DIMACS Security & Cryptography
Crash Course – day 4
Internet Cryptography Tools,
Part I: TLS/ SSL

Prof. Amir Herzberg
Computer Science Department, Bar Ilan University
http://amir.herzberg.name

© Amir Herzberg, 2003. Permission is granted for academic use without modification. For other use please contact author.
Sources

- This lecture is mostly covered in `SSL and TLS` by Eric Rescorla
- Partial but readable coverage also in Stalling’s book, `Cryptography and Network Security`
- TLS is defined in Internet Engineering Task Force (IETF) RFC Document 2246, see e.g. at www.ietf.org
Agenda – Transport Layer Security

- Example: SSL payments
- Evolution of SSL and TLS
- Layer and alternatives
  - Few words about S/MIME
- SSL Protocol
  - SSL phases and services
  - Sessions and connections
  - SSL Handshake
  - SSL protocols and layers
  - SSL Record protocol / layer
- Secure use of SSL
  - Designing SSL applications
  - Client & server authentication
  - Web spoofing attacks
- Cryptographic issues in SSL and TLS
- Conclusions
SSL / TLS in a Nutshell

- SSL provides a `secure TCP tunnel from client to server`:
  - Confidentiality
  - Authentication of server, optionally also of client
  - Message and connection integrity
- SSL: Secure Socket Layer
  - Since SSL (& TLS) operate on top of `standard` Sockets API
- TLS: Transport Layer Security
  - Since TLS (& SSL) secure TCP (the transport layer)
  - IETF standard version of SSL
  - When we describe common aspects we usually say just SSL
- Many implementations, libraries, e.g. Open-SSL
- Original goal and still main use: secure transfer of credit card number… hear more on this in later lecture.
SSL/ TLS Evolution

SSLv1 (1994)
No client auth; broken – weak `randomness`, other weaknesses

SSLv2 (1994)
Substantial redesign; add client authentication, support for DSS, DH, prevent truncation attack

SSLv3 (1995)
Microsoft’s improved SSLv2: security (e.g. strong exportable auth.), performance (flows)

PCT (1995)
Not released

STLP (1996)
Microsoft’s improved SSLv3: support for UDP, and shared-secret authentication

TLS (1997-1999), RFC 2246
SSLv3 but incompatible: improved key expansion and MAC, support DES3 and DH+DSS for key exchange

WTLS (1990-)

Agenda - Transport Layer Security

- Example: SSL payments
- Evolution of SSL and TLS
- Layer and alternatives
  - Few words about S/MIME
- SSL Protocol
  - SSL phases and services
  - Sessions and connections
  - SSL Handshake
  - SSL protocols and layers
  - SSL Record protocol / layer
- Secure use of SSL
  - Designing SSL applications
  - Client & server authentication
  - Web spoofing attacks
- Cryptographic issues in SSL and TLS
- Conclusions
Adding Security in Transport Layer (SSL / TLS)

- SSL: Secure Socket Layer (Sockets is TCP/IP API)
- TLS: Transaction Layer Security (IETF standard SSL)
  - When we say `SSL`, we refer also to TLS

Pros:
- Easy to implement and use
- Deployed in most browsers, servers, …

Cons:
- Protects only if used by appl.
- Vulnerable to Clogging (DOS)
  - Over TCP
- Only end to end
- Headers exposed
Adding Security

Alternative 1: Add to Each Application

- **Pros:** easy, independent; awareness of semantics
- **Cons:**
  - Change each app, computer… hard, wasteful, error-prone, must trust all computers
  - No protection for headers
- **Examples:**
  - S/Key (login)
  - Payment protocols, e.g. SET (credit card payments)
  - Tools: XML security, Kerberos, …
  - Secure E-mail (S/MIME, PGP, …)
Few words about...
S/ MIME - Secure E-Mail

- MIME – Multi-purpose Internet Mail Extensions (message + attached files)
- S/MIME services:
  - Non-repudiation of origin
  - Authentication and integrity (signatures)
  - Confidentiality (encryption)
- Message parts: signature, encrypted shared key, encrypted data (using shared key)
- X.509 certificates (also CRLs) sent with message
  - Problem: PKI not in place for public applications
- APIs for communicating via S/MIME
- Widely deployed standard; available e.g. in Open-SSL
Adding Security
Alternative 2: IP Security

Pros:
- Protect all applications, data (IP header, addresses)
- No change to applications
- Gateway can protect many hosts
- Anti-clogging mechanisms
- Implemented by operating systems, Routers, ...
- Standard

Cons:
- Implementation, interoperability, availability
- Application awareness/control is difficult
Agenda - Transport Layer Security

- Example: SSL payments
- Evolution of SSL and TLS
- Layer and alternatives
  - Few words about S/MIME
- SSL Protocol
  - SSL phases and services
  - Sessions and connections
  - SSL Handshake
  - SSL protocols and layers
  - SSL Record protocol / layer
- Secure use of SSL
  - Designing SSL applications
  - Client & server authentication
  - Web spoofing attacks
- Cryptographic issues in SSL and TLS
  - Key derivation (PRF)
  - Order of Encryption/Auth
  - Chosen ciphertext attack
- DOS attacks on Servers
- SSL payments: problems
- Conclusions
SSL Operation Phases (high level)

- TCP Connection
- Handshake
  - Negotiate (agree on) algorithms, methods
  - Authenticate server and optionally client
  - Establish keys
- Data transfer
- SSL Secure Teardown (why is this necessary?)
SSL Services

- Server Authentication (mandatory)
- Client Authentication (optional - if required by server)

Secure connection:
- Confidentiality (Encryption) – optional, possibly weak (export)
- Message Authentication
- Reliability: prevent re-ordering, truncating etc.

