# KRY05 - MNG

Model 2019

# Kryptografie

Část 5

Asymetrická správa klíčů

Post 19/20

Souhrnné materiály

Ver 0.1

©Petr Hanáček

# KRY

## Učebnice

5

- Nigel Smart: Cryptography - An Introduction, 3rd Edition,
  - Mcgraw-Hill College, 3rd Edition, 2013
  - ISBN-10: 0077099877
- Kapitoly
  - Kapitola 16 - Obtaining Authentic Public Keys
    - » Zajímavá je pro nás podkapitola 1 až 3.3

The third edition is now online. You may make copies and distribute the copies of the book as you see fit, as long as it is clearly marked as having been authored by N.P. Smart.

Učebnice je v dokumentovém skladu

CRYPTOGRAPHY
SMART

©Petr Hanáček

X.509

# Certifikáty podle X.509

Petr Hanáček
Ústav informatiky a výpočetní techniky
Vysoké učení technické v Brně
Božetěchova 2
612 66 Brno
tel. 7275 216
e-mail: hanacek@dcse.fee.vutbr.cz

©Petr Hanáček CLACRYPT Slide 3

## Potřeba

- prakticky všechny bezp. protokoly v počítačových sítích jsou založeny na kryptografických mechanismech
- protokoly potřebují ke své efektivní činnosti správu klíčů (klíčové hospodářství)
- jednou z možností implementace správy klíčů je systém kryptografických certifikátů veřejných klíčů
- nejrozšířenějším standardem pro kryptografické certifikáty je doporučení CCITT X.509
  - součást doporučení pro adresářové služby X.500
- příklady implementací
  - NetWare Directory Services
  - Internet (např. Lightweight Directory Access Protocol RFC 1487)

