Wie man SSH möglichst sicher konfiguriert

Secure Secure Shell

2015-01-04 crypto, nsa, and ssh

You may have heard that the NSA can decrypt SSH at least some of the time. If you have not, then read the latest batch of Snowden documents now. All of it. This post will still be here when you finish. My goal with this post here is to make NSA analysts sad.

TL;DR: Scan this post for fixed width fonts, these will be the config file snippets and commands you have to use.

Warning: You will need a recent (2013 or so) OpenSSH version.

The crypto

Reading the documents, I have the feeling that the NSA can 1) decrypt weak crypto and 2) steal keys. Let's focus on the crypto first. SSH supports different key exchange algorithms, ciphers and message authentication codes. The server and the client choose a set of algorithms supported by both, then proceed with the key exchange. Some of the supported algorithms are not so great and should be disabled completely. If you leave them enabled but prefer secure algorithms, then a man in the middle might downgrade you to bad ones. This hurts interoperability but everyone uses OpenSSH anyway.

Key exchange

There are basically two ways to do key exchange: Diffie-Hellman and Elliptic Curve Diffie-Hellman. Both provide forward secrecy which the NSA hates because they can't use passive collection and key recovery later. The server and the client will end up with a shared secret number at the end without a passive eavesdropper learning anything about this number. After we have a shared secret we have to derive a cryptographic key from this using a key derivation function. In case of SSH, this is a hash function.

DH works with a multiplicative group of integers modulo a prime. Its security is based on the hardness of the discrete logarithm problem.

```
Alice           Bob
---------------------------
Sa = random
Pa = g^Sa   --> Pa
                Sb = random
Pb          <-- Pb = g^Sb
s = Pb^Sa       s = Pa^Sb
k = KDF(s)      k = KDF(s)
```

ECDH works with elliptic curves over finite fields. Its security is based on the hardness of the elliptic curve discrete logarithm problem.

```
Alice           Bob
---------------------------
Sa = random
Pa = Sa * G --> Pa
                Sb = random
Pb          <-- Pb = Sb * G
```

```
s = Sa * Pb      s = Sb * Pa
k = KDF(s)       k = KDF(s)
```

OpenSSH supports 8 key exchange protocols:

1. curve25519-sha256: ECDH over Curve25519 with SHA2
2. diffie-hellman-group1-sha1: 1024 bit DH with SHA1
3. diffie-hellman-group14-sha1: 2048 bit DH with SHA1
4. diffie-hellman-group-exchange-sha1: Custom DH with SHA1
5. diffie-hellman-group-exchange-sha256: Custom DH with SHA2
6. ecdh-sha2-nistp256: ECDH over NIST P-256 with SHA2
7. ecdh-sha2-nistp384: ECDH over NIST P-384 with SHA2
8. ecdh-sha2-nistp521: ECDH over NIST P-521 with SHA2

We have to look at 3 things here:

- ECDH curve choice: This eliminates 6-8 because NIST curves suck. They leak secrets through timing side channels and off-curve inputs. Also, NIST is considered harmful and cannot be trusted.
- Bit size of the DH modulus: This eliminates 2 because the NSA has supercomputers and possibly unknown attacks. 1024 bits simply don't offer sufficient security margin.
- Security of the hash function: This eliminates 2-4 because SHA1 is broken. We don't have to wait for a second preimage attack that takes 10 minutes on a cellphone to disable it right now.

We are left with 1 and 5. 1 is better and it's perfectly OK to only support that but for interoperability, 5 can be included.

Recommended /etc/ssh/sshd_config snippet:

```
KexAlgorithms curve25519-sha256@libssh.org,diffie-hellman-group-exchange-sha256
```

Recommended /etc/ssh/ssh_config snippet:

```
Host *
    KexAlgorithms curve25519-sha256@libssh.org,diffie-hellman-group-exchange-sha256
```

If you chose to enable 5, open /etc/ssh/moduli if exists, and delete lines where the 5th column is less than 2000. If it does not exist, create it:

```
ssh-keygen -G /tmp/moduli -b 4096
ssh-keygen -T /etc/ssh/moduli -f /tmp/moduli
```

This will take a while so continue while it's running.