Efficiency: allow resumption of SSL session in new connection (no need to re-do handshake)
SSL Operation Phases

- Client uses SSL API to open connection
- SSL Handshake protocol:
  - For efficiency – resume `session` if possible
  - If not (session not kept, new connection, override)
    - Establish session - algorithms and master keys
  - Establish connection (keys, etc.)
- Data transfer (SSL Record protocol)
- Teardown – use Alert protocol:
  - By application closing connection
  - Or due to error (by handshake or record protocols)
SSL Sessions and Connections

- **Connection:**
  - TCP/IP connection – send/receive secure messages
  - Reliable: ensures Delivery, Matching, FIFO
  - Independent, different keys for each connection

- **SSL Session:**
  - May span multiple connections for efficiency
  - Agree on algorithms and options
    - Client specifies possibilities, server chooses or rejects
  - Use public keys to Establish shared *MasterSecret* key
  - Server sets `session_id` so connection can resume (use existing session, for efficiency)
    - Client, server may discard session
    - Recommended (in RFC): keep session at most 24 hours
SSL Session State Variables

- Session ID: 32 bytes selected by server
- Peer certificate (X.509 v3)
- Compression method
- Cipher spec (encryption, MAC, etc.)
- Is Resumable: flag: allow new connections
- master_secret: 48 bytes, known to both
  - Derived from 48 bytes pre_master_secret (from DH key exchange / sent encrypted by RSA)
  - Using random numbers chosen by server and client at 1st connection of session
  - Using Pseudo-Random Function (PRF)
  - How?
Deriving \textit{master\_secret} Key

\texttt{master\_secret} = \texttt{PRF}_\texttt{pre\_master\_secret}(\texttt{"master secret" || Client\_random || Server\_random})

\texttt{PRF} is based on MD5 and \texttt{SHA-1}; design differs btw SSL & TLS, see later

\begin{center}
\begin{tikzpicture}
\node[rectangle,fill=orange!50] (prf) {\texttt{PRF}};
\node[below of=prf,anchor=north] (ms) {\texttt{master\_secret}};
\node[below of=prf,anchor=north] (psms) {\texttt{pre\_master\_secret}};
\node[below of=prf,anchor=north] (cr) {\texttt{Client\_random}};
\node[below of=prf,anchor=north] (sr) {\texttt{Server\_random}};
\draw[-latex] (prf) -- (ms);
\draw[-latex] (prf) -- (psms);
\draw[-latex] (prf) -- (cr);
\draw[-latex] (prf) -- (sr);
\end{tikzpicture}
\end{center}

(Pseudo Random Function)
SSL Connection State Variables

- Session ID: 32 bytes selected by server
- Server and client sequence numbers
- `Server_random, client_random`: 32 bytes
  - Unique to each connection!
- Cryptographic keys and Initialization Vectors (IV)
  - Unique to each connection (why?)
  - Distinct encryption and authentication (MAC) keys (why?)
  - Distinct keys for client to server and server to client packets (why?)
  - How?
Deriving Connection Keys, IVs

\[ \text{Key\_Block} = \text{PRF}_{\text{master\_secret}} \left( \text{“key expansion”} || \text{Server\_random} \ || \text{Client\_random} \right) \]

Split \text{Key\_Block} to \text{ClientMACKey}, \text{serverMACKey}, \text{ClientEncryptKey}, \ldots (using fixed order)

\[ \text{Server\_random} \quad \text{master\_secret} \quad \text{Client\_random} \]

PRF

\[ \text{Key\_Block} \]

MAC keys Encrypt keys IVs
SSL Handshake Protocol

- Agree on _cipher suite_: algorithms and options:
  - Symmetric and Asymmetric Encryption
  - Signature and MAC
  - Compression
  - Options: client authentication, export (weak) versions,…
- Exchange random values
- Check for session resumption.
- Send certificate(s)
- Establish shared keys.
- Authenticate server
- Optionally authenticate client
- Confirm synchronization with peer
SSL Handshake - Overview

Client

Possible Cipher-suites, Client_random

Chosen cipher-suite, Server_random, Certificate

Encrypted Pre_Master_Secret

Server

Client, Server change to new, computed keys ("Cipher Spec")

Confirmation (MAC of handshake messages)

Confirmation (MAC of handshake messages)