©Petr Hanáček CLACRYPT Slide 4

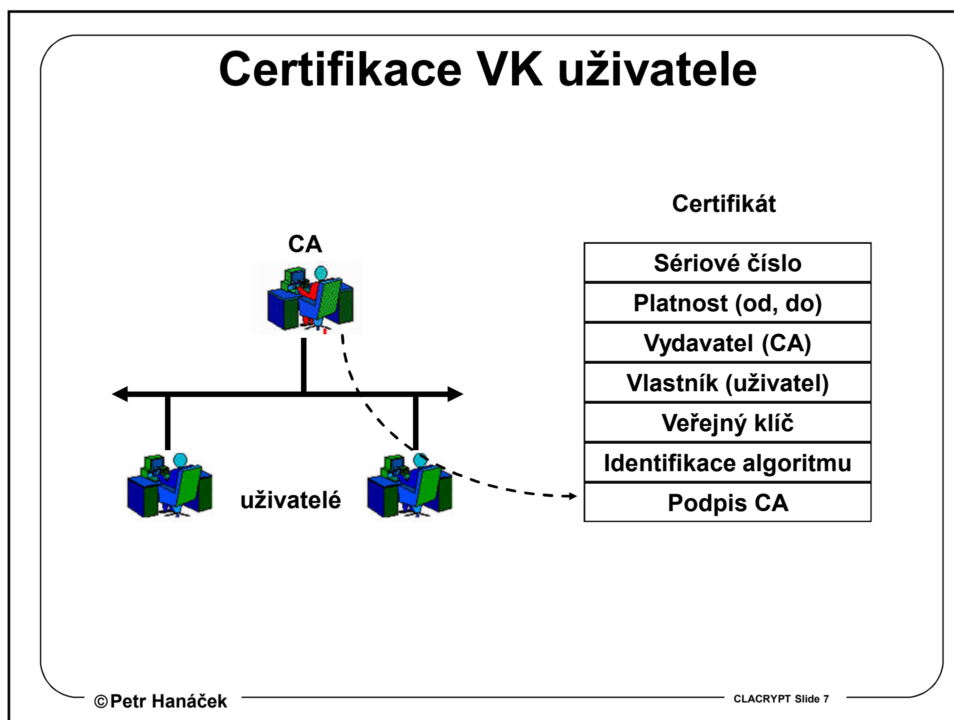
## Důvody pro certifikaci

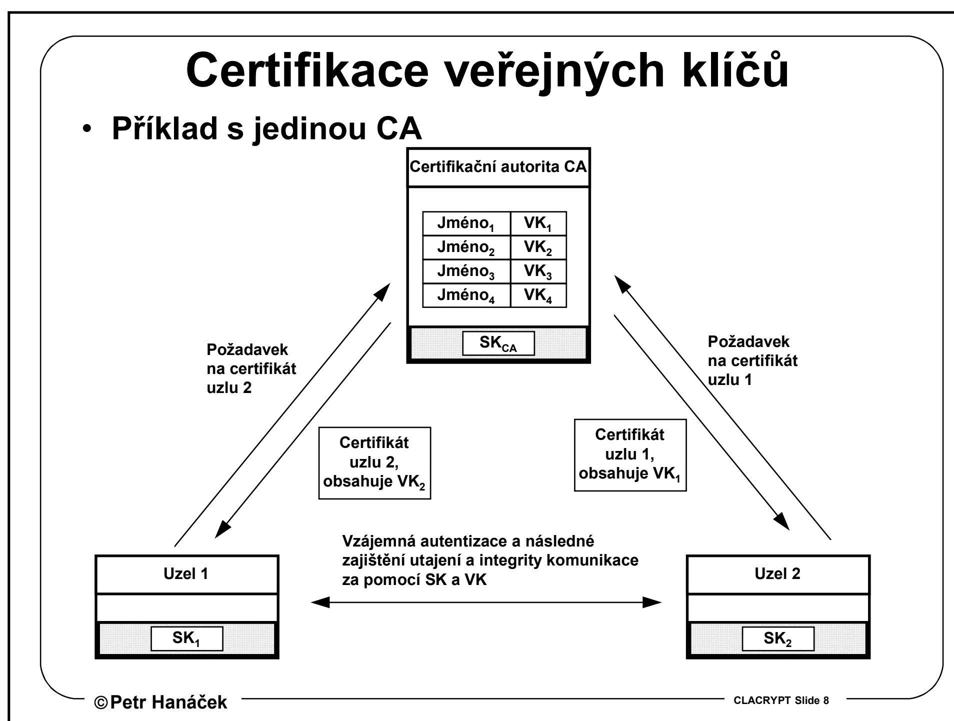
- **účely kryptografie**
  - autentizace
  - důvěrnost
  - integrita
  - nepopiratelnost
- **kryptografické algoritmy**
  - symetrické algoritmy - šifrování i dešifrování prováděno stejným klíčem
    - » oba klíče jsou tajné
  - asymetrické algoritmy - pro šifrování a dešifrování různé klíče
    - » druhý klíč je soukromý
      - nedistribuuje se, je třeba jej utajit
    - » jeden klíč je veřejný
      - distribuuje se, neutajuje se, je třeba zabránit jeho podvržení - certifikace



## Certifikace veřejného klíče

- **otázka autenticity veřejných klíčů (VK)**
  - např. ověřovatel el. podpisu si musí být jist, že VK, který používá k ověřování daného podpisu, je skutečně VK autora zprávy
  - spolehlivá vazba mezi VK a jménem
- **řešení - certifikace VK subjektem, kterému všichni důvěřují**
  - tento prostředník je certifikační autorita (CA)
- **certifikace**
  - CA podepíše VK uživatele a jeho jméno (a další údaje, např. doba platnosti) svým vlastním tajným klíčem
  - tyto údaje, podepsané CA, se nazývají certifikát
    - » certifikát může být ověřen VK certifikační autority
- **ověření VK partnera**
  - ověřením elektronického podpisu certifikátu pomocí VK CA
  - jediný klíč, kterému uživatel musí věřit, je VK CA


