# SSL AND PUBLIC KEY ENCRYPTION

Stephen Schaub

# Topics



- HTTPS
- SSL
- Public Key Encryption
- Public Key Certificates

# About HTTPS



- HTTPS – Secure HTTP
- Encrypts HTTP traffic between web browser and web server

# Using HTTPS



- Create public key certificate
- Install on web server
- Instruct (or redirect) clients to use https: URL to access application

# Web Encryption Standards



- SSL – Secure Sockets Layer
- TLS – Transport Layer Security
- HTTPS – Secure HTTP

# SSL and TLS

## SSL

- Secure Sockets Layer
- Encrypts network communications
- Invented by Netscape in 1990's
- Generally, no longer used

## TLS

- Transport Layer Security
- "Rebranded SSL" / "Next Generation SSL"
- Often called SSL

# SSL/TLS History

SSL was essentially **renamed** to TLS after SSL 3.0

| Year | Version |
|---|---|
| 1995 | SSL 2.0 |
| 1996 | SSL 3.0 |
| 1999 | TLS 1.0 (the "next version" of SSL) |
| 2006 | TLS 1.1 |
| 2008 | TLS 1.2 |

# SSL/TLS Features

SSL (and TLS) offer

- Confidentiality (privacy)
- Integrity (assurance the data has not been altered)
- Authentication (confirmation of who sent the message)

*Note: For the rest of this presentation, I will use "SSL" to refer to SSL/TLS*

# How SSL Works

SSL uses different two encryption mechanisms:

- **Secret key encryption** (aka symmetric encryption) is used to encrypt most of the traffic
- **Public key encryption** is used for message authentication and to exchange secret keys securely

# Secret Key Encryption

- Relies on both parties knowing a shared secret key
  - A key is a large number (ex. 2048 bits)
- Common secret key algorithms used in various versions of SSL:

- DES
- Triple DES
- AES
- RC2
- RC4
- IDEA
- Fortezza
- Camellia

# Public Key Encryption

- Each party has a pair of keys:
  - A private key known only to the owner
  - A public key shared with everyone
- Messages encrypted with a public key can be decrypted only with the paired private key
  - ... and messages encrypted with a private key can be decrypted only with the paired public key
- Public key encryption algorithms used in SSL: RSA, DSA

# Public Key Scenarios

- Send a private message
- Send an authenticated message
- Send a private, authenticated message

# Sending a Private Message



Alice wants to encrypt and send a message to Bob

1. Alice encrypts plaintext with Bob's public key
   - ciphertext = rsa(plaintext, Bob-PubKey)
2. Alice sends ciphertext to Bob
3. Bob decrypts ciphertext with Bob's private key
   - plaintext = rsa(ciphertext, Bob-PriKey)

Note: Same algorithm (here, "rsa()") used to either encrypt or decrypt

# Sending an Authenticated Message



Alice wants to publish a message publicly that everyone knows came from her

1. Alice encrypts plaintext with Alice's private key
   - ciphertext = rsa(plaintext, Alice-PriKey)
   - ciphertext is essentially a *digitally signed message*
2. Alice publishes ciphertext
3. Anyone who has Alice's public key can decrypt message
   - plaintext = rsa(ciphertext, Alice-PubKey)

# Sending a Private, Authenticated Message



Alice wants to send a message securely to Bob that Bob knows had to come from her

1. Alice encrypts plaintext with _____ key
   - ciphertext = rsa(plaintext, _____)
2. Alice encrypts ciphertext with _____ key and transmits to Bob
   - auth_ciphertext = rsa(ciphertext, _____)
3. Bob decrypts auth_ciphertext with _____ key
   - ciphertext = rsa(auth_ciphertext, _____)
4. Bob retrieves original plaintext using _____ key
   - plaintext = rsa(ciphertext, _____)