Confirms algorithms, no replay, client really sent Pre_Master_Secret
SSL Typical Handshake Messages

Client

- ClientHello (possible cipher-suites, \textit{Client\_random})
- ServerHello (Chosen cipher-suite, \textit{Server\_random})
- Certificate
- ServerHelloDone
- \textbf{ClientKeyExchange} (Encrypted \textit{Pre\_Master\_Secret})
- ChangeCipherSpec (CCS)
- \textbf{Finished} (Confirmation -MAC of handshake messages)
- ChangeCipherSpec (CCS)
- \textbf{Finished} (Confirmation -MAC of handshake messages)

Server

- \textbf{ServerHello} (Chosen cipher-suite, \textit{Server\_random})
- Certificate
- \textbf{ServerHelloDone}
- \textbf{ClientKeyExchange} (Encrypted \textit{Pre\_Master\_Secret})
- ChangeCipherSpec (CCS)
- \textbf{Finished} (Confirmation -MAC of handshake messages)
- ChangeCipherSpec (CCS)
- \textbf{Finished} (Confirmation -MAC of handshake messages)

Client begins using new key

Server begins using new key
Advanced Handshake Features

- Session resumption
- Client authentication
- Ephemeral public keys
  - For forward security – (usually?) using Diffie-Hellman
  - Support for DH, with DSS signatures, is mandatory in TLS
  - Or, for using weak encryption public keys for export reasons (signed by strong public key) – Often with RSA
  - RSA key generation is expensive – often same ephemeral (and short, 512 bits) key used for multiple clients/sessions
Handshake with Ephemeral public keys

- ClientHello
- ServerHello
- Certificate
- ServerKeyExchange
- ServerHelloDone
- ClientKeyExchange
- ChangeCipherSpec (CCS)
- Finished
- ChangeCipherSpec (CCS)
- Finished

RSA/DSA Signature over ephemeral RSA key or DH exponent

If RSA used: regular (encrypted pre-master);
If DH used: client’s exponent
SSL Client Authentication

- Usually, only the server has a certificate
  - Client can authenticate the server
  - Client sends some identification info (e.g. username, password) to server using the SSL tunnel – after it is established

- SSL also supports authentication with client certificates
  - Server requires certificate from client
  - Server signals acceptable Certificate Authorities (CAs) and certificate formats, options etc.
  - Client returns appropriate certificate (chain)
  - Client authenticates by signing using certified public key

- Client authentication using certificates is used mostly within organizations, communities – more on this later
Client Authentication Handshake

Client

ClientHello (ciphersuites, Client_random)

ServerHello (ciphersuite, Server_random)

Certificate

CertificateRequest

ServerHelloDone

Certificate

ClientKeyExchange (Encrypted Pre_Master_Secret)

CertificateVerify

CCS

Finished

Server

Acceptable CA and cert formats

Or certificate chain (same for server cert.)

Signature over hash of handshake messages

CCS

Finished
SSL Session Resumption

- SSL session setup has substantial overhead
  - Randomness generation (both)
  - Transmission of certificates (both)
  - RSA encryption of Pre-Master-secret (client)
  - RSA decryption of Pre-Master-secret (server)
  - Derivation of master secret and key block (both)

- Problems:
  - Significant performance penalty (mainly on server)
  - Server vulnerable to clogging (DOS) attacks

- Session resumption:
  - If client makes many connections to same server…
  - Server, client can re-use Pre-Master-secret from last connection
  - How? By identifying a session using session ID
Session Resumption Handshake

Client

ClientHello (cipher-suites, resume(session_id), Client_random)

ServerHello (Chosen cipher-suite, session_id, Server_random)

ChangeCipherSpec (CCS)

Finished (Confirmation -MAC of handshake messages)

ChangeCipherSpec (CCS)

Finished (Confirmation -MAC of handshake messages)

Server

In first session of connection (not resumed), client does not send session_id, and only server sends it with ServerHello to allow resumption
Session Resumption Issues