Certifikace VK uživatele
CA
uživatelé
Certifikát
Sériové číslo
Platnost (od, do)
Vydavatel (CA)
Vlastník (uživatel)
Veřejný klíč
Identifikace algoritmu
Podpis CA
©Petr Hanáček
CLACRYPT Slide 7


Certifikace veřejných klíčů
• Příklad s jedinou CA
Certifikační autorita CA
Jméno1 VK1
Jméno2 VK2
Jméno3 VK3
Jméno4 VK4
SKCA
Požadavek na certifikát uzlu 2
Požadavek na certifikát uzlu 1
Certifikát uzlu 2, obsahuje VK2
Certifikát uzlu 1, obsahuje VK1
Vzájemná autentizace a následné zajištění utajení a integrity komunikace za pomocí SK a VK
Uzel 1
SK1
Uzel 2
SK2
©Petr Hanáček
CLACRYPT Slide 8

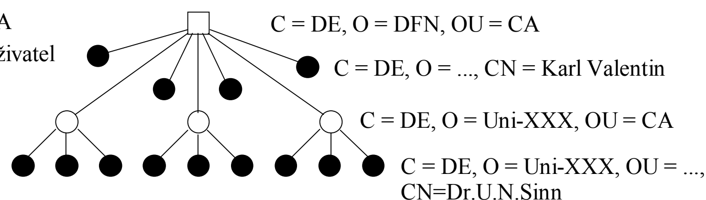
## Strom CA

- ve velkých skupinách uživatelů nestačí jediná CA
- VK certifikačních autorit mohou být opět certifikovány jinými certifikačními autoritami
- stromové struktury certifikačních autorit
  - křížová certifikace mezi stromy
- kořenový veřejný klíč
  - řetěz certifikací nemůže být nekonečný
  - veřejný klíč posledního certifikátu zůstává necertifikovaný - kořenový veřejný klíč
  - autenticita tohoto klíče musí být zajištěna jiným způsobem
    - » získání kurýrem
    - » z papírového média
    - » ...

