# Defending Computer Networks
# *Lecture 21: Transport Layer Security*

Stuart Staniford
Adjunct Professor of Computer Science

# Logistics

- HW 5 due tomorrow midnight
- HW6 out, due Friday midnight (short)
- Last lecture on Thursday
  - Will go over midterm quickly
- Final is Weds Dec 9th 2pm-4pm

# News

HOME ABOUT BENEFITS CONTACT

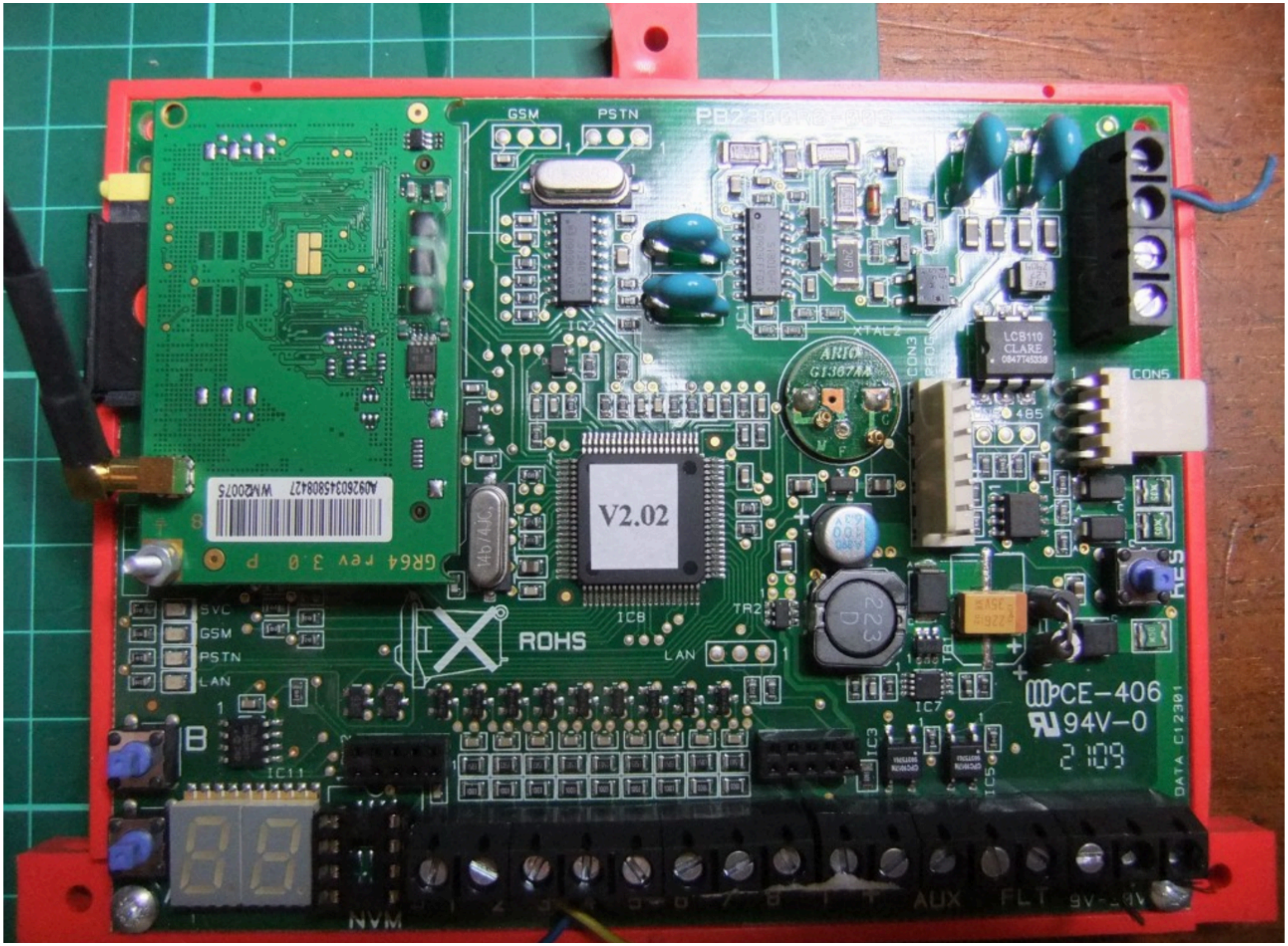
## CSL Dualcom CS2300-R signalling unit vulnerabilities