- Caching requires considerable server resources
  - Result: cache usually kept for only few minutes, not 24 hrs
- Resumption conflicts with replicated (cluster) servers
  - TCP connections routed to arbitrary server in cluster
  - Solution 1: server in cluster determined by client IP address \(\rightarrow\) but requests from many clients may use same NAT IP addr
  - Solution 2: shared storage of session information \(\rightarrow\) not easy!
  - Solution 3: SSL-session aware connection routing
  - Solution 4: Client side session caching – encrypted, authenticated cache; a non-standard SSL/TLS extension
- Session resumption helps only for repeating connections
  - SSL payments involve one (or few) connections \(\rightarrow\) not much help
- Other possible optimizations (not standardized)
  - Client caching of certificates and other server info (‘fast track’)
  - Encrypt using ephemeral, short server keys
  - Server encrypts Pre-Master-Secret using Client’s public key
## Handshake Protocol Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>M?</th>
<th>From</th>
<th>Meaning/Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloReq.</td>
<td>O</td>
<td>Srvr</td>
<td>Inform client to begin</td>
</tr>
<tr>
<td>ClientHello</td>
<td>M</td>
<td>Clnt</td>
<td>Version, <code>client_random</code>, <code>session_ID</code>, algorithms</td>
</tr>
<tr>
<td>ServerHello</td>
<td>M</td>
<td>Srvr</td>
<td>Version, <code>server_random</code>, <code>session_ID</code>, algorithms</td>
</tr>
<tr>
<td>Certificate</td>
<td>O</td>
<td>Both</td>
<td>X.509 certificate</td>
</tr>
<tr>
<td>ServerKeyExchng</td>
<td>O</td>
<td>Srvr</td>
<td>Ephemeral server pub key (this session only)</td>
</tr>
<tr>
<td>Cert. Request</td>
<td>O</td>
<td>Srvr</td>
<td>Cert. type (RSA/DSS, Sign/DH), CAs</td>
</tr>
<tr>
<td>ClientKeyExchng</td>
<td>M</td>
<td>Clnt</td>
<td>Encrypted <code>pre_master_key</code></td>
</tr>
<tr>
<td>Cert. verify</td>
<td>O</td>
<td>Clnt</td>
<td>Sign previous messages</td>
</tr>
<tr>
<td>Finished</td>
<td>M</td>
<td>Both</td>
<td>MAC on entire handshake</td>
</tr>
</tbody>
</table>
SSL Protocols, Layers and Records

Application (e.g. browser)

SSL API

SSL Record Protocol/Layer (MAC, encrypt, compress, counters)

SSL Alert

SSL Handshake Protocol

Alert record

Handshake record

CCS record

Reliable Transport Layer (TCP)

CCS= Change Cipher Spec

(original) Sockets API
SSL Record Layer

- Assumes underlying reliable communication (TCP)
- Fragmentation, compression, authentication, encryption

Message sent by the application, e.g. HTTP request

<16KB

Message sent by the application, e.g.

<16KB

HTTP request

<16KB

Fragment

Compress

MAC

Pad (if using block cipher)

Encrypt

Send each fragment via TCP
SSL Record Protocol

1. Fragments data – 16KB in a fragment
2. Compress each fragment; Compression must be lossless and never increase length (up to 1KB Ok)
3. Authenticate by appending MAC
   - Key: MAC_write_secret (from master_secret)
   - MAC computed over counter || length || content
   - Use counter (64 bits) to prevent replay in SSL session
   - The counter value is only input to MAC, not sent
     ■ Since we assume SSL is over TCP which ensures FIFO
     ■ So why SSL adds counter to MAC at all?
4. Padding to complete block (if using block cipher)
5. Encrypt fragment (including MAC)
Alert Protocol and Record

- Signal state changes and indicate errors
- Invoked by:
  - Application - to close connection *(close_notify)*
    - Connection should close with *close_notify*
    - This allows detection of *truncation attack* (dropping of last messages)
    - Notice: *close_notify* is normal, not failure alert!
  - Handshake protocol – in case of problem
  - Record protocol – e.g. if MAC is not valid
    - Notice: easy to tear-down (denial of service)
- Alert record carries alerts
**Agenda - Transport Layer Security**

- Example: SSL payments
- Evolution of SSL and TLS
- Layer and alternatives
  - Few words about S/MIME
- SSL Protocol
  - SSL phases and services
  - Sessions and connections
  - SSL Handshake
  - SSL protocols and layers
  - SSL Record protocol / layer
- Secure use of SSL
  - Designing SSL applications
  - Client & server authentication
  - Web spoofing attacks
- Cryptographic issues in SSL and TLS
- Conclusions
Secure Usage of SSL

- Designing Secure Applications using SSL API
- Validating Certificate (or certificates chain)
- Server Access Control (client authentication)
  - Using client certificates
  - Using username and password, etc.
- Client Access Control (server authentication)
- Site spoofing attacks on browsers
Designing Applications using SSL API

- Several SSL toolkits (e.g. OpenSSL); slightly different APIs
- Initialization tasks:
  - Load CA’s certificates (at clients; servers: only if using client auth)
  - Load keys and certificates
  - Seed random number generator (use collected noise)
  - Load allowed cipher suites
    - Most toolkits allow adding new (more secure?) cipher suites
  - In server: generate/load ephemeral DH and/or RSA keys (if used)
- Connection API calls
  - Very similar to standard TCP (Sockets) API
  - But returns server (and optionally client) certificate
  - Need to validate certificate
  - Close (tear-down) connection - to identify truncation attacks
Validating Certificates

- Validation done by application, not SSL!!
- Verify root CA is trusted
  - Predefined list of `trusted CAs` in application
    - E.g. look in your browser…
  - Do we really trust all of them?
- Validate certificate (chain)
  - Validate signature(s)
  - Check validity/expiration dates
  - Check identities, constraints, key usage…
  - Check for revocations – SSL does not carry CRLs; application must collect by itself if CRL’s are used.