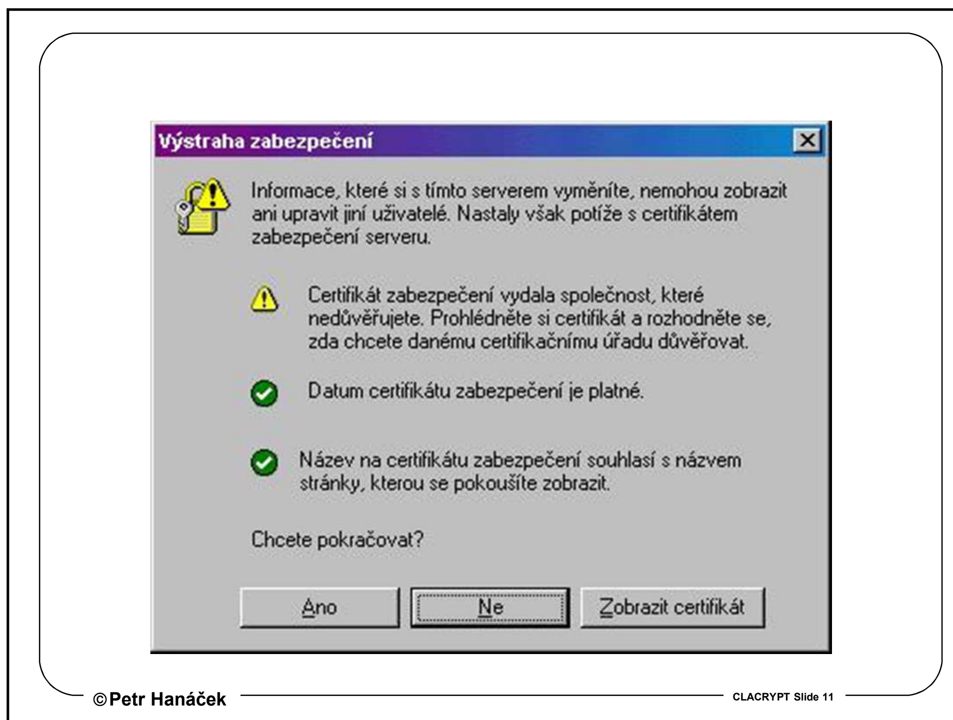
©Petr Hanáček 

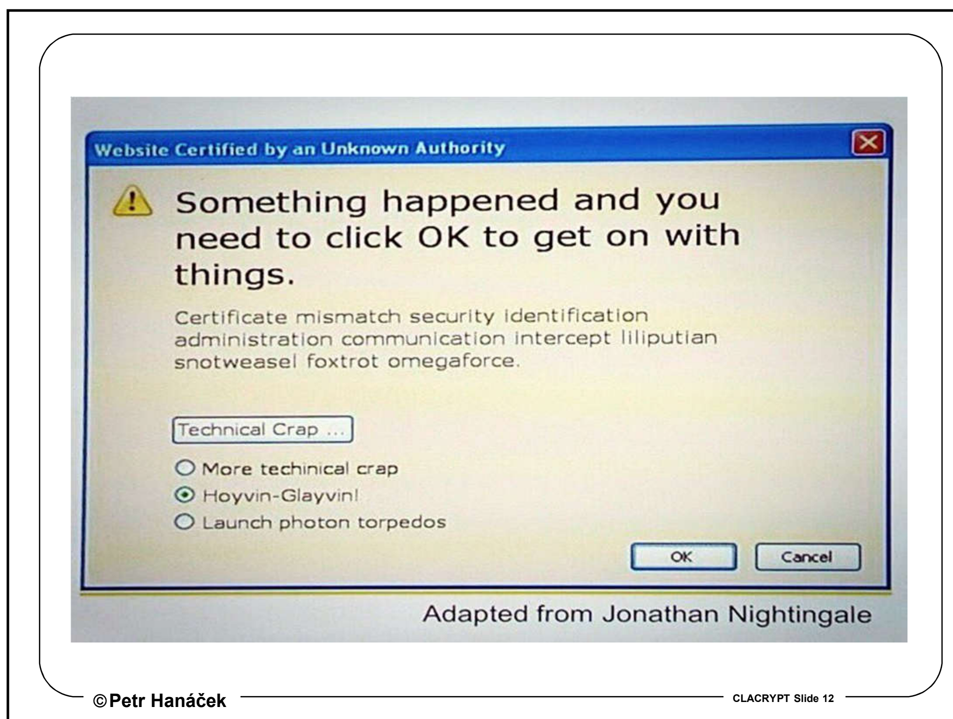
## Příklad certifikačního stromu

- jednoduchý certifikační strom
- na vrcholu tohoto stromu je jedna Kořenová CA, např.:
  - C = DE, O = DFN, OU = CA.
  - Jejím úkolem je například certifikace ostatních certifikačních autorit v celé organizaci:
  - C = DE, O = uni xxx, OU = CA.
  - tyto ostatní certifikační autority jsou uživatelské CA a vydávají certifikáty uživatelům v organizaci




©Petr Hanáček 


Výstraha zabezpečení
Informace, které si s tímto serverem vyměníte, nemohou zobrazit ani upravit jiní uživatelé. Nastaly však potíže s certifikátem zabezpečení serveru.
Certifikát zabezpečení vydala společnost, které nedůvěřujete. Prohlédněte si certifikát a rozhodněte se, zda chcete danému certifikačnímu úřadu důvěřovat.
Datum certifikátu zabezpečení je platné.
Název na certifikátu zabezpečení souhlasí s názvem stránky, kterou se pokoušíte zobrazit.
Chcete pokračovat?
Ano
Ne
Zobrazit certifikát
©Petr Hanáček
CLACRYPT Slide 11


Website Certified by an Unknown Authority
Something happened and you need to click OK to get on with things.
Certificate mismatch security identification administration communication intercept liliputian snotweasel foxtrot omegaforce.
Technical Crap ...
More techinical crap
Hoyvin-Glayvin!
Launch photon torpedos
OK
Cancel
Adapted from Jonathan Nightingale
©Petr Hanáček
CLACRYPT Slide 12