POSTED ON NOVEMBER 23, 2015 BY CYBERGIBBONS

Today, CERT/CC will be disclosing a series of vulnerabilities I have discovered in one particular alarm signalling product made by CSL Dualcom – the CS2300-R. These are:

- CWE-287: Improper Authentication – CVE-2015-7285
- CWE-327: Use of a Broken or Risky Cryptographic Algorithm – CVE-2015-7286
- CWE-255: Credentials Management – CVE-2015-7287
- CWE-912: Hidden Functionality – CVE-2015-7288

# The product



GSM
PSTN
V2.02
ROHS
SVC
GSM
PSTN
LAN
NVM
AUX
FLT
LAN
PCE-406
94V-0
LCB110
CLARE
CON5
GR64 rev 3.0 P

# Issues

- Weak homegrown cipher (simple substitution)
- Fixed key in firmware (same in all instances)
- No way to update product online

# Main Focus of Today

- Last bit of RSA
- Start on TLS (formerly known as SSL)
  - And various attendant technologies

# Public Key Cryptography

- Also known as Asymmetric Cryptography
  - To distinguish from symmetric cryptography
    - shared secret
- Instead of shared key, keys come in pairs
  - Public key used to encrypt data
  - Private key used to decrypt data
  - Not feasible to infer private key from public key

# RSA

- Underlying base is difficulty of factoring very large numbers
- Will sketch algorithm while again skipping worst of math details
- We choose two large primes *p*, *q*
  - hundreds of digits each
  - Modulus, $n = pq$
  - Size of *n* in bits is the key length
  - Then choose an exponent, *e* that
    - Has no common factors with $(p-1)(q-1)$
- Public key is *n* and *e*
- Private key can be computed from *p* & *q*

# Encryption

- Take the text and turn blocks of text into numbers.  Say M is number for some block
- Then take $M^e$ mod n
- Toy example
  - http://www.woodmann.com/crackz/Tutorials/Rsa.htm
  - p = 43, q = 59 (both prime)
  - n = 43*59 = 2537
  - e = 13 (no common factor with 42 or 58)

# Encryption Example

- Message is ST OP
- Encode as 1819 1415
  - $1819^{13} \bmod 2537 = 2081$
  - $1415^{13} \bmod 2537 = 2182$
  - Ciphertext is 2081 2182

# Decryption

- Find decryption exponent d such that
- $de \bmod (p-1)(q-1) = 1$
- Ie $d$ is the multiplicative inverse of $e$,
  - modulo $(p-1)(q-1)$
  - Tractable algorithms for this are known
- Then plaintext P can be computed from C
  - $P = C^d \bmod n$

# Decryption Example

- Suppose ciphertext is 0981 0461
- d = 937 for our previous example
- 937*13 mod 2436 = 12181 mod 2436 = 1
- $0981^{937}$ mod 2537 = 0704
- $0461^{937}$ mod 2537 = 1115
- Message was HE LP

# Applicability of Public Key

- Typically public key algorithms are computationally expensive.
- Not practical to apply to long messages
- Therefore generally used in the process of establishing a temporary symmetric key
  - Session key
    - Encrypted by public key crypto for transfer
    - Then used to encrypt lengthy communication session
    - Then thrown away

# Let’s Try It

- openssl genrsa -out private_key.pem 1024
- more private_key.pem
- openssl rsa -in private_key.pem -text -noout
- openssl rsa -pubout -in private_key.pem -out public_key.pem
- more public_key.pem

