Linux General Knowledge

Ever wonder what apt-get does underneath?

Shower thoughts

This question came from when I was showering one day: What does `apt-get` or `apt` or `yum, pacman` does underneath to install all those programs that you have specified?

Well you're in luck because the past me have did a good research on how it works. Here is the jist of it:

- When you are issuing the command `apt-get update` it will look into your `/etc/apt/source.list` file, this file contains a list of repository files. You will see lines like

  ```
  deb http://us.archive.ubuntu.com/ubuntu/ jammy-backports main restricted universe
  multiverse
  ```

  This indicates a Package file that `apt-get update` will be fetching. For example, http://security.ubuntu.com/ubuntu/dists/jammy/main/binary-amd64/Packages.gz for the Jammy release of Ubuntu. It will store these files for each line of repository that you have into `/var/lib/apt/lists`.

  If you decompress the file, you will see metadata of each package that you can install via `apt-get`, for example, neofetch. The metadata contains information such as where you can find the `.deb` file from the relative URL. It also list out the dependencies that `apt-get` need to install in order run the program.
- Finally, when you issue the `apt-get install` command, it will search for the program that you have specified and if it is found in one of those source list, it will go to the relative URL and fetch the `.deb`. `.deb` files are just pre-compiled binary and it will just install that into your system.

> note the actual installation process is done by `dpkg` underneath for Debian based distro. `apt-get` is really just a wrapped on top of `dpkg` for fetching those pre-compiled binaries for `dpkg` to install.

So under the hood package managers like `apt-get` is looking up list of packages to know where to find their pre-compiled binary. When you ask to install it, it will go and fetch the pre-compiled binary and install it into your system.

Pre-compiled vs compiling from source

Compiling from source is what it sounds like. You have the source code, for example, bunch of `.c` files, and you will compile it into a program and install it into your system yourself.

Pre-compiled binary is a binary executable that is compiled to work on most environments, although it might not be most optimized for your targeted system, but it will work.

When you compile from source, you get more options because you are able to change the flags to do different type of optimization, and the compiler will optimize instructions specifically for your system. Pre-compiled binary don't get to enjoy those type of optimization but the ease of use is a trade-off. You don't have to compile anything because you just need to download them and it will work for your system.

.gz .xz, .tar.gz files

To decompress `.gz` file use the tool `gzip`

To decompress `.xz` file use the tool `unxz`

To decompress `.tar.gz` file use `tar`

SSH Overview

Password SSH login

Typically when you setup a SSH remote server, you would login by entering the remote user's username and its password that you are logging as. The SSH server program will ask OS "I got this username and its password" Can I let him in and be connected, if the credentials are correct, then SSH allows you to be authenticated and be on your way.

Public-key authentication

The other way that the SSH server can authenticate you is via public-key authentication. Specific algorithm can vary but usually is RSA/DSA.

The way it works is that the user who is trying to log into the server will be creating two keys, one public key, and one private key using `ssh-keygen` program.

You can then place the public key into the remote server's `~/.ssh/authorized_keys` file, then when you attempt to connect to SSH with a username + your private key file using the `-i` option, SSH will ask the OS "i got this username and a private key" can he be let in? If yes then SSH will look at your private key to verify that it matches the public key in `authorized_keys` file then you are allowed in.

Specific process of SSH authentication

A secure communication channel has to be first established before authentication. The secure communication is established using symmetric key encryption. This is because asymmetric key pairs are only used for authentication and not encrypting entire connection.

After the secure communication between server and client is established the client must be authenticated to be allowed access. The server will use the client's public key to issue a challenge message to the client, if the client can prove that it is able to decrypt the message, which mean that it has the associated private key. Then server can allow the client in.

known_hosts file

The `known_hosts` file is used to authenticate the servers, just like how `authorized_keys` is used to authenticate users. This file contain the server's public key, and every time you connect to an SSH server, it will show you their public key along with a proof that it has the corresponding private key.

Executing Binary/Script

If binary is relative to your current directory

If the binary/script is relative to your current directory, then you would have to use ./ (dot slash) to execute the binary or script with respect to your current path.

This is because your current directory isn't in your PATH environment variable.

If binary is somewhere else

However, if the binary or script is in another path, then you can execute it by providing the full path without the dot slash, because then it wouldn't make sense, dot slash is used for relative path, if you provide full path and dot slash, then it is saying the binary is in your current directory, so it doesn't make sense.

`/usr/bin/node` would execute the node binary, you would not do `./usr/bin/node`.

Public and Private key encryption/decryption

Cryptography 101

Asymmetric key encryption is key to many things, especially for TLS handshake in HTTPS protocol. How it works is that you first generate a pair of key, one is referred to the public key, and the other is referred to as the private key.

Public and private key just consists of some numbers and uses modular exponentiation to do the actual encryption and decryption. You have several member at play here:

- **e** is the encryption exponent. This is a public value that everybody basically uses the same value, usually 65537
- **d** is the decryption exponent. You will be generating this and need to be kept as a secret, as part of your private key
- **n** is the modulus, same as **e** it is also public and is generated

How these numbers are generated aren't that important to the context of explaining cryptography, but if you would like to know refer to the bottom section

To do encryption, you raise the message to the power of e modulo n

To do decryption, you raise the ciphertext to the power of d modulo n, which will recover the original message

To do signing, you raise the message to the power of d modulo n, which signs the message

To do verification, you raise the signature to the power of e modulo n, which recover the original message verifying that you indeed signed it.