## Certifikát ve formátu X.509

```
Certificate ::= SIGNED SEQUENCE{
        version [0]     Version DEFAULT v1988,
        serialNumber    CertificateSerialNumber,
        signature       AlgorithmIdentifier,
        issuer          Name,
        validity        Validity,
        subject         Name,
        subjectPublicKeyInfo    SubjectPublicKeyInfo}

  Version ::=    INTEGER {v1988(0)}
  CertificateSerialNumber ::=    INTEGER

  Validity ::=    SEQUENCE{
        notBefore       UTCTime,
        notAfter        UTCTime}

  SubjectPublicKeyInfo ::=       SEQUENCE{
        algorithm              AlgorithmIdentifier,
        subjectPublicKey       BIT STRING}

  AlgorithmIdentifier ::= SEQUENCE{
        algorithm       OBJECT IDENTIFIER,
        parameters      ANY DEFINED BY algorithm OPTIONAL}
```

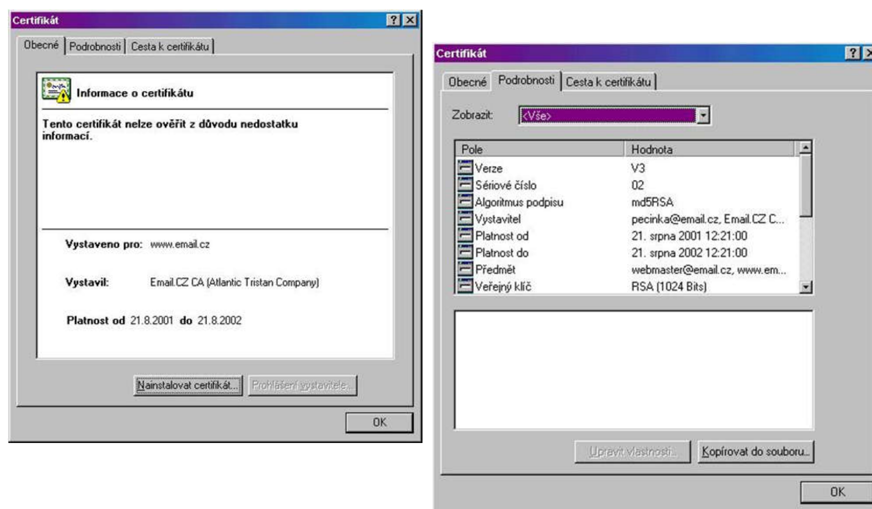
©Petr Hanáček CLACRYPT Slide 13

## Položky certifikátu

- *version* - standardně 0
- *serial number* (pořadové číslo certifikátu) - certifikátu obsahuje celé číslo
  - v „černých listinách" adresáře toto číslo (a jméno vydavatele certifikátu) tento certifikát jednoznačně identifikuje
  - certifikační autorita musí zajistit, aby čísla všech jí vydaných certifikátů byla navzájem jedinečná
- *issuer* - jméno vydávající certifikační autority
- *subject* - jméno vlastníka certifikátu
- *validity* - doba platnosti certifikátu
  - validity notBefore a notAfter (platnost ne před, platnost ne po)
    - » Podpis je platný, pouze je li datum jeho vytvoření v intervalu platnosti každého certifikátu z certifikační cesty
- *SubjectPublicKeyInfo* - VK subjektu *subject*
- *signature* - algoritmus podpisu certifikátu