- Reminder…
Recall: X.509 Certificate Validation

- Signature on the above fields
- Subject unique identifier
- Issuer unique identifier
- Subject (user) Distinguished Name (DN)
- Validity period
- Issuer Distinguished Name (DN)
- Key Usage extension(s)
- Policy (ID)
- Name Constraints extension
- Constraints
- Mappings
- SubjectAltName ext.
  - E-mail
  - DNS
  - URI
- Basic constraint: Cert_len [for CA>0]
- Acceptable?
- Kept?
- Cf. to subject ID
- Cf. to CA ID
- Cf. date/time
- Cf. to CA name

Valid?
After Validating Certificates: Access Control

- Application (e.g. browser or server):
  - Verify root CA is trusted
  - Validate certificate (chain):
    - Validity, expiration, revocation
    - Identities, constraints, key-usage, …
  - Extract name/ID from Distinguished Name, subjectAltName…

- Client access control (after server authentication):
  - Is this the server the client wanted to connect to?
  - Is this the *kind of server* the client had in mind? (e.g. Visa-authorized merchant)
  - Done by client application (e.g. browser) and client (manually)

- Server access control (after client authentication)
  - Is this an authorized client/customer?
  - What are his permissions?
Client Authentication with Certificates
(Server Access Control)

- Typically X.509 certificates are *identity certificates*
- **Client certificates**: identity should be known to server…
- Problem: no global, unique namespace ("John Smith12"…)
- Personal certificates from General-purpose CA’s (e.g. Verisign) are not very useful, and very uncommon
- Result: each server/community use their own certificates, naming
- Client has to chose certificate for each server → inconvenient
- **Server must be able to identify names of authorized clients**
Server Access Control
(Client Authentication) Methods

- Using client certificates…
  - High level of security
  - Requires issuing (buying?) certificates to each client
  - Browsers prompt user to select certificate (hassle)
  - If based on identity, requires database of clients in server

- Using Username-Password authentication
  - Browser sends password as argument of a form
    - Possibly filled by browser (``wallet`` function: passport, ECML)
  - Relies on SSL security (encryption+server authentication)
  - Better but non-standard: use password as key of MAC
    (never send password – don’t expose to spoofed server)
  - Inconvenience: typing/approving password per request
Secure Session

- Goal: authenticate **once** per application session
- How? Few options…
  - Application session = SSL session
    - Requires session identification – usually available in API
    - But session retention is limited (browsers, servers)
  - Or: identify application session… how?
    - Cookie contains application session id (and/or password)
    - Send cookie with each request/response:
      - Automated cookie mechanisms in browsers
      - Or: encode cookie as part of URLs
  - Risks: exposure, forgery, privacy
    - Exercise: design of secure cookie mechanism
### Server Authentication

- Critical – e.g. when user enters secrets (password, cc#, …)
- Based on Server’s X.509 *identity certificates*
- Certificate (chain) must pass validation
  - Responsibility of application
  - Browsers pre-configured with many CA’s and don’t test chain well
  - Usually CA validates ownership of site… using *insecure* DNS
  - You can remove untrusted CA’s from browser (but few do this)
- Server identity:
  - Typically (e.g. in browsers): DNS name, e.g. www.citibank.com
  - Not IP address since it is not meaningful and may change
- No standard mapping of DNS to Distinguished Name
  - *Usually* use dNSName field in subjectAltName extension
- User must specify or at least know and understand:
  - If connection is secure, server authenticated
  - What is the (DNS?) name of the server
Indicating Secure Connection and Server Identity

- Ensure user is aware of server’s identity
- Ensure user is aware of (in)secure connection
- The user should identify the server
  - Give same DNS Name as in certificate
  - Notice: the same server may host multiple sites (e.g. ISP)
  - Solution: must have certificate for each hosted site
- Spoofing attacks on browsers: directing user to spoofed site
  - Changing link (URL) in referring site…
    - Visible, but unnoticed by (most) users, or
    - Advanced spoofing: (almost?) non-visible – screen emulation
- Security degrading attacks
Site-Spoofing Attacks on Browsers

- User visits spoofing site, site becomes proxy
- User browsing is thru proxy
- User is not aware
  - Most users don’t look at URLs
  - Or: spoof sends phony certificate
  - Or: spoof *emulates* normal browsing
    - JavaScript: same window, fake URL, SSL indicator
    - Java: emulated window (supports interaction)
  - Or: spoof selects weakest security offered by client, E.g. SSL ver. 2, PCT, DES,…
Agenda - Transport Layer Security

- Example: SSL payments
- Evolution of SSL and TLS
- Layer and alternatives
  - Few words about S/MIME
- SSL Protocol
  - SSL phases and services
  - Sessions and connections
  - SSL Handshake
  - SSL protocols and layers
  - SSL Record protocol / layer
- Secure use of SSL
  - Designing SSL applications
  - Client & server authentication
  - Web spoofing attacks
- Cryptographic issues in SSL and TLS
- Conclusions
Cryptographic Issues in SSL & TLS

