Chaobin Tang (唐超斌)

A working understanding on SSL/TLS and HTTPS using Python

SSL is designed against man-in-the-middle attack. Safty is no easy thing. SSL can ensure a secured connection if it is **correctly implemented** (Remember the *heartbleed*?). Right now, the possibly most popular implementation is still OpenSSL. The *ssl* in Python's stdlib is essentially a wrapper around it. It provides a small set of very high level operations. To make use of it, a basic understanding on SSL is important.

(note: The Python code in this article uses Python 3.4)

Are you really you?

Even before you can communicate in a secured channel, there is a more immediate question of security: Am I really talking to the person I want to talk to? How can I be sure of that?

Upon establishing a secure connection, right after sending an initial **client hello**, the client expects the server side to reply back a message that inculdes a *digital certificate*. A digital certificate is essentially a way of proving the server is indeed the server client intends to connect to (or rather server claims whom it is).

Let's have a look at the certificate of one payment provider (the sort of ISPs you would expect an HTTPS channel):

```
import ssl
import urllib
```

```
# Alipay, a trusted payment provider
url = 'https://www.alipay.com'
addr = urllib.parse.urlsplit(url).hostname
port = 443
cert = ssl.get_server_certificate((addr, port), ssl_version=1)
print(cert)
```

This is what it shows:

```
-----BEGIN CERTIFICATE-----
MIIFPTCCBCWgAwIBAgIQNuTYtQBKE6idQDXmiii1UjANBgkqhkiG9w0BAQsFADB+
MQswCQYDVQQGEwJVUzEdMBsGA1UEChMUU3ltYW50ZWMgQ29ycG9yYXRpb24xHzAd
BgNVBAsTFlN5bWFudGVjIFRydXN0IE5ldHdvcmsxLzAtBgNVBAMTJlN5bWFudGVj
IENsYXNzIDMgU2VjdXJlIFNlcnZlciBDQSAtIEc0MB4XDTE0MTIxNjAwMDAwMFoX
DTE1MTIyMTIzNTk1OVowgcsxCzAJBgNVBAYTAkNOMREwDwYDVQQIDAhaSEVKSUFO
RzERMA8GA1UEBwwISEFOR1pIT1UxGzAZBgNVBAoMEkFsaXBheS5jb20gQ28uLEx0
ZDEeMBwGA1UECwwVT3BlcmF0aW9ucyBEZXBhcnRtZW50MUAwPgYDVQQLFDdPcmdh
bml6YXRpb24gYW5kIGRvbWFpbihzKSBhdXRoZW50aWNhdGVkIGJ5IGlUcnVzIENo
aW5hMRcwFQYDVQQDDA53d3cuYWxpcGF5LmNvbTCCASIwDQYJKoZIhvcNAQEBBQAD
ggEPADCCAQoCggEBAJ2lSCMIXh/7VIcNpsJnymDmWLiAWaxuudcb4T58SJJT00Am
0T2ZV9ZFsOMRrYJuyGpEFSh7oXxs6jwdB/JvK1kMrVOcT6LoI7kGktjsKXFhGhrl
KU6yz++2mcvcCb1UFeeGDaei6jTG3J3ymF+lv4o/e8ck5RVinSffgCvMVK6OFxUA
o185M7Dd1AlchfSLZknITF+jGLd4mbir8o/glS3TiiaRXdOBvHN4txWIAIvXfH0Z
oX7T1H9RsK/kY27jFWhz2QQVJi3la4nwMFOI31iCBVVdcIEaLAOqeZM9VjpQWqyh
cZc/vh5DEPh58MMuFyqyenRRSFIx1TDy5NiVUZ0CAwEAAaOCAWcwggFjMBkGA1Ud
EQQSMBCCDnd3dy5hbGlwYXkuY29tMAkGA1UdEwQCMAAwDgYDVR0PAQH/BAQDAgWg
MB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjBlBgNVHSAEXjBcMFoGCmCG
SAGG+EUBBzYwTDAjBggrBgEFBQcCARYXaHR0cHM6Ly9kLnN5bWNiLmNvbS9jcHMw
JQYIKwYBBQUHAgIwGRoXaHR0cHM6Ly9kLnN5bWNiLmNvbS9ycGEwHwYDVR0jBBgw
FoAUX2DPYZBV34RDFIpgKrL1evRDGO8wKwYDVR0fBCQwIjAgoB6gHIYaaHR0cDov
L3NzLnN5bWNiLmNvbS9zcy5jcmwwVwYIKwYBBQUHAQEESzBJMB8GCCsGAQUFBzAB
hhNodHRwOi8vc3Muc3ltY2QuY29tMCYGCCsGAQUFBzAChhpodHRwOi8vc3Muc3lt
Y2IuY29tL3NzLmNydDANBgkqhkiG9w0BAQsFAAOCAQEAhHl+N2K6QlSm4bJN33K5
26g+xfN1ix8ulo+4cYiTb80hN4IN6gRDOR0wYVCP940ioe+DoFrqnv38wOnsMOYO
30mQxU4fV4Pn8pSm0Mi905bTGIVeGUTkHWvpcYQt5xcyQmYmBpnsWt9Ycig5b8xJ
BCgX+HhlbmL69C9I0HkwPhgcYDFIjvC3RptT/VrkQtnODZ75UY6jvNJF7b4Le0Ul
1Clv4kpmK+W5Yz9z67DMTEw7E3lazcbvZASl0hEJJOMgC8e3fg1zcaBrQ2oSZDwe
aUeWXCzgzMrzDwO45Z6hiIhP4oflIQz2J5Urh48Ix8CVBc9zxNnYYW9tLmSh29yR
aw==
-----END CERTIFICATE-----
```

