) qui chiffreront tout le reste de la communication. Ce processus de négociation fournit l'agilité cryptographique (RFC 7696). Du fait de cette agilité, la liste des algorithmes utilisables n'est pas figée dans les RFC. Démarrée par le RFC 4250, sa version à jour et faisant autorité est à l'IANA <https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xml>.

Notez que j'ai simplifié le mécanisme de négociation : il y a d'autres choix à faire dans la négociation, comme celui de l'algorithme de MAC. Les autres fonctions de SSH, comme l'authentification du client, prennent place ensuite, une fois la session de transport établie selon les procédures définies dans notre RFC 4253.

L'établissement de connexion est normalisé dans la section 4 du RFC. SSH suppose qu'il dispose d'un transport propre, acheminant des octets sans les modifier et sans les interpréter. (TCP répond à cette exigence, qui est très banale.) Le port par défaut est 22 (et il est enregistré à l'IANA <https://www.iana.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc4251.txt

org/assignments/service-names-port-numbers/service-names-port-numbers.xml>, lisez donc le récit de cet enregistrement <https://www.ssh.com/ssh/port#how-ssh-port-became-22>). En pratique, il est fréquent que SSH utilise d'autres ports, par exemple pour ralentir certaines attaques par force brute <https://www.bortzmeyer.org/sshd-port-alternatif.html> (OpenSSH rend cela très simple, en indiquant dans son `/.ssh/config` le port du serveur où on se connecte). Une fois la connexion TCP établie, chaque machine envoie une bannière qui indique sa version de SSH, ici la version 2 du **protocole** (la seule survivante aujourd'hui) et la version 7.6 de sa mise en œuvre OpenSSH :

```
% telnet localhost 22
...
SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.3
```

Un établissement de session complet comprend :

— Connexion TCP,
— envoi des bannières (qui peut être fait simultanément par le serveur et le client, pas besoin d'attendre l'autre),
— début de la procédure d'échange de clés (KEX, pour "*Key EXchange*") avec les messages `SSH_MSG_KEXINIT` (qui peuvent aussi être envoyés simultanément),
— échange de clés proprement dit, par exemple avec des messages Diffie-Hellman.

Comme certains messages peuvent être envoyés en parallèle (sans attendre l'autre partie), il est difficile de compter combien d'aller-retours cela fait. (Optimiste, le RFC dit que deux aller-retours suffisent dans la plupart des cas.)

Le protocole décrit dans ce RFC peut être utilisé pour divers services (cf. RFC 4252 et RFC 4254). Une des utilisations les plus fréquentes est celle de sessions interactives à distance, où SSH avait rapidement remplacé telnet. À l'époque, certaines personnes s'étaient inquiétées de l'augmentation de taille des paquets due à la cryptographie (nouveaux en-têtes, et MAC). Le débat est bien dépassé aujourd'hui mais une section 5.3 du RFC discute toujours la question. SSH ajoute au moins 28 octets à chaque paquet. Pour un transfert de fichiers, où les paquets ont la taille de la MTU, cette augmentation n'est pas importante. Pour les sessions interactives, où il n'y a souvent qu'un seul octet de données dans un paquet, c'est plus significatif, mais il faut aussi prendre en compte le reste des en-têtes. Ainsi, IP et TCP ajoutent déjà 32 octets au minimum (en IPv4). Donc l'augmentation due à SSH n'est pas de 2800 % mais de seulement 55 %. Et c'est sans même prendre en compte les en-têtes Ethernet. En parlant d'Ethernet, il faut aussi noter que la taille minimale d'une trame Ethernet est de 46 octets (cf. RFC 894), de toute façon et que donc, sans SSH, il faudrait de toute façon remplir la trame...Bref, le problème n'est guère important. (Le RFC donne un autre exemple, avec des modems lents, qui n'est probablement plus d'actualité aujourd'hui.)