# Sending a Private, Authenticated Message



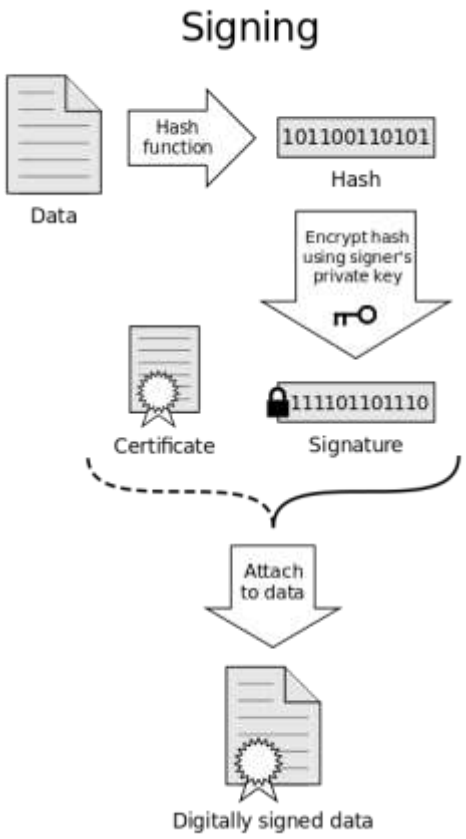
Alice wants to send a message securely to Bob that Bob knows had to come from her

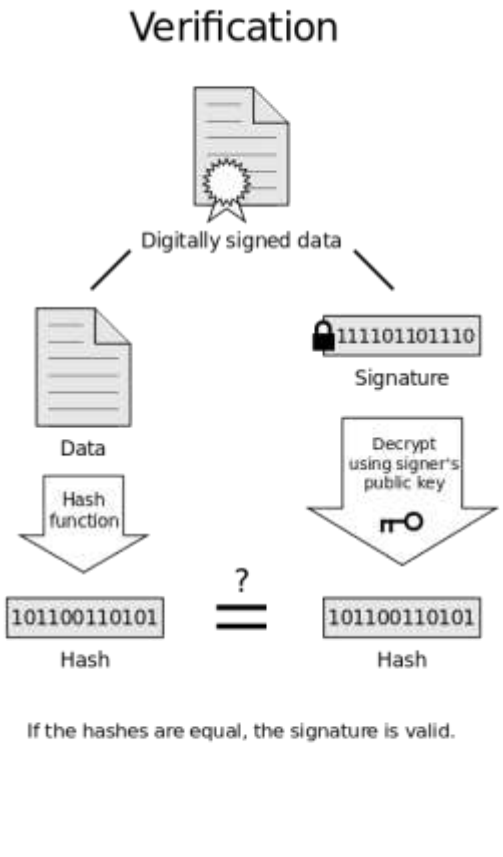
1. Alice encrypts plaintext with Bob's public key
   - ciphertext = rsa(plaintext, Bob-PubKey)
2. Alice encrypts ciphertext with Alice's private key and transmits to Bob
   - auth_ciphertext = rsa(ciphertext, Alice-PriKey)
3. Bob decrypts auth_ciphertext with Alice's public key
   - ciphertext = rsa(auth_ciphertext, Alice-PubKey)
4. Bob retrieves original plaintext using Bob's private key
   - plaintext = rsa(ciphertext, Bob-PriKey)