This is a PEM formatted key, the base64 encoded x509 ASN.1 key. Decoding it lets us read it:

```
from OpenSSL import crypto # pip install pyopenssl

cert = crypto.load_certificate(crypto.FILETYPE_PEM, cert)
```

Lots of information now can be read:

```
# The signer of the server's certificate
issuer = cert.get_issuer()
print(issuer.get_components())
```

```
[(b'C', b'US'), (b'O', b'Symantec Corporation'), (b'OU', b'Symantec Trust Network'), (b'CN', b'Symantec Class 3 Secure Server CA - G4
```

```
# The company's business information
subject = cert.get_subject()
components = dict(subject.get_components()) # convert to dict
print(components)
```

```
{b'C': b'CN',
 b'CN': b'www.alipay.com',
 b'L': b'HANGZHOU',
 b'O': b'Alipay.com Co.,Ltd',
 b'OU': b'Organization and domain(s) authenticated by iTrus China',
 b'ST': b'ZHEJIANG'}
```

A **Digital Certificate** is an effective way of assuring one is whom one claims to be. The way it works is to have *certificate authority* (CA) to vouch for one's identity by signing his digital certificate. In other words, the trust is **delegated** to these CAs (Trust is really an interesting thing of our human society). In reality, the CA will imaginably carry out some real world investigations on the identity claim by demanding all sorts of business certificates and other forms of hard evidence. After confirming the identity, CA will digitally sign the certificate (A certificate is a combination of several things, the public key, the hostname that the organization runs its online business on, and other metadata like company name.) so that all the information will NOT be tangled with later on (e.g., you cannot take this signed certificate home and then change the hostname and use it with the new hostname and doing something bad with it, because then your certificate will not be valid and the client won't trust your side of connection).

Upon recieving the digital certificate, the client side of connection will, among other things, check the validity of the certificate, several checks are:

1. Is it expired?
2. Is the hostname being certified the one client is connecting to?
3. Is this certificate ever modified (by signing it with the same algorithm CA uses and compare the two signatures, yes, so you need CA's public keys, you are absolutely right. A set of these public keys of CAs are made public and usually come together with your operating system, or they can be retrieved in places like *here*).
4. Is the CA that signed your certificate one that I trust?

The *ssl* module in Python 3.4 provides almost all APIs that do the above checks.

