Using Cryptography

CMSC 414

October 16, 2017

Digital Certificates

Recall:

$K_{\mathrm{pub}} = (n, e) \Leftarrow$ This is an RSA public key

How do we know who this is for?

Need to bind *identity* to a *public key*

We can do this using a **Digital Certificate**

This is a binding that has been *signed* by some third party

- Often a *trusted third party*
- Sometimes *self-signed*

What's in a Digital Certificate?

```
$ openssl x509 -in mmarsh.req.cert -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1428829381 (0x552a34c5)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=CA, OU=CA, O=soucis
        Validity
            Not Before: May  5 20:29:24 2017 GMT
            Not After : Jan 30 20:29:24 2020 GMT
        Subject: O=soucis, OU=user, CN=Michael Marsh
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (4096 bit)
                Modulus (4096 bit):
                    00:fe:e2:a3:4c:1c:63:1a:f2:aa:d3:70:bd:d2:8c:
                    ...
                    4f:a6:0c:ef:b3:f6:9c:46:65:94:9f:03:45:73:64:
                    e0:ff:f7
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Authority Key Identifier:
                keyid:CE:6A:A6:71:63:CF:58:0B:F1:25:E2:B6:5C:0E:AD:73:51:3E:D6:E7

            X509v3 Subject Key Identifier:
                DF:69:B6:41:CF:FC:21:25:C8:5D:CC:A7:89:A2:C8:3F:92:00:66:3D
    Signature Algorithm: sha256WithRSAEncryption
        6b:10:11:19:fc:e7:d4:0a:b8:67:58:c4:f8:97:99:51:76:60:
        ...
        45:a0:b0:64:ff:f8:3f:97:ec:22:23:74:bc:61:0a:a3:b3:cf:
        08:ab:ee:29
```

How Do We Know if This is Valid?

Certificates have an **Issuer**

The issuer signs the certificate, with an **validity period**

We then need the certificate for the *issuer*

How do we know *that* certificate is valid...?

[Note: Slides marked with an asterisk contain cynicism]

Certification Authorities (*)

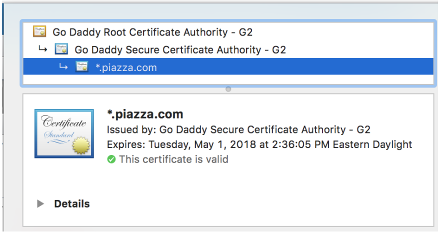
A *Root Certification Authority* (root CA) is a **Root of Trust**

That means we trust it implicitly

CAs issue certificates to *users*, *servers*, or *intermediate CAs*

Usually, a root CA only certifies intermediate CAs

What we end up with is a **Certificate Chain**



Your browser is pre-loaded with a *lot* of root CAs, and you trust them, whether you really do or should

What Does a CA Do? (*)

A CA *should* check that the name in a *certificate request* matches the principal *sending* the request

If the certificate is for a service, the CA *should* check that this is, in fact, the public key for that service, and the requester *actually represents* the service

This usually works, but...

Rogue CAs might sign bogus certificates

Some CAs only provide this level of verification for higher-paying customers, and issue *less-secure* certificates for others

What Does a Certificate Request Look Like?

```
$ openssl req -in ~/Downloads/XXXX.req -noout -text
    Certificate Request:
        Data:
            Version: 0 (0x0)
            Subject: O=soucis, OU=user, CN=XXXX
            Subject Public Key Info:
                Public Key Algorithm: rsaEncryption
                RSA Public Key: (4096 bit)
                    Modulus (4096 bit):
                    00:a8:fe:4a:3e:d0:4e:d1:ad:93:b3:76:fe:c1:78:
                    ...
                    65:17:36:d9:49:22:9a:c9:45:79:e5:14:9f:bd:ed:
                    a1:0d:8d
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha1WithRSAEncryption
        48:cb:42:38:f7:d0:2a:d7:8d:95:96:20:60:ae:19:9d:82:ac:
        ...
        91:0c:66:b5:4a:92:4b:ec:f1:41:59:ee:47:0e:9f:c7:b7:05:
        08:e9:1e:1d:83:1c:2b:32
```

Unlike the certificate, this is signed with the *subject's* private key, not the *issuer's*

PGP, GPG, and the Web of Trust

Pretty Good Privacy, *GNU Privacy Guard*

Similar idea, but *without a root of trust*

Instead of relying on a third party we don't know...