# Digital Signatures



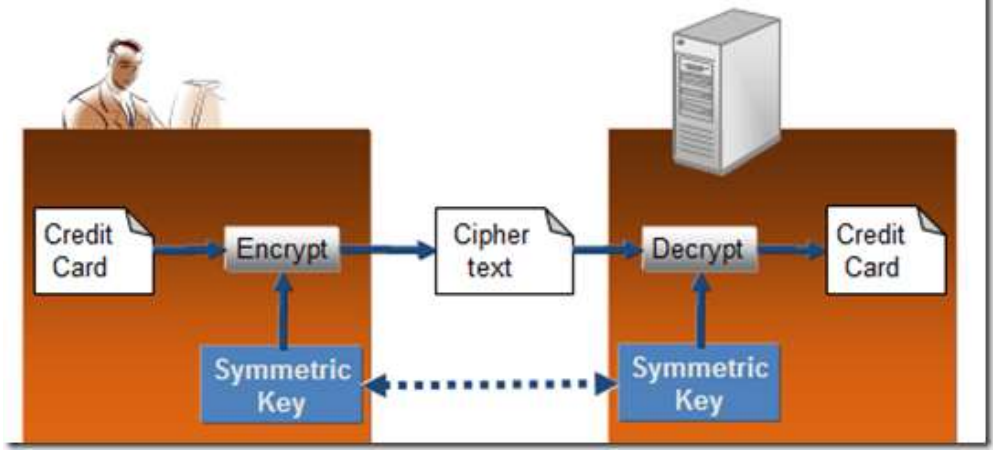
- Public key encryption algorithms are slow
- For efficiency, SSL uses cryptographic hashing algorithms to verify message integrity and digital signatures
- Common cryptographic algorithms for SSL:
  - SHA-1, SHA-2, SHA-3



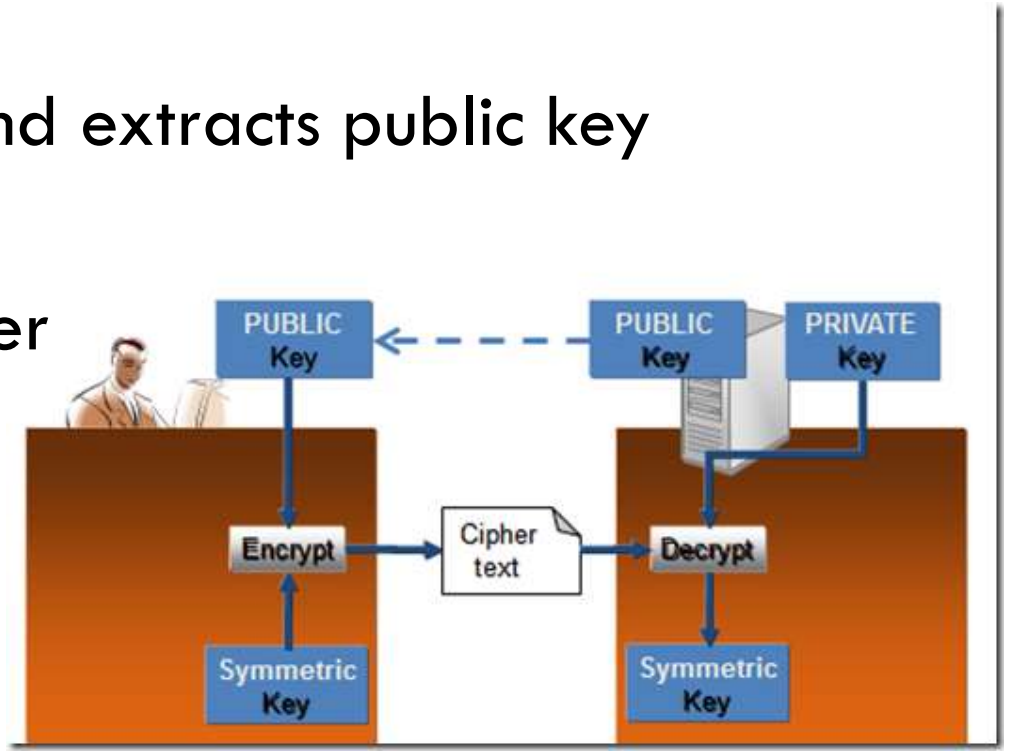
# How SSL Works

1. Client encrypts data using a symmetric encryption algorithm and shared secret key ("session key")
2. Client computes and appends a MAC and transmits message containing data + MAC to server
   - MAC = cryptographic-hash(data + session key)
3. Server decrypts data using session key
4. Server computes MAC on decrypted data and compares to MAC to verify message integrity