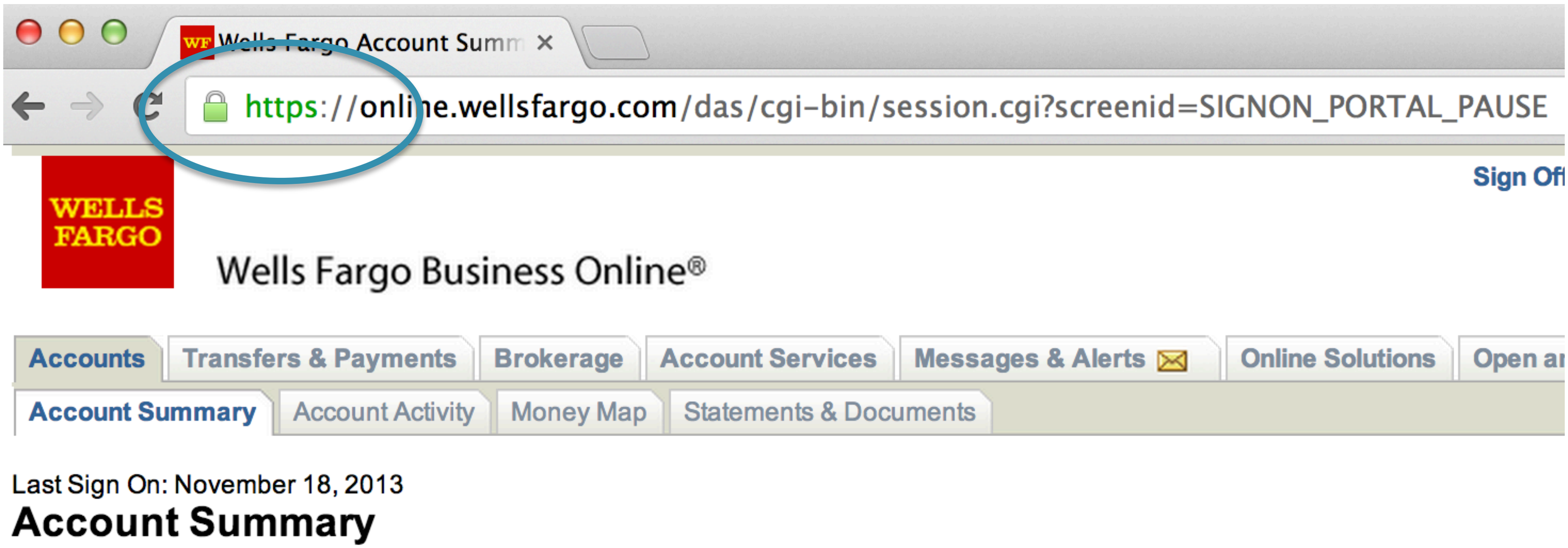
# Main Advantage

- Enormously simplifies key management
  - Imagine large group that need to communicate by shared key schemes
    - Share one key amongst many and risk losing everything
    - Or keep track of $n^2$ keys
- With public key crypto, I give out my public key, everyone can know it.
  - Anyone can send me a message
  - Only I can read those messages
  - I only have to worry about one secret key (mine)
  - Don’t have to share my secret (private) key with anyone

# Main Goals of TLS/SSL

- Transport Layer Security/Secure Sockets Layer
- Original goal was secure HTTP: HTTPS
- Now heavily used as basis for VPNs
  - Virtual Private Network
  - Way to provide secure network connections to remote users
- Used for email (POP/SMTP/IMAP over SSL)
- Used for SIP (VOIP protocol)

# HTTPS


Wells Fargo Account Summ
https://online.wellsfargo.com/das/cgi-bin/session.cgi?screenid=SIGNON_PORTAL_PAUSE
Sign Off
WELLS FARGO
Wells Fargo Business Online®
Accounts
Transfers & Payments
Brokerage
Account Services
Messages & Alerts
Online Solutions
Open ar
Account Summary
Account Activity
Money Map
Statements & Documents
Last Sign On: November 18, 2013
Account Summary

# TLS History

- 1995: SSL 2.0 (Netscape)
- 1996: SSL 3.0 (Netscape, later RFC 6101)
- 1999: TLS 1.0 (RFC 2246)
- 2006: TLS 1.1 (RFC 4346)
- 2008: TLS 1.2 (RFC 5246, 6176)
  - Supported in latest versions of all major browsers
- TLS 1.3 in IETF draft status