- Much research and security improvements in evolution of SSL & TLS
- We do not cover the (critical!) fixes to SSLv1, v2
  - See e.g. in Rescola’s book ("SSL and TLS").
  - SSLv2 is enabled by default in many browsers
- TLS improves security cf. to SSLv3:
  - Cryptanalysis-tolerance
  - In particular: passes US FIPS-140 criteria
  - Internal design of MAC, hash functions, etc.
- Details: in `extras`...
Conclusion

SSL / TLS is the most widely deployed security protocol, standard
- Easy to implement, deploy and use; widely available
- Flexible, supports many scenarios and policies
- Mature cryptographic design

But SSL is not always the best tool…
- Use IP-Sec e.g. for anti-clogging, broader protection
- Use application security, e.g. s/mime, for non-repudiation, store-and-forward communication (not online)

Beware of host-spoofing and web-spoofing
- Many browsers allow hard-to-detect spoofing
- Many users will not detect simple spoofing (similar URL)
Extras...
Crypto in SSL & TLS: Key Derivation

- Key derivation in SSL, TLS:
  - Key block (block of connection keys) from master_secret
  - master_secret from pre_master_secret

- Critical for security

- Design based on hash functions
  - Why not on block ciphers e.g. AES? Not available when SSL designed; DES was already too weak, no other standard and free cipher

- Which hash function to use?
  - Two main candidates: MD5 and SHA1
  - SSLv2: use MD5; SSLv3 and TLS: use both!

- How to use the hash functions?
  - Different design for TLS and SSL
  - SSL design: intuitive
  - TLS design: Cryptanalysis-tolerant PRF
Key Derivation in SSLv3

- Based on HMAC: \( \text{HMAC}_h(k(m) = \text{h}(k \oplus \text{opad} || \text{h}(k \oplus \text{ipad} || m)) \)
- Intuition: output of HMAC should be unpredictable
- Idea: modify HMAC to use both MD5 and SHA-1
- SSL modifications:
  - Use SHA for the `internal` hash, MD5 for the `external`
  - Prepend different strings to generate enough output
  - Slightly different for master secret and key block (not sure why)

- \( pms=\text{PreMasterSecret}, \ cr=\text{Client_random}, \ sr=\text{Server_random} \)
- \( ms=\text{Master_secret}= \text{MD5}(pms||\text{SHA(“A”||pms||cr||sr)||MD5(pms||\text{SHA(“BB”||pms||cr||sr)||MD5(pms||\text{SHA(“CCC”||pms||cr||sr))}})) \)
- \( \text{Key_block}= \text{MD5}(ms||\text{SHA(“A”||ms||sr||cr)||MD5(ms||\text{SHA(“BB”||ms||sr||cr)||MD5(ms||\text{SHA(“CCC”||ms||sr||cr))}})||… \)
Key Derivation in SSLv3 - Criticism

- Recall *Key_block* (same argument for *MasterSecret*):
  - Let $ms=\text{MasterSecret}$, $cr=\text{Client_random}$, $sr=\text{Server_random}$
  - $Key\_block = MD5(ms||SHA(“A”||ms||sr||cr)) || MD5(ms||SHA(“BB”||ms||sr||cr)) || ...$

- Completely intuitive, no justification / analysis
- HMAC analysis/proof depend on *both* internal and external hash having security properties:
  - Internal hash: Collision-resistant-only VIL MAC
  - External hash: Fixed-Input Length secure MAC
- If either MD5 or SHA is weak, derivation may be weak
- No cryptanalysis-tolerance!
- Fails FIPS-140: security should depend only on FIPS-approved cryptographic mechanisms
Key Derivation in TLS: use PRF

- Idea: the `standard` secure mechanism for key derivation is a Pseudo-Random Function (PRF)
- For example, using master key $k$ and PRF $f_k$:
  - To derive an encryption key: $EncKey = f_k(\text{“encrypt”})$
  - To derive authentication key from client to server, use: $C2SAuthKey = f_k(\text{“auth, client to server”})$
  - To use different encryption keys in each connection, (using same master key): $EncKey = f_k(\text{“encrypt”, random})$
  - Or, in TLS: derive one long $Key\_block$, then split it and use different (fixed) parts of it for keys for encryption, authentication, and IV, in each direction

- How? Recall Pseudo-Random Function (PRF)…
Pseudo-Random Functions (PRF)

- An \( m \) to \( n \) FIL-PRF is a collection of efficient functions \( \{ f_k : \{0,1\}^m \rightarrow \{0,1\}^n \} \), such that no adversary can efficiently distinguish between \( f_k \) for random key \( k \), and a random function \( r \) from \( \{0,1\}^m \) to \( \{0,1\}^n \)

\[
\begin{align*}
a & \in_R \{0,1\}, \; k & \in_R \{0,1\}^n, \; r & \in_R \{ \text{fun: } \{0,1\}^m \rightarrow \{0,1\}^n \}
\end{align*}
\]
Key Derivation: Two Steps...