La section 6 du RFC décrit ensuite le format des en-têtes SSH, un format binaire, comme le DNS ou BGP, mais pas comme SMTP ou HTTP version 1. Le tout est évidemment chiffré (même la longueur du paquet, qu'on ne peut donc pas connaitre avant de déchiffrer). Le chiffrement se fait avec l'algorithme sélectionné dans la phase de négociation. Si vous utilisez OpenSSH, l'option `-v` affichera l'algorithme, ici ChaCha20 :

```
% ssh -v $ADDRESS
...
debug1: kex: server->client cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
debug1: kex: client->server cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
```

ChaCha20 n'existait pas à l'époque du RFC 4253. Il n'est toujours pas enregistré à l'IANA <https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xml#ssh-parameters-17>. C'est ce qui explique que le nom d'algorithme inclut un arobase suivi d'un nom de domaine, c'est la marque des algorithmes locaux (ici, local à OpenSSH), non enregistrés officiellement. Un article du programmeur <http://blog.djm.net.au/2013/11/chacha20-and-poly1305-in-openssh.html> décrit l'intégration de ChaCha20 dans OpenSSH. Il y avait eu une tentative de normalisation (dans l'"*Internet-Draft*" draft-josefsson-ssh-chacha20-poly1305-openssh <https://datatracker.ietf.org/doc/draft-josefsson-ssh-chacha20-poly1305-openssh/>) mais elle a été abandonnée. C'était peut-être simplement par manque de temps et d'intérêt, car ChaCha20 est utilisé dans d'autres protocoles cryptographiques de l'IETF comme TLS (cf. RFC 7905 et, pour une vision plus générale, RFC 8439). OpenSSH permet d'afficher la liste des algorithmes de chiffrement symétriques qu'il connait avec l'option `-Q cipher` :

```
% ssh -Q cipher
3des-cbc
aes128-cbc
aes192-cbc
aes256-cbc
rijndael-cbc@lysator.liu.se
aes128-ctr
aes192-ctr
aes256-ctr
aes128-gcm@openssh.com
aes256-gcm@openssh.com
chacha20-poly1305@openssh.com
```

À l'inverse, le RFC 4253 listait des algorithmes qui ont été abandonnés depuis comme RC4 (nommé `arcfour` dans le RFC pour des raisons juridiques), retiré par le RFC 8758. La liste comprend également l'algorithme `none` (pas de chiffrement) qui avait été inclus pour des raisons douteuses de performance et dont l'usage est évidemment déconseillé (section 6.3 de notre RFC).

Le paquet inclut également un MAC pour assurer son intégrité. Il est calculé avant le chiffrement. Pour le MAC aussi, la liste des algorithmes a changé depuis la parution du RFC 4253. On peut afficher ceux qu'OpenSSH connait (les officiels sont dans un registre IANA <https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xml#ssh-parameters-18>) :

```
% ssh -Q mac
hmac-sha1
hmac-sha1-96
hmac-sha2-256
hmac-sha2-512
hmac-md5
hmac-md5-96
umac-64@openssh.com
umac-128@openssh.com
hmac-sha1-etm@openssh.com
hmac-sha1-96-etm@openssh.com
hmac-sha2-256-etm@openssh.com
hmac-sha2-512-etm@openssh.com
hmac-md5-etm@openssh.com
hmac-md5-96-etm@openssh.com
umac-64-etm@openssh.com
umac-128-etm@openssh.com
```

Bon, mais les algorithmes de chiffrement symétrique comme ChaCha20 nécessitent une clé secrète et partagée entre les deux parties. Comment est-elle négociée ? Un algorithme comme par exemple Diffie-Hellman sert à choisir un secret, d'où sera dérivée la clé.

Le format binaire des messages est décrit en section 6 en utilisant des types définis dans le RFC 4251, section 5, comme `byte[n]` pour une suite d'octets ou `uint32` pour un entier non signé sur 32 bits. Ainsi, un certificat ou une clé publique sera :

```
string    certificate or public key format identifier
byte[n]   key/certificate data
```

Un paquet SSH a la structure (après le point-virgule, un commentaire) :

```
uint32    packet_length
byte      padding_length
byte[n1]  payload; n1 = packet_length - padding_length - 1
byte[n2]  random padding; n2 = padding_length
byte[m]   mac (Message Authentication Code - MAC); m = mac_length
```

La charge utile (`payload`) a un format qui dépend du type de messages. Par exemple, le message d'échange de clés initial, où chacun indique les algorithmes qu'il connait :

```
byte         SSH_MSG_KEXINIT
byte[16]     cookie (random bytes)
name-list    kex_algorithms
name-list    server_host_key_algorithms
name-list    encryption_algorithms_client_to_server
name-list    encryption_algorithms_server_to_client
name-list    mac_algorithms_client_to_server
name-list    mac_algorithms_server_to_client
name-list    compression_algorithms_client_to_server
name-list    compression_algorithms_server_to_client
name-list    languages_client_to_server
name-list    languages_server_to_client
boolean      first_kex_packet_follows
uint32       0 (reserved for future extension)
```

Les types de message ("*message ID*" ou "*message numbers*") possibles sont listés dans un registre IANA `<https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xml#ssh-parameters-1>`. Par exemple l'exemple ci-dessus est un message de type `SSH_MSG_KEXINIT`, type qui est défini en section 12 et dans le RFC 4250, section 4.1.2. Il a la valeur 20.