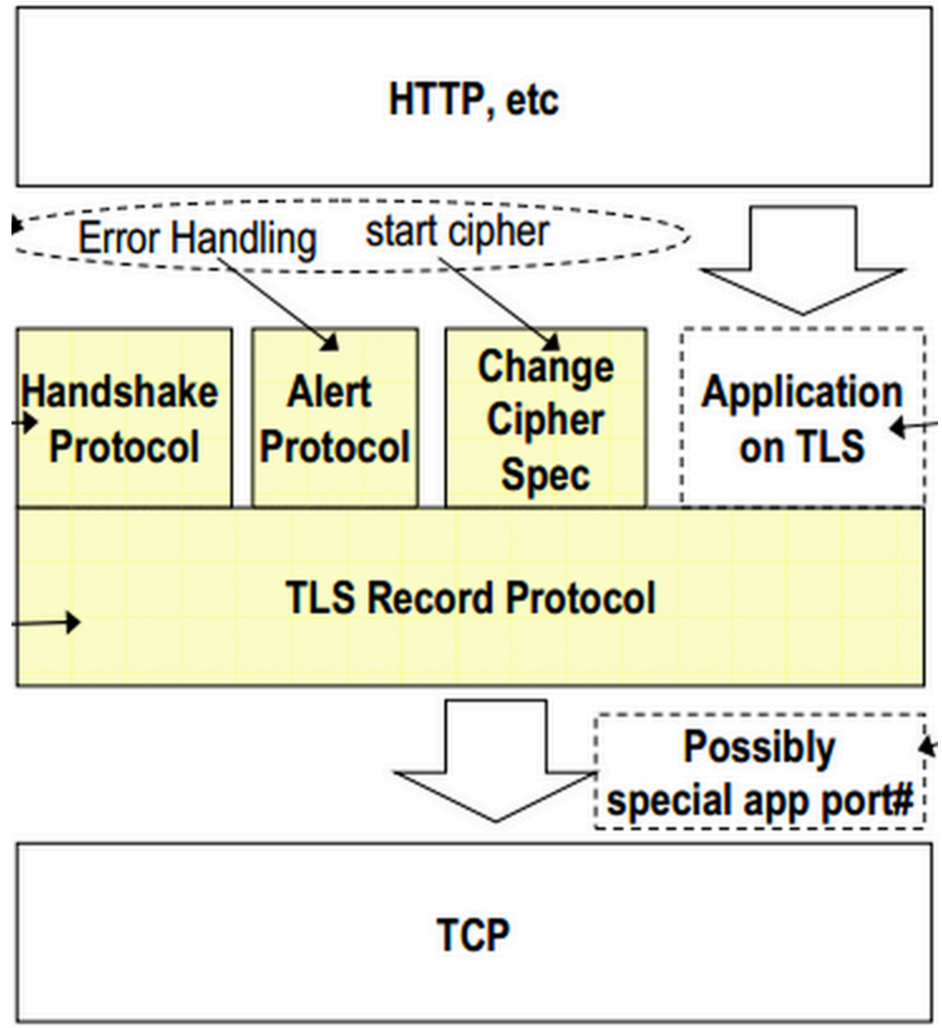
# TLS: Step 1

- Establish unencrypted connection
  - Typically over TCP
  - Eg HTTPS over port 443
  - Has also been implemented over UDP
  - And…