Authentication

The key exchange ensures that the server and the client shares a secret no one else knows. We also have to make sure that they share this secret with each other and not an NSA analyst. There are 4 public key algorithms for authentication:

1. DSA
2. ECDSA
3. Ed25519
4. RSA

Number 2 here involves NIST suckage and should be disabled. Unfortunately, DSA keys must be exactly 1024 bits so let’s disable that as well.

```
Protocol 2
HostKey /etc/ssh/ssh_host_ed25519_key
HostKey /etc/ssh/ssh_host_rsa_key
```

This will also disable the horribly broken v1 protocol that you should not have enabled in the first place. We should remove the unused keys and only generate a large RSA key and an Ed25519 key. Your init scripts may recreate the unused keys. If you don’t want that, remove any ssh-keygen commands from the init script.

```
cd /etc/ssh
rm ssh_host_*key*
ssh-keygen -t ed25519 -f ssh_host_ed25519_key < /dev/null
ssh-keygen -t rsa -b 4096 -f ssh_host_rsa_key < /dev/null
```

Generate client keys using the following commands:

```
ssh-keygen -t ed25519 -o -a 100
ssh-keygen -t rsa -b 4096 -o -a 100
```

Symmetric ciphers

Symmetric ciphers are used to encrypt the data after the initial key exchange and authentication is complete.

Here we have quite a few algorithms:

1. 3des-cbc
2. aes128-cbc
3. aes192-cbc
4. aes256-cbc
5. aes128-ctr
6. aes192-ctr
7. aes256-ctr
8. aes128-gcm
9. aes256-gcm
10. arcfour
11. arcfour128
12. arcfour256
13. blowfish-cbc
14. cast128-cbc
15. chacha20-poly1305

We have to consider the following:

- Security of the cipher algorithm: This eliminates 1 and 10-12 - both DES and RC4 are broken.

Again, no need to wait for them to become even weaker, disable them now.
- Key size: At least 128 bits, the more the better.
- Block size: Does not apply to stream ciphers. At least 128 bits. This elminates 14 because CAST has a 64 bit block size.
- Cipher mode: The recommended approach here is to prefer AE modes and optionally allow CTR for compatibility. CTR with Encrypt-then-MAC is provably secure.

This leaves 5-9 and 15. Chacha20-poly1305 is preferred over AES-GCM because the latter does not encrypt message sizes.

Recommended /etc/ssh/sshd_config snippet:

```
Ciphers chacha20-poly1305@openssh.com,aes256-gcm@openssh.com,aes128-gcm@openssh.com,
aes256-ctr,aes192-ctr,aes128-ctr
```

Recommended /etc/ssh/ssh_config snippet:

```
Host *
    Ciphers chacha20-poly1305@openssh.com,aes256-gcm@openssh.com,aes128-gcm@openssh.
com,aes256-ctr,aes192-ctr,aes128-ctr
```

Message authentication codes

Encryption provides confidentiality, message authentication code provides integrity. We need both. If an AE cipher mode is selected, then extra MACs are not used, the integrity is already given. If CTR is selected, then we need a MAC to calculate and attach a tag to every message.

There are multiple ways to combine ciphers and MACs - not all of these are useful. The 3 most common:

- Encrypt-then-MAC: encrypt the message, then attach the MAC of the ciphertext.
- MAC-then-encrypt: attach the MAC of the plaintext, then encrypt everything.
- Encrypt-and-MAC: encrypt the message, then attach the MAC of the plaintext.

Only Encrypt-then-MAC should be used, period. Using MAC-then-encrypt have lead to many attacks on TLS while Encrypt-and-MAC have lead to not quite that many attacks on SSH. The reason for this is that the more you fiddle with an attacker provided message, the more chance the attacker has to gain information through side channels. In case of Encrypt-then-MAC, the MAC is verified and if incorrect, discarded. Boom, one step, no timing channels. In case of MAC-then-encrypt, first the attacker provided message has to be decrypted and only then can you verify it. Decryption failure (due to invalid CBC padding for example) may take less time than verification failure. Encrypt-and-MAC also has to be decrypted first, leading to the same kind of potential side channels. It’s even worse because no one said that a MAC’s output can’t leak what its input was. SSH by default, uses this method.