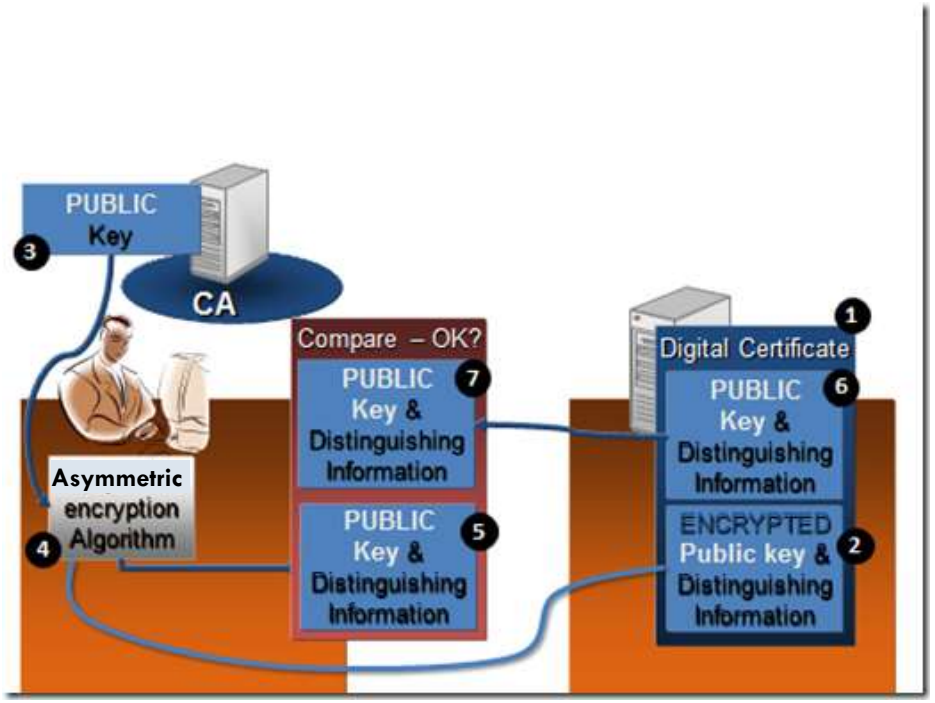
# Sharing a Secret Key

1. During initial handshake, Client generates secret key ("session key") using random number generator
2. Client obtains web server's certificate and extracts public key
3. Client encrypts session key with web server's public key and transmits to server
4. Server decrypts session key using its private key



# SSL Authentication

1. Certificate contains unencrypted data and a digital signature created by CA
2. Client verifies CA's digital signature using CA's certificate



# SSL Overhead



- SSL imposes overhead of up to 50% when using SSL.
  - Overhead largely due to the handshaking necessary to initialize the first SSL connection between a browser and a server. It only affects the first page retrieved from a server.
  - Subsequent pages reuse the same session key.
- The data itself is encrypted using fast symmetric encryption algorithms; thus, large amounts of data do not impose much overhead.

# Certificate Management

# Certificates

- Certificates associate a server or organization name with a public key
- Contain
  - Public key
  - Naming information
  - Other attributes
  - Digital signature
- Signature asserts that the named party owns the specified public key
  - Signature = rsa(sha-1(certificate-data), CA-Private-Key)

# Certificate Data

| Field | Description |
| --- | --- |
| Serial Number | A unique integer assigned by the certificate issuer |
| Signature algorithm | Specifies algorithm used to sign certificate |
| Issuer | The DN of the certificate signer |
| Validity | Date range certificate is valid |
| Subject | The DN of the certificate owner (server or organization name) |
| Subject Public Key | Public key of certificate owner |
| Extensions | Additional fields |
| Signature | Digital signature created by issuer |