# Possible Future Implementations

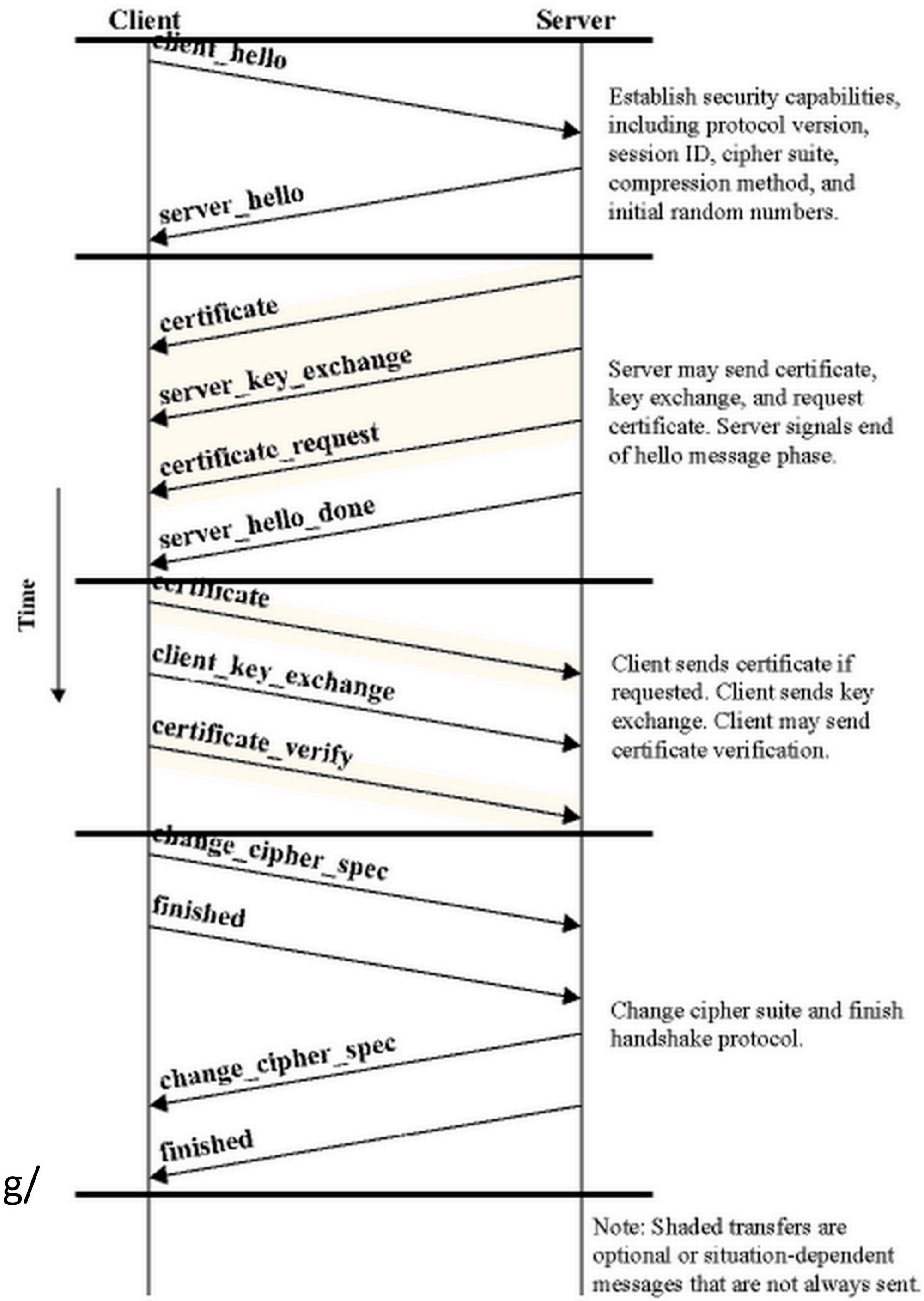
# TLS Handshake

- To establish parameters of remaining conversation
- Works over TLS Record layer
- Can also change operation of record layer




http://tech.yanatm.com/?p=338

# Handshake Overview




http://www.cs.bham.ac.uk/~mdr/teaching/modules06/netsec/lectures/tls/tls.html

# TLS Client Hello

**Client Hello.** The client initiates a session by sending a Client Hello message to the server. The Client Hello message contains:

- **Version Number.** The client sends the version number corresponding to the highest version it supports. Version 2 is used for SSL 2.0, version 3 for SSL 3.0, and version 3.1 for TLS. Although the IETF RFC for TLS is TLS version 1.0, the protocol uses 3.1 in the version field to indicate that it is a higher level (newer and with more functionality) than SSL 3.0.

- **Randomly Generated Data.** ClientRandom[32], the random value, is a 4-byte number that consists of the client's date and time plus a 28-byte randomly generated number that will ultimately be used with the server random value to generate a master secret from which the encryption keys will be derived.

- **Session Identification (if any).** The sessionID is included to enable the client to resume a previous session. Resuming a previous session can be useful, because creating a new session requires processor-intensive public key operations that can be avoided by resuming an existing session with its established session keys. Previous session information, identified by the sessionID, is stored in the respective client and server session caches.

- **Cipher Suite.** The A list of cipher suites available on the client. An example of a cipher suite is TLS_RSA_WITH_DES_CBC_SHA, where TLS is the protocol version, RSA is the algorithm that will be used for the key exchange, DES_CBC is the encryption algorithm (using a 56-bit key in CBC mode), and SHA is the hash function.