Affiché par Wireshark, voici quelques messages que SSH transmet en clair, avant le début du chiffrement. Après l'échange des bannières, il y aura le `KEXINIT` en clair, Wireshark suit les termes du RFC donc vous pouvez facilement comparer ce message réel à la description abstraite du RFC, ci-dessus :

```
SSH Protocol
    SSH Version 2 (encryption:chacha20-poly1305@openssh.com mac:<implicit> compression:none)
        Packet Length: 1388
        Padding Length: 4
        Key Exchange
            Message Code: Key Exchange Init (20)
            Algorithms
                kex_algorithms length: 269
```

```
            kex_algorithms string [truncated]: curve25519-sha256,curve25519-sha256@libssh.org,ecdh-sha2-nistp
            server_host_key_algorithms length: 358
            server_host_key_algorithms string [truncated]: ecdsa-sha2-nistp256-cert-v01@openssh.com,ecdsa-sha
            encryption_algorithms_client_to_server length: 108
            encryption_algorithms_client_to_server string: chacha20-poly1305@openssh.com,aes128-ctr,aes192-ct
            encryption_algorithms_server_to_client length: 108
            encryption_algorithms_server_to_client string: chacha20-poly1305@openssh.com,aes128-ctr,aes192-ct
            mac_algorithms_client_to_server length: 213
            mac_algorithms_client_to_server string [truncated]: umac-64-etm@openssh.com,umac-128-etm@openssh
            mac_algorithms_server_to_client length: 213
            mac_algorithms_server_to_client string [truncated]: umac-64-etm@openssh.com,umac-128-etm@openssh
            compression_algorithms_client_to_server length: 26
            compression_algorithms_client_to_server string: none,zlib@openssh.com,zlib
            compression_algorithms_server_to_client length: 26
            compression_algorithms_server_to_client string: none,zlib@openssh.com,zlib
```

Notez qu'un seul message SSH contient tous les algorithmes, aussi bien ceux servant à l'authentification qu'à l'échange de clés, ou qu'au chiffrement symétrique. Le premier algorithme listé est le préféré. Si les deux parties ont le même algorithme préféré, il est choisi. Autrement, on boucle sur les algorithmes listés par le client, dans l'ordre, en sélectionnant le premier qui est accepté par le serveur. Voici maintenant le lancement de l'échange de clés :

```
SSH Protocol
    SSH Version 2 (encryption:chacha20-poly1305@openssh.com mac:<implicit> compression:none)
        Packet Length: 44
        Padding Length: 6
        Key Exchange
            Message Code: Diffie-Hellman Key Exchange Init (30)
            Multi Precision Integer Length: 32
            DH client e: c834ef50c9ce27fa5ab43886aec2161a3692dd5e3267b567...
            Padding String: 000000000000
```

Et la réponse en face :

```
SSH Protocol
    SSH Version 2 (encryption:chacha20-poly1305@openssh.com mac:<implicit> compression:none)
        Packet Length: 260
        Padding Length: 9
        Key Exchange
            Message Code: Diffie-Hellman Key Exchange Reply (31)
            KEX host key (type: ecdsa-sha2-nistp256)
            Multi Precision Integer Length: 32
            DH server f: 2e34deb08146063fdba1d8800cf853a3a6830a3b8549ee5b...
            KEX H signature length: 101
            KEX H signature: 0000001365636473612d736861322d6e6973747032353600...
            Padding String: 0000000000000000
    SSH Version 2 (encryption:chacha20-poly1305@openssh.com mac:<implicit> compression:none)
        Packet Length: 12
        Padding Length: 10
        Key Exchange
            Message Code: New Keys (21)
            Padding String: 00000000000000000000
```