©Petr Hanáček CLACRYPT Slide 14

# Certifikát - Prohlížeč certifikátu



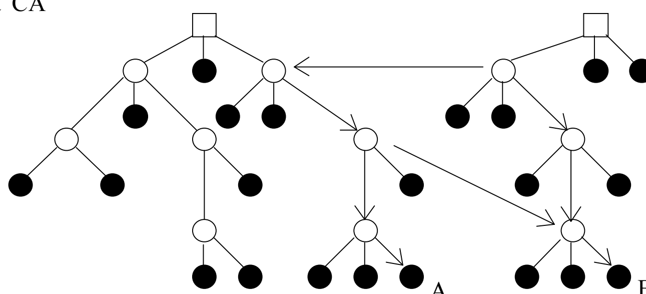



# Formát jména

- **formát**
  - je definován v Directory Information Model X.501 [1]
- **posloupnost „relativních jedinečných jmen“ Relative Distinguished Names (RDN), jako jsou jméno země, jméno organizace nebo osobní jméno**
- **složky jména dle X.520:**
  - » C countryName O organizationName
  - » OU organizationalUnitName S surname
  - » CN commonName L localityName
  - » SP stateOrProvinceName ST streetAddress
  - » T title SN serialNumber
  - » BC businessCategory D description
- **Jméno autora**
  - C = CZ, O = FEI, OU = UIVT, CN = Hanacek

# Křížové certifikáty

- Co v případě, že je třeba komunikovat mezi členy různých certifikačních stromů
  - vytvořit společnou kořenovou CA
    - » často není možné
  - křížové certifikáty



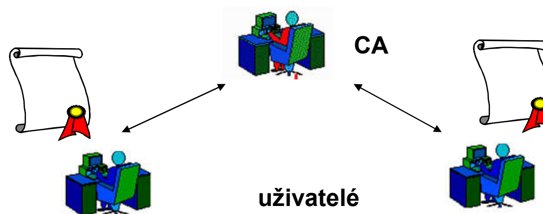



# Certifikační autorita

- Úkoly CA
  - registrace veřejných klíčů
  - distribuce certifikátů
  - rušení certifikátů
- Typy CA
  - kořenová CA
  - "policy" CA - vydává certifikáty CA
  - uživatelská CA - vydává certifikáty uživatelům

Certifikát

| Sériové číslo |
|---|
| Platnost (od, do) |
| Vydavatel (CA) |
| Vlastník (uživatel) |
| Veřejný klíč |
| Identifikace algoritmu |
| Podpis CA |



## Úkoly uživatelské CA

- uživatelská CA provádí
  - vydávání uživatelských certifikátů
  - údržbu záznamů ve svém adresáři
  - údržbu černých listin certifikátů vydaných a posléze odebraných touto CA
  - zaslání jména a nových veřejných klíčů kořenové CA formou spolehlivého křížového certifikátu při každé změně kořenové CA nebo jejích klíčů
  - zaslání dopředné certifikační cesty při změně kterékoli vyšší CA nebo jejích klíčů
  - aktualizaci hodnot uživatelských certifikátů v záznamech svých uživatelů v adresáři
- uživatelská CA neprovádí
  - generování klíčů
  - klíče si uživatelé generují sami
  - CA nemůže zaručit kvalitu nebo jedinečnost uživatelských klíčů



## Vydání certifikátu

- uživatel si lokálně vygeneruje klíče
- uživatelské CA zašle prototypový certifikát
- po ověření prototypového certifikátu uživatelská CA pošle uživateli zpět jeho podepsaný certifikát
- výhody decentralizovaného vytváření klíčů
  - celý proces je decentralizován a osvobozuje centrální orgán od práce. Uživatelé samotní provádí generování klíčů jen zřídka.
  - soukromé klíče, které jsou vysoce citlivé, nejsou nikdy přenášeny sítí.
  - je zaručeno, že soukromé klíče existují jen v jediném provedení a to v rukou uživatele

## Prototypový certifikát

- Je definován v „Privacy Enhanced Mail“ US Internet, Part IV
- má strukturu certifikátu X.509
- obsah
  - version: standardně 0
  - serial number: „1“ (bude změněno CA)
  - issuer” jméno CA
  - validity: zvolena uživatelem, může být omezena CA
  - subject: jméno uživatele
  - subjectPublicKeyInfo: identifikátor algoritmu, parametr a bitový řetězec veřejného klíče vytvořeného uživatelem

