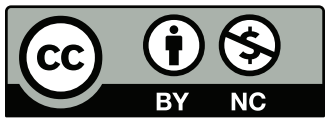
# SSL: Secure Socket Layer



# Choices in Key Exchange

- We have two basic ways to do key exchange, public key (with PKI or pki) or KDC
- Which is better?
- What are the properties of each?

# PKI/pki

- Offline issuance of credentials (but are CAs really offline?)
- Computationally expensive
- CA server does not see private key; compromise of server can lead to credential forgery, but does not expose past conversations
- Suitable for offline use
- However — revoking a credential requires either online operation or a CRL, and CRLs do not provide real-time revocation
- Availability problem can mask revocation

# KDC

- Online only
- Computationally cheap
- Compromise of server does expose past converations. (Look back at the Needham-Schroeder protocol or at Kerberos and convince yourselves of that. . . )
- Revocation is instant
- Availability an issue if KDC is unreachable

# "It Depends"

- Do you need offline operation?
- Can you afford the CPU time, especially on the server side?
- What is the (perceived) risk of KDC compromise?
- How fast does revocation have to be?

# Case Study: SSL

- Security for the web
- What should be protected?
- What should be deployed? What could be deployed?
- How?
- Caution: SSL is quite complex. I'm going to teach a subset of it — and even it is complex.

# Security for the Web

- Imagine 1995
- The web is new. Graphical interfaces — web browsers — are new, and newly commercialized. (Netscape commercialized Mosaic, the first graphical browser.)
- (Windows machines were rare online; Windows 95 was the first version that included TCP/IP.)
- There was *no* e-commerce.
- One hold-up: security
- How could e-commerce be protected?

# Choices

- Public key or symmetric crypto?
- Protect *transmissions* or protect *transactions*?
- To encryption transmissions, at what layer should it be done?
- Encrypt credit cards or avoid using them?

# Encrypting Transactions

- Could encrypt just the sensitive portion of the purchase
- But — the expensive part is the key setup; symmetric encryption is almost free
- Can we properly define “sensitive”?
- It would be nice if customers digitally signed their purchases — but no consumers had key pairs or certificates then

# Payment Scheme

- It would be nice if payment could be done without credit cards, say by binding a certificate to a credit card
- But — no one had such certficates, and very few people would get them
- Why should they? There was very little e-commerce, and most people hadn't (and haven't) heard of the concept
- Netscape didn't want to get into the banking business (and probably couldn't, for legal and practical reasons)
- Conclusion: *only* feasible choice is encrypted credit card numbers

# Layers

- At what layer of the network stack should the crypto be done?
- Network layer — but then Netscape would have to touch many different kernels
- Transport layer — same problem.
- (Note: by this time, there were already draft DoD and ISO standards for encryption at these layers. The concept was known, but not implemented.)
- HTML layer — useful only for HTML, and in 1995 it wasn't clear that that would be the winner
- Ultimate decision: provide protection immediately above the socket layer

# Encryption Above TCP

- TCP provides a reliable byte stream
- Can use stream cipher on top of it
- But — suppose the enemy forges a TCP packet
- The receiving TCP will accept it; the authentic version will be rejected as a duplicate
- Or — the attacker can forge a TCP RST or FIN and tear down the connection
- A MAC can detect forged data, but the sender and receiver are out of synchronization

# Encryption Below TCP

- Can discard forged packets before TCP processing
- Never seen by TCP; no ACK segment sent
- Ordinary TCP retransmission by the sender will repair the problem
- Some difficulty doing key exchange protocol at this layer
- Other than that, a good choice, but only available to OS vendors, not Netscape

# SSL: Secure Socket Layer

- General-purpose encryption and authentication package layered on top of TCP
- Optimized for web use
- Version 1.0 was never released. Version 2.0 was used extensively, but had cryptographic flaws
- Version 3.0 is in use today, and is believed to be secure
- The IETF version is TLS — Transport Layer Security — and is actively being enhanced up to the present. Version 1.1 is in use today; extensions to it and a new version, 1.2, are being deployed.