How are we going to talk securely now?

After verifying the server's certificate is trustworthy, the two sides of connection will negotiate a secret key called **master secret** that will be used to *symetrically encrypt* all future traffic. It's important to note that during negotiating this key, the key itself is encrypted using server's public key, and server will decrypt it using its matching private key (the fact it can decrypt it

further proves its identity). The reason it uses a key to symetrically encrypt the future traffic instead of continuing using the public key encryption, is because asymetric encryption is computationally very expensive. So here is a case of *hybrid cryptosystem*, where the handshake is **key encapsulation**, and future traffic is **data encapsulation**.

Put it altogether in HTTPS

Here is a primitive version of how you can open an HTTPS page in Python:

```
from collections import namedtuple
from http.client import HTTPResponse
import cgi
import socket
import ssl
import urllib

# certifi is a pkg whose sole purpose is distributing Mozilla's CA bundle
import certifi # pip install certifi
ca_certs = certifi.where()

def http(url, secure=443, encoding='utf-8'):
    components = urllib.parse.urlsplit(url)
    path = '%s' % components.path if components.path else '/'
    HTTPS = (components.scheme == 'https')
    addr = components.hostname
    port = secure if HTTPS else 80
    CRLF = '\r\n\r\n'

    Page = namedtuple('Page', ('code', 'headers', 'body'))

    def handshake(sock):
        # this will trigger the handshake
        # if sock is already connected
        new_sock = ssl.wrap_socket(sock,
               ciphers="HIGH:-aNULL:-eNULL:-PSK:RC4-SHA:RC4-MD5",
               ssl_version=ssl.PROTOCOL_TLSv1,
               cert_reqs=ssl.CERT_REQUIRED, # I will verify your certificate
               ca_certs=ca_certs # using a list of trusted CA's certificates
               )
        return new_sock

    def make_header():
        headers = [
            'GET %s HTTP/1.1' % (path),
```

```
            'Host: %s' % addr,
            'User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)',
            'Charset: %s' % encoding
        ]
        header = '\n'.join(headers)
        header += CRLF
        return bytes(header, encoding=encoding)

    def parse_response(resp):
        '''
        resp
            http.client.HTTPResponse, not closed
        '''
        resp.begin()
        code = resp.getcode()
        # assume status 200
        headers = dict(resp.getheaders())
        def get_charset():
            ctt_type = headers.get('Content-Type')
            return cgi.parse_header(ctt_type)[1].get('charset', encoding)
        body = resp.read()
        body = str(body, encoding=get_charset())
        return Page(code, headers, body)

    def request(header):
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        sock.connect((addr, port))
        try:
            if HTTPS: sock = handshake(sock)
            sock.sendall(header)
            # receive it
            resp = HTTPResponse(sock)
            return parse_response(resp)
        finally:
            sock.shutdown(1)
            sock.close()

    return request(make_header())
```

Of course, real world Python programer wouldn't do this. Instead, a more attractive version would be this:

```
import requests # pip install requests
```

```
resp = requests.get(url)
```

Other security concerns

The handshake explained above skipped many other things.

Randomness

One of them is the few random bytes exchanged between client and server, which is used to derive, among many other things, the master key explained above. So its approximity to randomness is really important. In lots of programming languages, you can seed your random number generator with more sources. In some mordern OS, the random number generator can have more than 100 sources.

Replay attack

What if someone captures whatever is going between you and the server (e.g., your router can, your ISP can) and later *replay* it? SSL comes to rescue again. SSL builds this protection into itself by design. A one-time off session ID is generated to protect you. So when the attacker replays your session, the server will refuse to talk with him because for the new conversation, a new session is used, and the replayed traffic using previous session ID is untrusted. However, it is a bigger picture here with session ID. With what's explained above about handshake having lots of overhead due to public key algorithms, an efficient web application will have a hard time trading off between security and performance. So new techniques are explored to avoid a **full handshake** by **resuming** a previous session. *Here* is an excellent article that explains this technique.

Email: cbtchn[at]gmail.com