- **Compression Algorithm.** The requested compression algorithm (none currently supported).

http://technet.microsoft.com/en-us/library/cc785811(v=ws.10).aspx

# TLS Server Hello

**Server Hello.** The server responds with a Server Hello message. The Server Hello message includes:

- **Version Number.** The server sends the highest version number supported by both sides. This is the lower of: the highest version number the server supports and the version sent in the Client Hello message.
- **Randomly Generated Data.** ServerRandom[32], the Random Value, is a 4-byte number of the server's date and time plus a 28-byte randomly generated number that will be ultimately used with the client random value to generate a master secret from which the encryption keys will be derived
- **Session Identification (if any).** This can be one of three choices.
  - New session ID – The client did not indicate a session to resume so a new ID is generated. A new session ID is also generated when the client indicates a session to resume but the server can't or won't resume that session. This latter case also results in a new session ID.
  - Resumed Session ID– The id is the same as indicated in the client hello. The client indicated a session ID to resume and the server is willing to resume that session.
  - Null – this is a new session, but the server is not willing to resume it at a later time so no ID is returned.
- **Cipher Suite.** The server will choose the strongest cipher that both the client and server support. If there are no cipher suites that both parties support, the session is ended with a "handshake failure" alert.
- **Compression Algorithm.** Specifies the compression algorithm to use (none currently supported).

# TLS Certificate

- Next the server sends its certificate.
- So what is a certificate?
- That requires something of a detour…
  - Cryptographic hash/message digest
  - Digital signature
  - Certificate

# Cryptographic Hash

- Take an input plain text *m*
- Outputs a message digest of fixed size *h(m)*
- Such that
  - Any change in input will change output significantly (usually totally)
  - Computationally infeasible, given *h*, to find *m*
  - Computationally infeasible to find $m_1$ and $m_2$ such that $h(m_1) = h(m_2)$

# Example Cryptographic Hashes

- ~~MD5~~ (long gone bad)
- ~~SH~~A-1 (starting to go bad)
- SHA-256 (currently recommended)
- SHA-3 (on horizon)
- Try it
  - openssl md5 bank.c
  - openssl sha1 bank.c
  - openssl sha256 bank.c
  - openssl no-sha256 or openssl help
  - openssl version

# How SHA-1 Works

- Examine the pseudo-code at
- http://en.wikipedia.org/wiki/SHA-1

# Digital Signature

- Scheme for guaranteeing that a message is what it appears to be
  - Authentication
    - was not created by an imposter instead of sender
  - Non-repudiation
    - Sender cannot deny having sent it
  - Integrity
    - Message not altered in transit

# Digital Signature Implementation

- Sender
  - Hash message with cryptographic hash fn
  - Encrypt hash with *private* key (eg RSA)
  - Attach signature to message
- Receiver
  - decrypt signature with *public* key
  - Recompute hash
  - Make sure signature matches message hash
- Essentially proves that sender was in possession of private key associated with given public key

# X.509 Certificates

- Public Key Infrastructure
  - Dates back to 1988
  - RFC 5280 (IETF version) for practical purposes
- Tries to solve the problem
  - How do I find/validate the public key for X?
- Relies on the idea of chain of trust