- **Step 1: FIL $\rightarrow$ VIL** (Fixed $\rightarrow$ Variable Input Length)
  - SHA’s output is 160 bits, MD5 output is 128 bit… and more bits are needed anyway
  - Transform FIL PRF $HMAC_{h_k}$ to VIL $PRF_{h_k}$
  - $h$ is either SHA or MD5

- **Step 2: cryptanalysis-tolerant VIL PRF composition:** given VIL $PRF_{MD5_k}$ and $PRF_{SHA_k}$, design VIL $PRF_k$ to be secure as long as either $PRF_{MD5_k}$ or $PRF_{SHA_k}$ is secure
Step 1: FIL PRF $\rightarrow$ VIL PRF

- Assume: $HMAC_{h_k}$ is a FIL PRF
- Design of VIL $PRF_h$: concatenate outputs, using different `labels` $A(i)$:
  
  \[
  PRF_{h_k}(r) = HMAC_{h_k}(A_h(1)||r) \parallel HMAC_{h_k}(A_h(2)||r) \parallel \ldots
  \]

- Labels $A_h(i)$ derived by HMAC:
  
  \[
  A_h(i) = HMAC_{h_{secret}}(A_h(i-1)); \quad A_h(0) = cr||sr
  \]

- Simpler design $A_h(i) = i$ is also secure (assuming $HMAC_{h_k}$ is a FIL PRF)

- But more complex design above is (almost) as efficient, and seems more robust to `typical` attacks against $HMAC_{h_k}$ (e.g. attack that finds $HMAC_{h_k}(2)$ given $HMAC_{h_k}(1)$)
Step 2: Cryptanalysis Tolerance

- Given two candidate VIL PRFs: 
  \( PRF_{MD5}, PRF_{SHA} \)
- Intuition: cryptanalysis-tolerant composition:
  \( PRF_k(r) = PRF_{MD5_k(r)} \oplus PRF_{SHA_k(r)} \)
- Question/exercise: is this composition cryptanalysis-tolerant?
Cryptanalysis-Tolerant PRF: 1st try…

- Consider any two PRF-candidates $f, g$
- Define $P_k(m) = f_k(m) \oplus g_k(m)$
- Question: assume either $f$ or $g$ is a PRF. Is then $P$ a PRF?
- Answer: NO.
- Trivial examples: $f_k(m) = g_k(m), f_k(m) = \sim g_k(m)$
- Intuition may hold for `independent` $f, g$… (e.g. MD5 and SHA?)
- Making input different, e.g. $f_k(1||m) \oplus g_k(0||m)$, does not help (why?)
- Idea: use different keys!
**TLS: Cryptanalysis-Tolerant PRF**

- Define $P_{k_1,k_2}(m)=f_{k_1}(m)\oplus g_{k_2}(m)$
- **Claim:** if either $f$ or $g$ is a PRF, then $P$ a PRF.
- **Proof sketch:** assume $g$ is a PRF but $P$ is not a PRF. Namely there is an algorithm $A$, that can distinguish between a box computing $P_{k_1,k_2}(\ )$ and a box computing a random function.

Assume now we are given a box computing either $g_{k_2}(m)$ or a random function. We use it to compute $P_{k_1,k_2}(m)=f_{k_1}(m)\oplus g_{k_2}(m)$ (selecting $k_1$ ourselves). Now we use $A$ to distinguish between this and random.

- This is what is done in TLS!
PRF in TLS - Details

- PRF keys \((\text{PreMasterSecret, MasterSecret})\) are 48B
- Use only half of it (24 bytes) for each PRF-candidate (PRF_MD5 and PRF_SHA)

\[ \text{TLS}_{\text{PRF}}(r) = \text{PRF}_{\text{MD5}}[48...25](r) \oplus \text{PRF}_{\text{SHA}}[1...24](r) \]

- Deriving as many bytes as necessary
  - E.g. 48 bytes for Master Secret
- To derive Master Secret:
  - Let \(m_{\text{MS}} = \text{"master secret"} \| \text{client_random} \| \text{server_random}\)
  - \(\text{MasterSecret} = \text{TLS}_{\text{PRF}}_{\text{PreMasterSecret}}(m_{\text{MS}})\)
- To derive Key Block:
  - Let \(m_{\text{KB}} = \text{"key expansion"} \| \text{client_random} \| \text{server_random}\)
  - \(\text{KeyBlock} = \text{TLS}_{\text{PRF}}_{\text{MasterSecret}}(m_{\text{KB}})\)
Cryptographic Issues in SSL & TLS: Finished Message Computation

- Finished message is sent at end of handshake:
  - From client to server and vice versa
- Goal: to authenticate entire handshake using \textit{master\_secret}
- Authentication uses both MD5 and SHA (for cryptanalysis-tolerance)
- Computation differs between SSL and TLS
- SSL: for both \( h=MD5 \) and \( h=SHA \), send
  \[
  h(master\_secret \| opad \| h(messages \| Sender \| master\_secret \| ipad))
  \]
  This differs from HMAC: \( h(k \oplus opad \| h(k \oplus ipad \| m)) \)
- Motivation for difference: key \((master\_secret)\) defined just at Finish…
- But consider hash design (Merkle-Damgard), this may be insecure!
- TLS is simpler and more secure: send 12 bytes from output of
  \[
  PRF_{master\_secret}(label\|MD5(messages)\|SHA(messages))
  \]
  - Label is either “server” or “client”
Cryptographic Issues in SSL & TLS: Client Certificate Verification

