Certificates x509 and RSA

X.509 is an ITU-T standard for public key infrastructure. X.509v3 is defined in **RFC 5280** (which obsoletes **RFC 2459** and **RFC 3280**). X.509 certificates are commonly used in protocols like TLS.

- Creating a Certificate Signing Request (CSR)
- Creating a self-signed certificate
- Determining Certificate or Certificate Signing Request Key Type
- SignedCertificateTimestamp
- Version
- LogEntryType
- SignatureAlgorithm

Practical examples: We load the permissions in python and the data which is based on byte iterations. X509 certificates are signatures in computer files and some practical applications.

```
  File "<stdin>", line 3
    -> Certificate:
    ^
SyntaxError: invalid syntax
>>>     return rust_x509.load_der_x509_certificate(data)
  File "<stdin>", line 1
    return rust_x509.load_der_x509_certificate(data)
IndentationError: unexpected indent
>>> cert = x509.load_pem_x509_certificate(data)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'data' is not defined
>>> cert = x509.load_pem_x509_certificate(data:bytes)
  File "<stdin>", line 1
    cert = x509.load_pem_x509_certificate(data:bytes)
                                          ^
SyntaxError: invalid syntax
```

```
>> from cryptography import x509
>>> from cryptography.x509.oid import NameOID
>>> from cryptography.hazmat.primitives import hashes

csr = x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
...    # Provide various details about who we are.
...    x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
...    x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"California"),
...    x509.NameAttribute(NameOID.LOCALITY_NAME, u"San Francisco"),
...    x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"My Company"),
...    x509.NameAttribute(NameOID.COMMON_NAME, u"mysite.com"),
... ])).add_extension(
...    x509.SubjectAlternativeName([
...        # Describe what sites we want this certificate for.
...        x509.DNSName(u"mysite.com"),
...        x509.DNSName(u"www.mysite.com"),
...        x509.DNSName(u"subdomain.mysite.com"),
...    ]),
...    critical=False,
... # Sign the CSR with our private key.
... ).sign(key, hashes.SHA256())
>>> # Write our CSR out to disk.
>>> with open("path/to/csr.pem", "wb") as f:
...    f.write(csr.public_bytes(serialization.Encoding.PEM))
```

Now we can give our CSR to a CA, who will give a certificate to us in return.>

Creating a self-signed certificate

While most of the time you want a certificate that has been *signed* by someone else (i.e. a certificate authority), so that trust is established, sometimes you want to create a self-signed certificate. Self-signed certificates are not issued by a certificate authority, but instead they are signed by the private key corresponding to the public key they embed.

One of the most important things to work with certificates and hashes is the byte exponent and the key size. At the time we want to create a private RSA key pair we need to submit our methods and our exponent 65537. Another important point is the serialization of data chunks.

We have to serialize data for streaming performance and byte allocation out of the CPU memory. The network traffic shows how to serialize and automate bytes in streaming.

```
...     size=2048,
... )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: generate_private_key() got an unexpected keyw
ord argument 'public'
>>> key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
>>> # key into disk safe
>>>
```

Hashing algorithms and SSL certificates

65537 is commonly used as a public exponent in the RSA cryptosystem. Because it is the Fermat number $F_n = 2^{2^n} + 1$ with $n = 4$, the common shorthand is "F_4" or "F4".[3] This value was used in RSA mainly for historical reasons; early raw RSA implementations (without proper padding) were vulnerable to very small exponents, while use of high exponents was computationally expensive with no advantage to security (assuming proper padding).

65537 is also used as the modulus in some Lehmer random number generators, such as the one used by ZX Spectrum, which ensures that any seed value will be coprime to it (vital to ensure the maximum period) while also allowing efficient reduction by the modulus using a bit shift and subtract.

```
You can run some tests of the hashrat youve: No such file or directory
-e
 ################ Testing Hash Types
-e   OKAY    md5 Hashing works
-e   OKAY    sha1 Hashing works
-e   OKAY    sha256 Hashing works
-e   OKAY    sha512 Hashing works
-e   OKAY    whirlpool Hashing works
-e   OKAY    jh224 Hashing works
-e   OKAY    jh256 Hashing works
-e   OKAY    jh384 Hashing works
-e   OKAY    jh512 Hashing works
-e
 ################ Testing Repeated Iterations (may take some time)
-e   OKAY    1000 md5 works
-e   OKAY    1000 sha1 works
-e   OKAY    1000 whirlpool works
-e   OKAY    1000 jh384 works
-e
 ################ Testing Encoding
-e   OKAY    base 8 (octal) encoding works
-e   OKAY    base 10 (decimal) encoding works
-e   OKAY    UPPERCASE base 16 (HEX) encoding works
-e   OKAY    base 64 encoding works
-e   OKAY    uu-encode style base 64 encoding works
-e   OKAY    xx-encode style base 64 encoding works
-e   OKAY    'website compatible' base 64 encoding works
-e   OKAY    ASCII85 encoding works
-e   OKAY    ZEROMQ85 encoding works
-e
```