# Design Principles

- Requested by special URL (`https://...`)
- Operates on a separate port number
- Designed for stateless web operation
- Servers are assumed to have certificates; clients may have them
- Negotiate session keys; also negotiate cryptographic algorithms to be used

# Session Start

- Client sends ClientHello message
- Identifies highest SSL version supported (TLS is called SSL 3.1)
- Random number for key generation
- Session ID
- List of supported cipher suites
- List of supported compression algorithms
- (Note: everything is encoded in ASN.1)

# Compression?

- Cannot compress encrypted data
- (Why not?)
- Must compress first, then encrypt — when SSL was first deployed, many people used dial-up modems that did compression, which would be rendered useless by encryption
- In practice, no compression algorithms were ever defined. . .

# SessionID

- Setting up a session is expensive because of the public key operations
- Every web transaction can be a separate TCP connection — even one for each embedded image
- Server *may* cache crypto parameters and resume session, with new key negotiated cheaply
- If the server doesn't want to resume the session, continue with new session

# Why Specify Cipher Suites?

- Handle different security/cost tradeoffs
- Handle newer ciphers
- Handle special situations, i.e., government-only ciphers

# Server Hello

- Actual SSL version to be used
- Server random number
- Actual cipher suites and compression algorithms to be used
- Actual sessionID to be used; if the same as the client's, this is a resumed session

# Server Certificate and Key Exchange Message

- Supplies the chain of certificates from this node up to (but rarely including) the root certificate
- Client *must* have the root certificate already. (Why?)
- Server optionally sends its key exchange data; existence and type are method-dependent.

# Client Key Exchange

- Client supplies its key exchange data; again, this is method-dependent
- The two key exchange messages are used to calculate the *pre-master secret*
- The pre-master secret is used to derive all keying material
- Client may send its certificate, too, if it has one and if the server requested it

# Key Exchanges

- Many different types!
- Most common: value encrypted with RSA by client
- Other important choice: signed Diffie-Hellman exponential ($g^x \bmod p$)

→ Provides forward secrecy

# Change Cipher Spec

- At this point, the client requests that encryption be activated via a Change Cipher Spec message
- The server does the same thing
- All communications from this point on are encrypted and authenticated
- They both then send Finished messages

# The Finished Message

- Crucial — *not* a simple end-of-handshake message
- Each side sends a PRF of the *master secret*, a label (either “client finished” or “server finished”) and a hash of all handshake messages
- (A PRF — pseudo-random function — is a keyed cryptographic function. No more details in this class! SSL’s PRF is very complex.)
- Each side verifies the other side’s data
- Guard against *downgrade attacks*

# Downgrade Attacks

*Normal Negotiation*

$C \rightarrow S$: (list of ciphers)

$S \rightarrow C$: Use Strong Cipher $n$

$C \rightarrow S$: (strongly encrypted text)

*Man in the Middle Attack*

$C \rightarrow S$: (list of strong ciphers)

[message intercepted and dropped by attacker!]

$X \rightarrow S$: (list of only weak ciphers)

$S \rightarrow C$: Use Weak Cipher $n$

$C \rightarrow S$: (weakly encrypted text)

The enemy tricks the server into using a broken algorithm. The same trick can be used to force use of old, buggy versions of protocols, e.g., SSLv2.

# (Why Does $C$ Accept Weak Ciphers?)

- Is a weak cipher better than cleartext? Almost certainly.
- Some servers only support weak crypto. Which is more important, talking to them with a weak cipher or not talking at all?
- It depends on the nature of the conversation — $C$ has to decide.
- What does your browser do?

# The Master Secret

- A PRF of the pre-master secret, the phrase “master secret”, and the client and server random numbers
- Note: “type strings” are included in calculations to prevent using a value from one computation as part of another
- Always 48 bytes long

# Calculating Keys

- Calculate the PRF of the master secret, the phrase ”key expansion”, and the two random values
- Call a *KDF* — Key Derivation Function
- Note: since both sides include random numbers, both sides affect — but neither controls — the actual keys used.
- The PRF can generate arbitrary amounts of data; use it for the client MAC key, the server MAC key, the client confidentiality key, and the server confidentiality key
- The cipher suite AES_256_CBC_SHA needs two 32-byte keys for AES, two 20-byte keys for MACs, and two 16-byte IVs, totaling 136 bytes