©Petr Hanáček 

## Klasický X.509: (1988)

- “The Directory: Authentication Framework”
- Součást dokumentu „OSI-Directory standard X.500“
- Definuje datové položky:
  - userCertificate; cACertificate
  - crossCertificatePair
  - certificateRevocationList
- Definuje mechanismy pro autentizaci
- Certifikát obsahuje DN uživatele
- Certifikát obsahuje DN vydávající CA

©Petr Hanáček 

## X.509v3 (1997)

- Nový mechanismus „rozšíření“ – „extensions“
- Předdefinovaná rozšíření:
  - Informace o klíči: identifier, usage, ...
  - Informace o certifikační politice: certificate policy, ...
  - Rozšíření týkající se uživatele a CA : alternative name, ...
  - Omezení na certifikační cestu
- Oddělování X.509v3 od X.500
  - Např. : změna DN
  - Nemusí být unikátní



## X.509v4 (2000)

- Možnost verifikace certifikační cesty obsahující CA z různých domén a hierarchií
- Rozšíření možností revokace cerifikátu
- Nové možnosti tzv. atributových certifikátů (AC)
- Definuje použití AC pro řízení přístupu a autorizaci

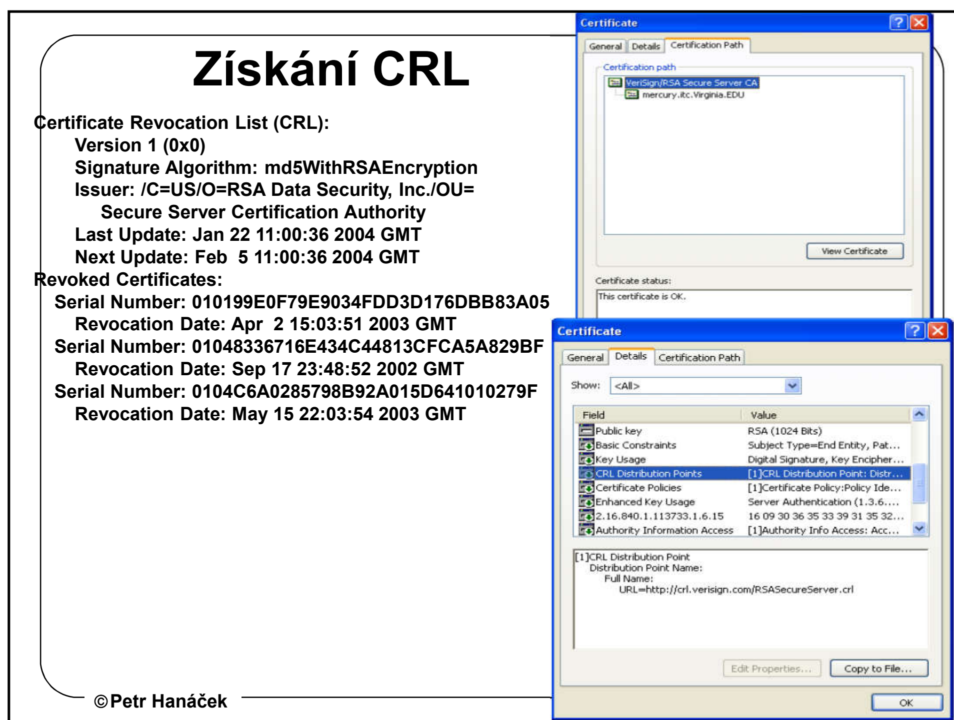
## Rušení certifikátu

- Certifikační autorita musí být schopna zrušit vydaný certifikát před skončením doby jeho platnosti.
- důvody zrušení certifikátu
  - byl prozrazen soukromý klíč uživatele
  - změnil se zaměstnavatel uživatele (příslušnost uživatele), čímž je neplatné jméno obsažené v certifikátu
  - uživatel již nemá být certifikován danou CA
  - soukromý klíč CA byl kompromitován
  - uživatel porušil bezpečnostní pravidla CA
- Dvě možnosti:
  - Zrušený certifikát je umístěn na “černou listinu” - seznam zrušených certifikátů - CRL (Certificate Revocation List)
  - Na stav certifikátu je možno se dotázat mechanismem Online Certificate Status Protocol (OCSP)