À partir de là, tout est chiffré et Wireshark ne peut plus afficher que du binaire sans le comprendre :

```
SSH Protocol
    SSH Version 2 (encryption:chacha20-poly1305@openssh.com mac:<implicit> compression:none)
        Packet Length (encrypted): 44cdc717
        Encrypted Packet: ea98dbf6cc50af65ba4186bdb5bf02aa9e8366aaa4c1153f
        MAC: e7a6a5a222fcdba71e834f3bb76c2282
```

OpenSSH avec son option `-v` permet d'afficher cette négociation :

```
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex: algorithm: curve25519-sha256
debug1: kex: host key algorithm: ecdsa-sha2-nistp256
debug1: kex: server->client cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
debug1: kex: client->server cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
```

Et les algorithmes acceptés pour l'échange de clés :

```
% ssh -Q kex
diffie-hellman-group1-sha1
diffie-hellman-group14-sha1
diffie-hellman-group14-sha256
diffie-hellman-group16-sha512
diffie-hellman-group18-sha512
diffie-hellman-group-exchange-sha1
diffie-hellman-group-exchange-sha256
ecdh-sha2-nistp256
ecdh-sha2-nistp384
ecdh-sha2-nistp521
curve25519-sha256
curve25519-sha256@libssh.org
```

(Ceux à courbes elliptiques comme `curve25519-sha256` n'existaient pas à l'époque. `curve25519-sha256` a été ajouté dans le RFC 8731.) On trouve aussi les algorithmes acceptés pour l'authentification du serveur via sa clé publique :

```
% ssh -Q key
ssh-ed25519
ssh-ed25519-cert-v01@openssh.com
ssh-rsa
ssh-dss
ecdsa-sha2-nistp256
ecdsa-sha2-nistp384
ecdsa-sha2-nistp521
ssh-rsa-cert-v01@openssh.com
ssh-dss-cert-v01@openssh.com
ecdsa-sha2-nistp256-cert-v01@openssh.com
ecdsa-sha2-nistp384-cert-v01@openssh.com
ecdsa-sha2-nistp521-cert-v01@openssh.com
```

Ce RFC 4253 ne décrit que la couche basse de SSH. Au-dessus se trouvent plusieurs services qui tirent profit de la sécurité fournie par cette couche basse. Une fois la session chiffrée établie, le protocole SSH permet de lancer d'autres services. Ils ne sont que deux depuis le début, mais d'autres pourront se rajouter dans le registre IANA <https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xml#ssh-parameters-9>. Actuellement, il y a `ssh-userauth` (authentifier l'utilisateur, par mot de passe ou par clé publique, RFC 4252) et `ssh-connection` (connexion à distance, et services qui en dépendent, cf. RFC 4254).

Nous avons vu plus haut le type de messages `SSH_MSG_KEXINIT` (code numérique 20). Il y a de nombreux autres types, présentés dans la section 11 du RFC. Par exemple `SSH_MSG_DISCONNECT` (type 1 dans le registre IANA <https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xml#ssh-parameters-1>) est défini ainsi :

```
byte      SSH_MSG_DISCONNECT
uint32    reason code
string    description in ISO-10646 UTF-8 encoding [RFC3629]
string    language tag [RFC3066]
```

Il indique la fin de la session SSH (l'étiquette de langue pour indiquer la raison est désormais normalisée dans le RFC 5646, et plus le RFC 3066). Autre exemple, `SSH_MSG_IGNORE` (code 2) pour faire du remplissage afin de diminuer les risques d'analyse du trafic. Pour une vision complète des risques de sécurité de SSH, (re)lisez la très détaillée section 9 du RFC 4251.

Notez qu'il avait existé une version 1 du protocole SSH, qui n'a pas fait l'objet d'une normalisation formelle. La section 5 de notre RFC traite de la compatibilité entre la version actuelle, la 2, et cette vieille version 1, aujourd'hui complètement abandonnée. Ça, c'est le passé. Et le futur ? Le cœur de SSH n'a pas bougé depuis les RFC de la série du RFC 4250 au RFC 4254. Mais il existe un projet (pour l'instant individuel, non accepté par l'IETF de remplacer ce RFC 4253 par QUIC <https://www.bortzmeyer.org/quic.html>, en faisant tourner le reste de SSH sur QUIC (lisez `draft-bider-ssh-quic`).