# Example: Client Hello (via sslᵈump)

```
6 1  0.0142 (0.0142)  C>SV3.1(53)  Handshake
      ClientHello
        Version 3.1
        random[32]=
          49 93 c7 2f ad f3 27 1f cf 0e b9 f9 b2 eb 62 e5
          16 ff bc d9 ea 94 d2 7c bd 39 c8 b8 c7 75 b2 a3
        cipher suites
        TLS_RSA_WITH_3DES_EDE_CBC_SHA
        TLS_RSA_WITH_RC4_128_SHA
        compression methods
                  NULL
```

# Server Hello

```
6 2  0.0145 (0.0002)  S>CV3.1(74)  Handshake
      ServerHello
        Version 3.1
        random[32]=
          49 93 c7 2f 04 8e 66 51 4d 94 3a f7 98 d9 7d ba
          17 16 49 cf 46 86 46 d7 cf 1f eb 4f c3 e1 8f 25
        session_id[32]=
          ac 00 e0 ed ea c7 34 70 51 47 3e 6c f3 f8 f6 80
          c8 26 96 3a 57 c6 ef a1 52 a5 8b 10 9a e6 0c 02
        cipherSuite         TLS_RSA_WITH_3DES_EDE_CBC_SHA
        compressionMethod                     NULL
```

# Key Exchange

```
6 3  0.0145 (0.0000)  S>CV3.1(2390)  Handshake
      Certificate
6 4  0.0145 (0.0000)  S>CV3.1(4)  Handshake
      ServerHelloDone
6 5  0.0375 (0.0229)  C>SV3.1(262)  Handshake
      ClientKeyExchange
        EncryptedPreMasterSecret[256]=
          (much hex data...)
```

If Diffie-Hellman were used, both sides would send $g^x \bmod p$ exponentials.

# Start-Up

```
6 6  0.0375 (0.0000)  C>SV3.1(1)  ChangeCipherSpec
6 7  0.0375 (0.0000)  C>SV3.1(40)  Handshake
      Finished
        verify_data[12]=
          90 56 e3 e1 ec 18 e3 63 9e 54 1c 99

6 8  0.0562 (0.0187)  S>CV3.1(1)  ChangeCipherSpec
6 9  0.0562 (0.0000)  S>CV3.1(40)  Handshake
      Finished
        verify_data[12]=
          58 9c 4c f8 46 72 f9 e9 89 13 ab d8
```

# Resuming A Session

- Uses just the two Hello, Change Cipher Spec, and Finished messages
- New random numbers in the Hello messages; used to calculate new keying material
- Cache contains the master secret
- Six messages, no big exponentiations: very efficient
- But — for long does a server cache session data? As long or as short a time as it wants: tradeoff between memory and CPU consumption. Also some modest incremental security risk.
- Note: no forward secrecy for cached sessions

# The SSL Record Layer

- All of the SSL crypto messages are embedded in a record format
- Record header contains message type (handshake, Change Cipher Spec, application data, Alert) and length
- Alert is for error messages, end-of-file, etc.
- Application data — such as HTTP messages — are encrypted and MACed
- This is the purpose of it all!

# MAC Failures

- If the MAC on a received message fails, must abort session
- May indicate an active attacker
- SSL is above TCP; as indicated before, no way to recover

# Export Silliness

- The US government used to restrict export of strong crypto: RSA or Diffie-Hellman moduli of $> 512$ bits, or symmetric ciphers with keys $> 40$ bits
- Exportable browsers only used weak ciphers
- But — the NSA wasn't interested in spying on e-commerce, so legitimate merchants could get a special certificate that turned on strong crypto in the exportable browser
- This meant that the strong crypto code was actually there — and it was very easy to patch the binary to enable it all the time. . .

# Conclusions

- Architecturally, SSL wasn't the best choice; arguably, it was the worst
- Need to convert every application; vulnerable to injected TCP segments
- No non-repudiation
- Credit card numbers in the clear on client and server
- But — *it was the only possible choice*
- Nothing else could have been deployed
- TLS has proven extraordinarily valuable for securing many other types of traffic, such as email
- It is the encryption protocol of choice for securing TCP streams