# Example Certificate

```
Certificate:
  Data:
    Version: 1 (0x0)
    Serial Number: 7829 (0x1e95)
    Signature Algorithm: md5WithRSAEncryption
    Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
        OU=Certification Services Division,
        CN=Thawte Server CA/emailAddress=server-certs@thawte.com
    Validity
      Not Before: Jul  9 16:04:02 1998 GMT
      Not After : Jul  9 16:04:02 1999 GMT
    Subject: C=US, ST=Maryland, L=Pasadena, O=Brent Baccala,
        OU=FreeSoft, CN=www.freesoft.org/emailAddress=baccala@freesoft.org
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public Key: (1024 bit)
        Modulus (1024 bit):
          00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
          33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
          66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
          70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
          16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
          c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
          8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
          d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
          e8:35:1c:9e:27:52:7e:41:8f
        Exponent: 65537 (0x10001)
  Signature Algorithm: md5WithRSAEncryption
    93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
    92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
    ab:2f:4b:cf:0a:13:90:ee:2c:0e:43:03:be:f6:ea:8e:9c:67:
    d0:a2:40:03:f7:ef:6a:15:09:79:a9:46:ed:b7:16:1b:41:72:
    0d:19:aa:ad:dd:9a:df:ab:97:50:65:f5:5e:85:a6:ef:19:d1:
    5a:de:9d:ea:63:cd:cb:cc:6d:5d:01:85:b5:6d:c8:f3:d9:f7:
    8f:0e:fc:ba:1f:34:e9:96:6e:6c:cf:f2:ef:9b:bf:de:b5:22:
    68:9f
```

# Certificate Creation

1. Generate public / private key pair
2. Create Certificate Signing Request (CSR)
3. Get CSR signed:
    1. By a public CA
    2. By an organizational CA
    3. Self sign

- letsencrypt.org offers free 1-year certificates

# Further Reading

- SSL and TLS: Theory and Practice
  Rolf Oppliger
- https://blogs.msdn.microsoft.com/plankytronixx/2010/10/28/crypto-primer-how-does-ssl-work/
  A nice primer (We borrowed its illustrations)

# Certificate File Formats

X.509 Certificates and RSA keys can be stored in files using

- DER format (binary)
- PEM format (Base64 encoded DER)
- PKCS12 format (Microsoft)
- JKS KeyStore format (Java)

# Certificate Files

| Extension | Format | Contents |
|---|---|---|
| .csr | PEM | Certificate signing request |
| .pem | PEM | One or more certificates and/or public/private keys |
| .key | PEM | Private key |
| .cert, .cer, .crt | PEM or DER | One or more certificates and/or public/private keys |
| .p7b | PEM | One or more certificates. Never contains a private key. |
| .pfx, .p12, .pkcs12 | PKCS12 | One or more certificates and/or public/private keys<br>Common on Windows |
| .jks | JKS | One or more certificates and/or public/private keys (Java) |

# PEM Example (Certificate Request)

```
-----BEGIN CERTIFICATE REQUEST-----
MIIERDCCAywCAQAwZDELMAkGA1UEBhMCVVMxEjAQBgNVBAgTCUthcm5hdGFrYTES
MBAGA1UEBxMJQmFuZ2Fsb3JlMQswCQYDVQQKEwJNUzEMMAoGA1UECxMDQ1NTMRIw
EAYDVQQDEwlhbmdlbC0yazMwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIB
AAAAADANBgkqhkiG9w0BAQUFAAOCAQEANR/QwFBkvXx7WVlnGWpsZNjMyNoBuwsP
Wmjwu2FQ90+TSGexY0NI6cS1Xc9E0NlFuONcxJjaLclcW4Ptz1IpEUzK6t1CYV5q
zJnyt7Fb2d6qY4Is6wrWo9IGOA0G814oxk8oMbBIXsjTZaE6JRW2NUts3lHSlgEY
E1POkVex84jbmmIhJlqyBlSLH3d6rRYy8WaXMkaUTSBlp6vb3ealIsu5YTKtE1YW
9BYv1MHhVVIXoGts10y9s/NRrdVqDnVjgdYR+bjZaxbIca5loyYaMRCUBzFFIC7F
W80lqPN3EcpySUoZbdDBM8R5M6sGWIbiagwToVMkx1KNpNA3lxYh5g==
-----END CERTIFICATE REQUEST-----
```