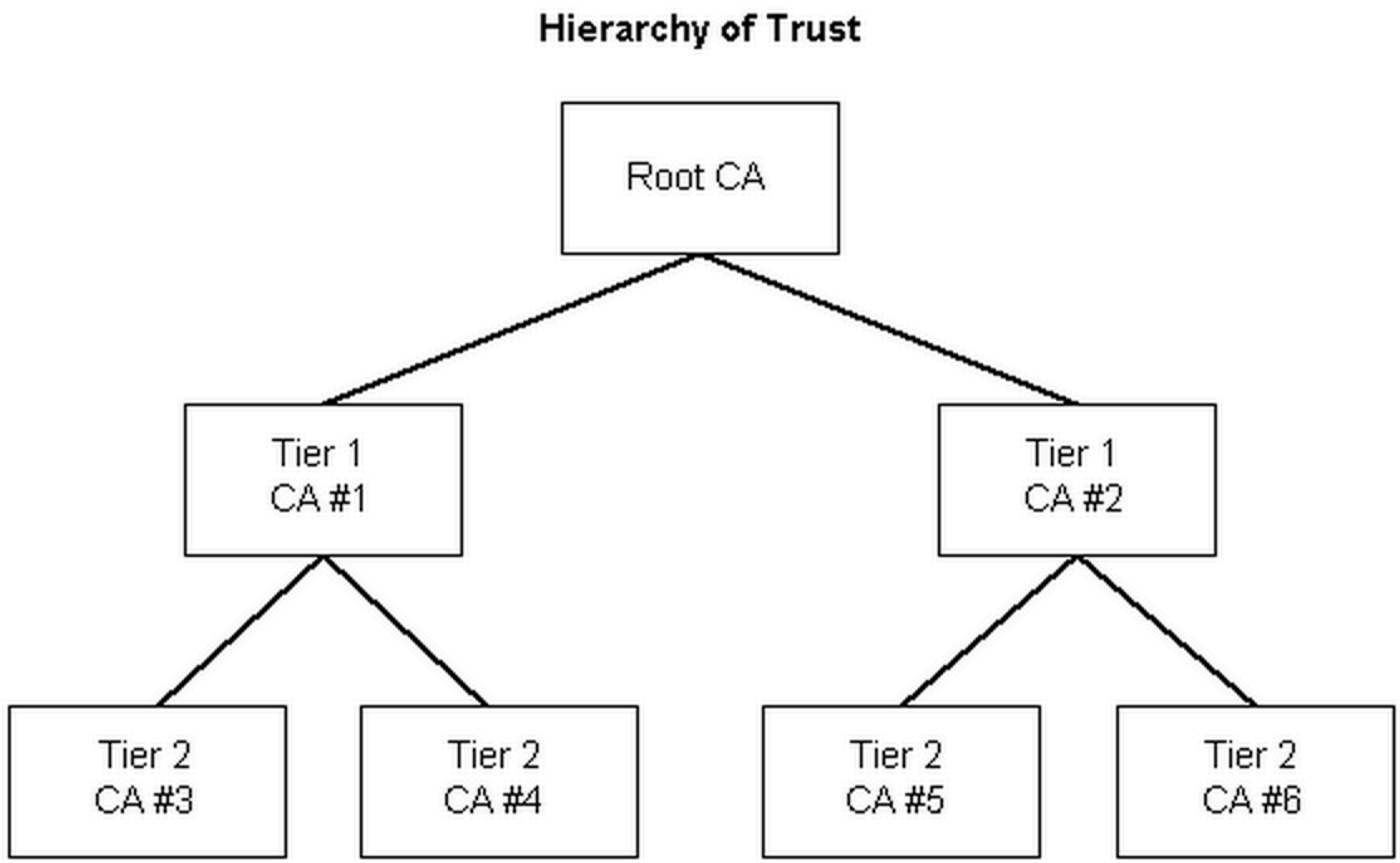
# What is in X.509 Cert?

- Certificate version (eg 3)
- Serial Number (eg 480025)
- Algorithm ID (eg 'sha1WithRSAEncryption')
- Issuer (CA = Certificate Authority, eg Geotrust)
- Validity Time Interval
  - Cert not to be used outside of this

# More in X.509 Cert

- Subject: (eg domain of website)
- Subject Public Key info
  - Algorithm
  - Public Key data
- X509 extensions
- Signature

# Certificate Authorities




http://msdn.microsoft.com/en-us/windows/aa382479(v=vs.90)