Here are the available MAC choices:

1. hmac-md5
2. hmac-md5-96
3. hmac-ripemd160
4. hmac-sha1
5. hmac-sha1-96
6. hmac-sha2-256
7. hmac-sha2-512
8. umac-64

9. umac-128
10. hmac-md5-etm
11. hmac-md5-96-etm
12. hmac-ripemd160-etm
13. hmac-sha1-etm
14. hmac-sha1-96-etm
15. hmac-sha2-256-etm
16. hmac-sha2-512-etm
17. umac-64-etm
18. umac-128-etm

The selection considerations:

- Security of the hash algorithm: No MD5 and SHA1. Yes, I know that HMAC-SHA1 does not need collision resistance but why wait? Disable weak crypto today.
- Encrypt-then-MAC only: This eliminates the first half, the ones without -etm. You may be forced to enable non-etm algorithms on for some hosts (github). I am not aware of a security proof for CTR-and-HMAC but I also don't think CTR decryption can fail.
- Tag size: At least 128 bits. This eliminates umac-64-etm.
- Key size: At least 128 bits. This doesn't eliminate anything at this point.

Recommended /etc/ssh/sshd_config snippet:

```
MACs hmac-sha2-512-etm@openssh.com,hmac-sha2-256-etm@openssh.com,hmac-
ripemd160-etm@openssh.com,umac-128-etm@openssh.com
```

Recommended /etc/ssh/ssh_config snippet:

```
# Github supports neither AE nor Encrypt-then-MAC.
Host github.com
    MACs hmac-sha2-512-etm@openssh.com,hmac-sha2-256-etm@openssh.com,hmac-ripemd160-
etm@openssh.com,umac-128-etm@openssh.com,hmac-sha2-512,hmac-sha2-256,hmac-
ripemd160,umac-128

Host *
    MACs hmac-sha2-512-etm@openssh.com,hmac-sha2-256-etm@openssh.com,hmac-
ripemd160-etm@openssh.com,umac-128-etm@openssh.com
```

Preventing key theft

Even with forward secrecy the secret keys must be kept secret. The NSA has a database of stolen keys - you do not want your key there.

System hardening

This post is not intended to be a comprehensive system security guide. Very briefly:

- Don't install what you don't need: Every single line of code has a chance of containing a bug. Some of these bugs are security holes. Fewer lines, fewer holes.
- Use free software: As in speech. You want to use code that's actually reviewed or that you can review yourself. There is no way to achieve that without source code. Someone may have reviewed proprietary crap but who knows.
- Keep your software up to date: New versions often fix critical security holes.
- Exploit mitigation: Sad but true - there will always be security holes in your software. There

are things you can do to prevent their exploitation such GCC's -fstack-protector. One of the best security projects out there is Grsecurity. Use it or use OpenBSD.

Traffic analysis resistance

Set up Tor hidden services for your SSH servers. This has multiple advantages. It provides an additional layer of encryption and server authentication. People looking at your traffic will not know your IP, so they will be unable to scan and target other services running on the same server and client. Attackers can still attack these services but don't know if it has anything to do with the observed traffic until they actually break in.

Now this is only true if you don't disclose your SSH server's fingerprint in any other way. You should only accept connections from the hidden service or from LAN, if required.

If you don't need LAN access, you can add the following line to /etc/ssh/sshd_config:

ListenAddress 127.0.0.1:22

Add this to /etc/tor/torrc:

```
HiddenServiceDir /var/lib/tor/hidden_service/ssh
HiddenServicePort 22 127.0.0.1:22
```

You will find the hostname you have to use in /var/lib/tor/hidden_service/ssh/hostname. You also have to configure the client to use Tor. For this, socat will be needed. Add the following line to /etc/ssh/ssh_config:

```
Host *.onion
    ProxyCommand socat - SOCKS4A:localhost:%h:%p,socksport=9050

Host *
    ...
```

If you want to allow connections from LAN, don't use the ListenAddress line, configure your firewall instead.

Key storage

You should encrypt your client key files using a strong password. Additionally, you can use ssh-keygen -o -a $number to slow down cracking attempts by iterating the hash function many times. You may want to store them on a pendrive and only plug it in when you want to use SSH. Are you more likely to lose your pendrive or have your system compromised? I don't know.

Unfortunately, you can't encrypt your server key and it must be always available, or else sshd won't start. The only thing protecting it is OS access controls.

The end

It's probably a good idea to test the changes. ssh -v will print the selected algorithms and also makes problems easier to spot. Be extremely careful when configuring SSH on a remote host. Always keep an active session, never restart sshd. Instead you can send the SIGHUP signal to reload the configuration without killing your session. You can be even more careful by starting a new sshd instance on a different port and testing that.

Quelle: https://stribika.github.io/2015/01/04/secure-secure-shell.html

Eindeutige ID: #1025
Verfasser: n/a
Letzte Änderung: 2015-01-07 14:03