How to get detail about your private key

```
openssl rsa -text -in private_key.pem
```

By running the above command it will output all the number components of your private key which just consists of those numbers we have discussed:

```
Private-Key: (1024 bit)
publicModulus:
    00:e0:ae:82:8f:6a:92:0a:bb:66:95:34:04:e6:82:
    03:9a:fd:93:1d:6c:ed:e7:50:7a:74:da:ba:70:8f:
    f9:a4:b4:16:de:f9:9c:30:bf:15:d5:0e:6d:27:24:
    9f:ea:69:4d:a2:21:22:6e:47:fe:cc:9f:8d:b0:84:
    3f:f3:8e:fc:04:83:44:71:0d:ba:fd:7d:3f:f3:28:
    05:49:1e:47:a9:a7:14:94:57:71:5f:47:4f:4a:54:
    9f:b3:e4:48:6d:28:13:50:48:37:56:8b:33:d5:fa:
    b1:f5:89:b9:a6:16:2f:47:c2:9b:fd:14:0d:d1:ba:
    3a:41:4b:88:88:ed:a8:a1:ef
publicExponent: 65537 (0x10001)
privateExponent:
    26:7d:5e:ba:68:dc:49:e0:5e:ab:72:b4:e0:34:27:
    9f:f6:8e:ac:3c:cb:e8:93:7d:d6:e4:dd:89:88:f0:
    90:49:95:9d:6f:0f:55:be:76:64:00:4b:ac:a7:f6:
    89:36:ae:e8:f6:5a:2a:a0:44:c3:13:16:37:c6:00:
    1a:9e:45:07:c2:af:c7:0b:66:a0:ef:60:01:c1:e1:
    e8:d2:c7:f5:bb:f0:f9:82:3a:67:f8:08:46:1e:76:
    63:29:94:c8:3b:d3:ce:0a:fb:90:84:ce:f8:b2:a5:
    17:2c:73:3e:c4:fd:7f:b1:08:61:be:0b:6c:b3:81:
    f8:50:fe:20:62:09:b0:31
```

The number are in hexadecimal, every two hexadecimal character is separated by a colon for readability.

> One hexadecimal integer can be represented by 4 bits, 2 hexadecimal integer together is 8 bits which is 1 byte. Which is probably why it is separated into groups of 2. And on top of being readable.

You can recover the actual n and d value by just using a simple python script convert the hexadecimal to base 10.

How to do encryption and decryption using openssl?

First you have to generate a public and private key pair by running

```
openssl genrsa -out key.pem
```

The .pem file contain both the private and public key, because remember private key consists of d and n, and public key consists of e and n. The .pem file contain all the numbers.

So to extract out the public key you would run the command:

```
openssl rsa -in key.pem -pubout -out public.pub
```

Finally to encrypt a message run the following command:

```
openssl rsautl -encrypt -inkey public.pub -pubin -in plaintext.txt -out encrypted
```

To decrypt the message run the following command:

```
openssl rsautl -decrypt -inkey key.pem -in encrypted
```

How to sign a message and verify it using openssl?

To sign a message:

```
openssl dgst -sha256 -sign key.pem -out message.sig message
```

This will sign (encrypt with private key), the hashed 256 of the message input and output the signature to the file `message.sig`

To verify a signature:

```
openssl dgst -sha256 -verify pub.pub -signature message.sig message
```

If everything goes well, the message is indeed sent by the sender it will output "Verified OK"

This command basically take the hash of the input file, then verify (decrypt with public key the signed message) and compared whether or not the hash retrieved from signed message is equal to the hash that you took on the message file.

More info please! How are e, d, n generated?

1. First you pick two prime numbers as p and q, any is fine

```
p = 7
q = 13
```

2. Multiply them together

```
n = p * q
n = 7 * 13
n = 91
```

3. Then find the Euler's totient function of n

```
φ(n) = (p - 1) * (q - 1)
φ(91) = (7 - 1) * (13 - 1)
φ(91) = 6 * 12
φ(91) = 72

// φ(n) = (p - 1) * (q - 1) is a special case of the Euler totient function
// For more explanation on the proof of this https://crypto.stackexchange.com/a/5716
```

4. Then pick a random **e** such that it is between φ(n) and 1 and is coprime with φ(n), meaning no common factors between **e** and φ(n)

```
1 < e < φ(91)
1 < e < 72
```

Let's say **e=23**

5. Finally compute **d** which is the modular multiplicative inverse of **e**

```
e^-1 = d (mod φ(n))
23^-1 = d (mod φ(91))
23^-1 = d (mod 72)
23 * d = 1 (mod 72)
23 * 47 = 1 (mod 72)
d = 47
```

Then public key is (n = 91, e=23)

And private key is (n=91, d=47)

Big thanks to https://www.onebigfluke.com/2013/11/public-key-crypto-math-explained.html for simply explaining the math behind asymmetric key generation.

Compressed file vs Archived file

Archived file

An archived file is basically a collection of files and directories stored into one file. You can extract the files and directories out from the archived file. It is one singular file that have all the archived files put into it.

Do note that it uses the same amount of disk space as all the individual files and directories combined, it doesn't do any compression.

Archived file on it's own is not compressed, however, with `tar` you have an option to add compression to it to make the archived file smaller.

Compressed file

Compressed file is a collection of files and directories that are stored into one file and are stored in a way that uses less disk space than all the individual files and directories combined.