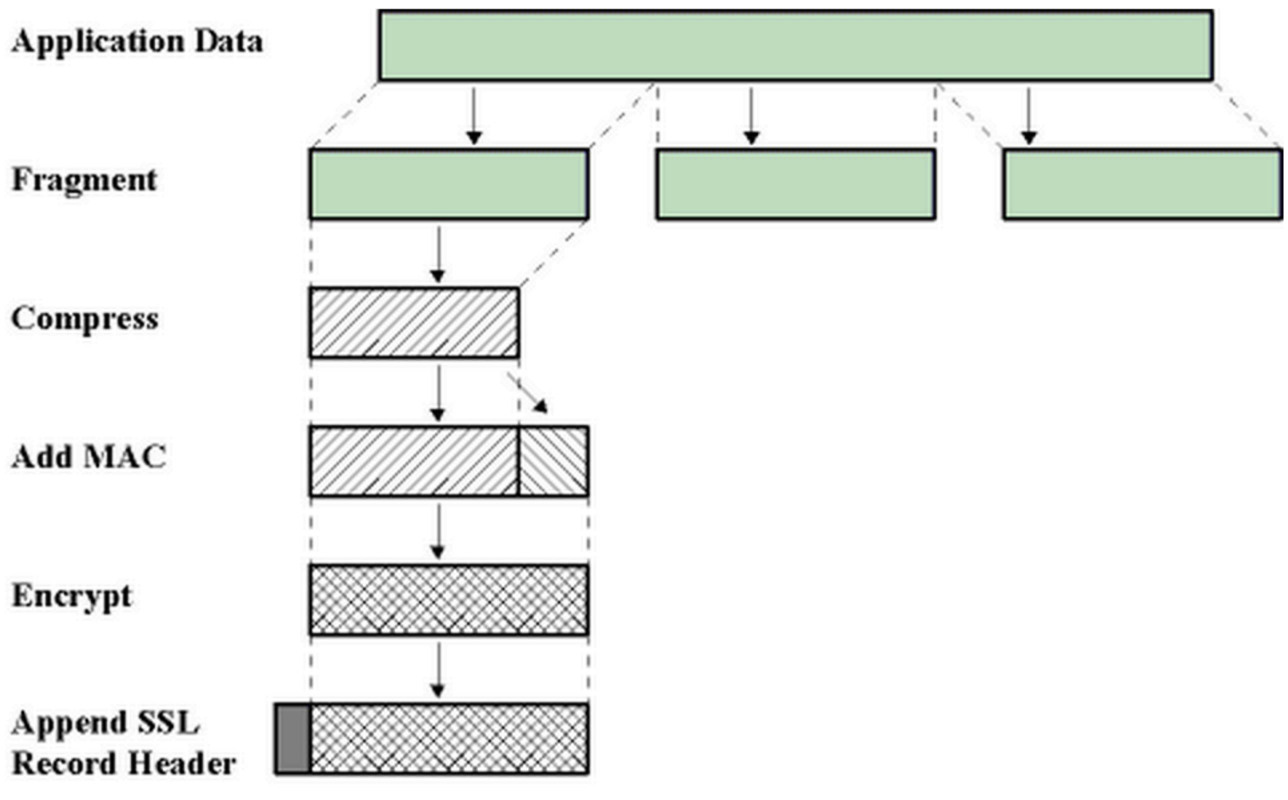
# Hundreds of Root CAs

- Eg let’s examine the list for Mozilla:
- http://www.mozilla.org/projects/security/certs/included/

# Obtaining Certificate to Play

- https://www.networking4all.com/en/support/tools/site+check/
- Eg try with www.nytimes.com
- openssl x509 -in nytimes-cert.txt -text |more

# TLS Record Layer




http://www.cs.bham.ac.uk/~mdr/teaching/modules06/netsec/lectures/tls/tls.html

# TLS Record Format

<table>
<tr><th>Byte +0</th><th>Byte +1</th><th>Byte +2</th><th>Byte +3</th></tr>
<tr><td>Content type</td><td colspan="3"></td></tr>
<tr><td colspan="2">Version</td><td colspan="2">Length</td></tr>
<tr><td>(Major)</td><td>(Minor)</td><td>(bits 15..8)</td><td>(bits 7..0)</td></tr>
<tr><td colspan="4">Protocol message(s)</td></tr>
<tr><td colspan="4">MAC (optional)</td></tr>
<tr><td colspan="4">Padding (block ciphers only)</td></tr>
</table>

| Major Version | Minor Version | Version Type |
|---|---|---|
| 3 | 0 | SSL 3.0 |
| 3 | 1 | TLS 1.0 |
| 3 | 2 | TLS 1.1 |
| 3 | 3 | TLS 1.2 |

| Hex | Dec | Type |
|---|---|---|
| 0x14 | 20 | ChangeCipherSpec |
| 0x15 | 21 | Alert |
| 0x16 | 22 | Handshake |
| 0x17 | 23 | Application |

http://tech.yanatm.com/?p=338