1. Alice and Bob meet (possibly at a *key-signing party*)
2. They exchange public keys
3. They sign those keys and hand each other the certs

Anyone with a cert for Alice can validate Bob's cert from Alice

If Bob has a cert signed by Alice, we sometimes refer to him as "Alice's Bob" when he uses this cert

Forms a **Web of Trust** that can connect users

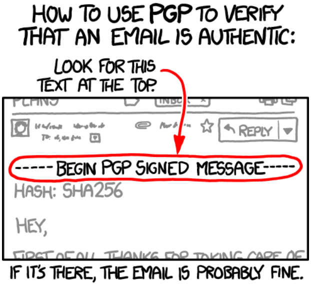
Requires ability to form a *trusted path* from verifier to subject

How Do We Use Certificates? (*)

CA-based certificates are commonly used on the web for encrypted connections (HTTPS)

PGP-based certificates are “commonly” used in email for signing messages

Requires the mail client to have a PGP/GPG-capable extension



Public Key Infrastructure (*)

Certificates are only one component

We also need (this is not the precise technical specification)

- Distribution mechanism for CA certificates
- **Revocation Lists**

A *Revocation* is a CA-issued statement that a certificate is *no longer valid* (other than due to expiry)

⇒ This can be due to issues like private key disclosure

⇒ Your browser almost certainly never checks for these

With these, we have a **Public Key Infrastructure** (PKI)

Group Exercise 1

In your `crypto-exercises` repository run `git pull upstream master` to merge changes to the upstream repository that you initially forked.

"Using Crypto Exercise 1" contains instructions for setting up and using your own OpenSSL-based Certification Authority.

Symmetric and Asymmetric Crypto

Notation

To keep things sane, let's define

k_{AB}	a *secret key* shared by A and B
K_A	the *public key* of A
k_A	the *private key* of A
$\{P\}_{k_{AB}}$	symmetric-key encryption of plaintext P with k_{AB}
$E_A(P)$	asymmetric-key encryption of plaintext P with K_A
$D_A(C)$	asymmetric-key decryption of ciphertext C with k_A
$S_A(M)$	message M signed with k_A

The short version:

- $\{P\}_k$ is symmetric-key encryption
- $E()$, $D()$, and $S()$ are asymmetric-key operations

Symmetric and Asymmetric Crypto

Key Sizes and Security

From NIST publication SP 800-57 Part 1 Rev. 4 (January 2016), by Elaine Barker

Security Strength	**Symmetric Cryptosystem**	**RSA Key Length**	**ECC Key Length**	**Hash Functions**
80	2-key 3DES	1024	160–223	SHA-1
112	3-key 3DES	2048	224–255	SHA-224
128	AES-128	3072	256–383	SHA-256
192	AES-192	7680	384–511	SHA-384
256	AES-256	15360	512+	SHA-512

Symmetric and Asymmetric Crypto

Performance

3DES and AES can be implemented in *hardware* $\Rightarrow$ very fast

software implementations slower, but still fairly fast using lookup tables

RSA, ElGamal, ECC must be implemented in software, and private key operations are fairly slow

Symmetric and Asymmetric Crypto

Getting the Best of Both Worlds

A simple example:

@A:

1. select random nonce n_A
2. $M_1 = E_B(n_A)$
3. $s_1 = S_A(M_1)$

$A \rightarrow B$: $\langle M_1, s_1 \rangle$

@B:

1. $V_A(M_1, s_1)$
2. compute $n_A = D_B(M_1)$
3. select symmetric key k_{AB}
4. compute $n_B = n_A \oplus k_{AB}$
5. $M_2 = E_A(n_B)$
6. $s_2 = S_B(M_2)$

$B \rightarrow A$: $\langle M_2, s_2 \rangle$

@A:

1. $V_B(M_2, s_2)$
2. compute $n_B = D_A(M_2)$
3. compute $k_{AB} = n_B \oplus n_A$

Group Encryption

Sometimes we want to send encrypted data to a large number of principals

- Videoconferencing
- Premium cable channels
- Subscription services

Public keys $\Rightarrow$ too many, takes too long to compute them all

Secret keys $\Rightarrow$ too many, expensive key exchange (though doesn't have to be done often)

Group Encryption

Both result in one of

- lots of individual messages (can't take advantage of efficient delivery mechanisms)
- extremely long messages that contain lots of copies of encryption
- extremely long messages that contain one encrypted message and lots of encryptions of the message key

Must be a better way...

Group Keys

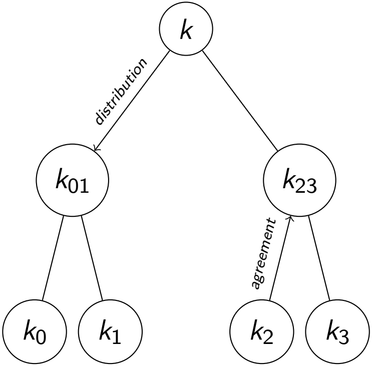
We would like to have a *single* symmetric key for a stream, that can be shared with everyone

This could be distributed by a central authority individually to every user, or agreed upon/determined by the set of users

A common way to do either of these is with **Key Trees**

There are many ways to do this, all of which share common issues:

- Key distribution/agreement needs to be fairly efficient
- Only legitimate group members should be able to learn the key
- Keys must be changed when members *join* or *leave* the group



Group Exercise 2

Now you're going to combine public-key and secret-key cryptography for yourself. See "Using Crypto Exercise 2" in the README. The instructions assume that you're using python, but you may use any language you like.