- Recall client authentication handshake
Client Authentication Handshake

Client

- ClientHello (ciphersuites, Client_random)
- Certificate
- CertificateRequest
- ServerHello (ciphersuite, Server_random)
- ServerHelloDone
- Certificate

Server

- ClientKeyExchange (Encrypted Pre_Master_Secret)
- CertificateVerify
- CCS
- CCS
- Finished
- Finished

Signature over hash of handshake messages
CertificateVerify Message

- Sent from client to server to authenticate client
- Contains signature over hash of handshake messages
  - Using RSA: both MD5 hash and SHA hash (for cryptanalysis-tolerance)
  - Using DSA: only SHA hash
- Hash computation differs between SSL and TLS:
  - SSL: \( h(master\_secret \| h(messages \| master\_secret \| pad)) \)
  - TLS: \( h(messages) \)
- Why?
  - Unnecessary complication in SSL; messages are not secret, hashing is (supposed to be) collision-resistant
  - Possible, unnecessary exposure of \( master\_secret \)
  - This is the only place it is used directly as key (of MAC...)
Cryptographic Issues in SSL & TLS:
RSA Encryption Format (PKCS#1)

- SSL and TLS are using PKCS #1 Version 1.5
- Recall: Subject to Feedback-only Chosen-Ciphertext Attack (CCA) [Bleichenbacher’98]
- Attack is practical against some SSL, TLS implementations (see later…)

```
00 02 padding string 00 message
```

at least 8 bytes

\( k \) bytes
Reminder: Feedback-only Chosen-Ciphertext Attack [Bleichenbacher'98]

Alice
PK: (n=pq, e)

Bob
SK: (p, q, d: ed=1 mod φ(n))

Eve
C

C' = C S^e (mod n)

R = 0/1 (depending on correctness of padding of C')
Preventing CCA Attack

- Some SSL, TLS implementations send specific alert immediately on detecting bad PKCS#1 format
- Helps attacker; need only 1 million trials (chosen ciphertexts) to decrypt message
- Prevention is easy…
  - Send same alert if pre-master-secret is not formatted correctly, attacker needs about $2^{40}$ trials → not practical
  - RFC224 recommendation: don’t send alerts, use random pre-master-secret → will fail in Finish message validation
  - USE PKCS#1 version 2 (OAEP) or another format secure against CCA
Cryptographic Issues in SSL & TLS: order of Auth / Encrypt

- SSL authenticates, then encrypts:
  - $A = MAC(m)$, $C = Enc(m,A)$, send $C$

- IPSEC encrypts, then authenticates:
  - $C = Enc(m)$, $A = MAC(C)$, send $(C,A)$

- Which is better? Does it matter?
Question: Order of Auth / Encrypt

- SSL authenticates, then encrypts (AtE):
  - $A = MAC(m)$, $C = Enc(m,A)$, send $C$
- IPSEC encrypts, then authenticates (EtA):
  - $C = Enc(m)$, $A = MAC(C)$, send $(C,A)$

Which is better? Does it matter?

- $Enc(m,A)$ may be harder to cryptanalyze cf. to $Enc(m)$, so AtE seems to strengthen encryption
- But we should use secure encryption, not depend on $A = MAC(m)$ to fix it!
Question: Order of Auth/ Encrypt

- SSL authenticates, then encrypts (AtE):
  - $A = \text{MAC}(m)$, $C = \text{Enc}(m,A)$, send $C$
- IPSEC encrypts, then authenticates (EtA):
  - $C = \text{Enc}(m)$, $A = \text{MAC}(C)$, send $(C,A)$

- EtA seems better:
  - EtA resistant to clogging (verify MAC before decrypt)
  - EtA allows to authenticate (also) public data
    - E.g. extend to multiple recipients (multicast)
  - AtE subject to attack if attacker knows if authentication failed or not
    - Although not with standard encryption – OTP, CBC
    - Recall attack from day 6, `Authentication`…
Feedback-only Chosen-Ciphertext Attack on Authenticate-then-Encrypt

- Assume: attacker can choose ciphertext, and whether it passes or fails authentication validation.

- Define the following cipher $E$ based on One Time Pad (OTP) (or a pseudo-random generator):
  - $E_k(x) = \text{Transform}(x) \oplus k$ [bit-wise XOR]
  - $\text{Transform}$ each bit of the plaintext to two bits:
    - Zero bits (0) are transformed to two zeros (00)
    - One bits (1) are transformed to (01) or (10) randomly

- $E$ indistinguishable under chosen plaintext attack
- We show an attack on $\text{auth-then-encrypt}$ when using $E$
- Attack: flip first two bits of ciphertext.
  - If authentication is still valid, first plaintext bit is 1
  - If authentication fails, first plaintext bit is zero.