Note: Hashing algorithms is try to guess byte exponents ratio in a hash format such MD5 or SHA

Here I am catching credentials from one user to show how powerful is hashing the security algorithms and decrypt certificates:

I am going to use my directory, my key log server which is in ubuntu and my key box with a gpb. The server key-signatures are created from the public key that I want to create and decrypt in DNS format.

Here is an example in which I show how easy is to hack the DNS server keys:

```
sudo: pkg: command not found
victor88@penguin:~$ gpg --keyserver keyserver.ubuntu.com --recv-keys 71A3B16735405025D44
7E8F274810B012346C9A6
gpg: directory '/home/victor88/.gnupg' created
gpg: keybox '/home/victor88/.gnupg/pubring.kbx' created
gpg: key 74810B012346C9A6: 8 duplicate signatures removed
gpg: /home/victor88/.gnupg/trustdb.gpg: trustdb created
gpg: key 74810B012346C9A6: public key "Wladimir J. van der Laan <laanwj@protonmail.com>"
 imported
gpg: Total number processed: 1
gpg:               imported: 1
victor88@penguin:~$
```

Here I am cracking passwords in an easy way across programming languages calculating the hash key processing per character:

```
        2: from (irb):2
        1: from /usr/lib/ruby/2.7.0/digest.rb:16:in `con
st_missing'
LoadError (library not found for class Digest::ND5 -- di
gest/nd5)
irb(main):003:0> puts Digest::MD5.hexdigest '7cee9cb3ae1
849bcf1e0d84f0b8f5591'
a4bebd803b73700d47bc089fd950b0ac
=> nil
irb(main):004:0> puts Digest::MD5.hexdigest 'a4bebd803b73700d47bc089fd950b0ac'
4ba2db0c7b1b32c3103e013c0bbb525d
=> nil
irb(main):005:0>
```

These hashes come from web servers:

But we can test for a summary of the entire MD5 hash. Many servers use MD5:

```
date: ???; 03 Jan 2023 13:03:40 GMT
content-type: text/html; charset=UTF-8
cross-origin-embedder-policy: require-corp
cross-origin-opener-policy: same-origin
cross-origin-resource-policy: same-origin
permissions-policy: accelerometer=(),autoplay=(),camera=
(),clipboard-read=(),clipboard-write=(),geolocation=(),g
yroscope=(),hid=(),interest-cohort=(),magnetometer=(),mi
crophone=(),payment=(),publickey-credentials-get=(),scre
en-wake-lock=(),serial=(),sync-xhr=(),usb=()
cf-mitigated: challenge
cache-control: private, max-age=0, no-store, no-cache, m
ust-revalidate, post-check=0, pre-check=0
expires: Thu, 01 Jan 1970 00:00:01 GMT
report-to: {"endpoints":[{"url":"https:\/\/a.nel.cloudfl
are.com\/report\/v3?s=6jO6J1aP1OLdNSy13tVKwj9rqqzhb6pZkH
RsOFI9Ha36hsbjJjpebeQfhddOaWYlCN1Htpru9FpuXwSjMCYlA98Nmz
VzU8XFmhqmuc75Bn0uJEYDUvD6qkUiVxs%3D"}],"group":"cf-nel"
,"max_age":604800}
nel: {"success_fraction":0,"report_to":"cf-nel","max_age
":604800}
expect-ct: max-age=86400, enforce
referrer-policy: same-origin
x-content-type-options: nosniff
x-frame-options: SAMEORIGIN
x-xss-protection: 1; mode=block
server: cloudflare
cf-ray: 7d49939c8ebe2f95-MAD
```