## CRL

- CRL je vlastně seznam zneplatněných a nevyexpirovaných certifikátů
- Podepsaný certifikační autoritou
- Příjemce certifikátu se dívá do seznamu, aby zjistil zda certifikát nebyl zneplatněn
- Dva modely
  - Pull model: Příjemce certifikátu si stahuje podle potřeby CRL od CA
  - Push model: CA pravidelně posílá CRL příjemcům certifikátů
- Problémy?
- Podobné jako blacklisty platebních karet
  - Seznam bývá hodně velký
    - Delta CRL
  - Časové zpoždění mezi okamžikem revokace a publikováním CRL
- Rozšířené CRL slouží příliš mnoha uživatelům a “push” model je nerealizovatelný
- Náchylné na DOS útoky
  - Co je implicitní? Přijmout certifikát nebo odmítnout?




# OCSP

- Online Certificate Status Protocol
- RFC 2560
- Protokol Request / Response
  - Příjemce certifikátu dostává aktuální informaci
- Odpověď je vytvářena na straně serveru
  - Zpět se posílá pouze relevantní informace, redukuje se množství přenášených dat
  - Server může požadovat, aby požadavky byly podepsány
    - » Umožňuje zpoplatnění
- Stále náchylné k DOS útokům

©Petr Hanáček

**Source:** http://www.openvalidation.org/whatisocsp/whatocsp.htm

CLACRYPT Slide 28

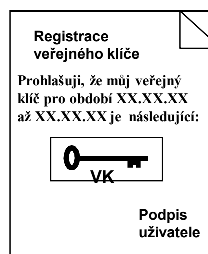
# Dokumentace

- **Certifikační politika (CP)**
  - Certificate Policy
  - Dokument nejvyšší úrovně, popisující vlastnosti certifikátu
  - Identifikována pomocí OID (Object ID)
- **CPS – Certification Practice Statement**
  - Detailní dokument, popisující, jak byl certifikát vydán

©Petr Hanáček 

# Míra důvěry v certifikáty

- **Třída 1**
  - certifikát zajišťuje pouze jedinečnost jména vlastníka
  - lze jej získat anonymně
- **Třída 2**
  - identita vlastníka musí být ověřena třetí stranou (notářsky ověřený formulář, zaslaný poštou)
- **Třída 3**
  - vlastník musí osobně navštívit CA
  - ověření osobní totožnosti
- **Třída 4**
  - Třída 3 + prokázání oprávněnosti žadatele požadovat certifikát
- **Registrace veřejných klíčů**
  - je třeba pořídit protokol o registraci veřejného klíče, aby uživatel nemohl popřít svůj veřejný klíč



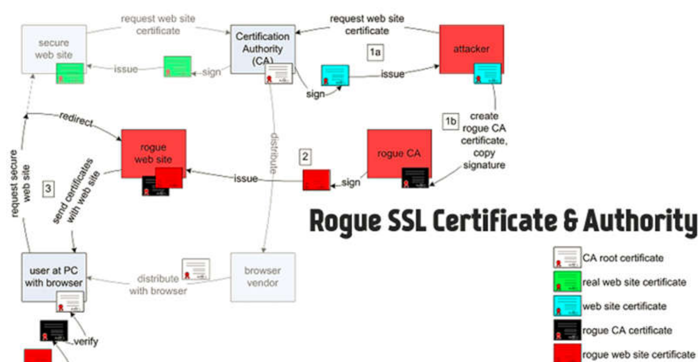

©Petr Hanáček 

## SSL Chain-of-Trust is Broken!

- Last year, Google discovered that Symantec (one of the CAs) had improperly issued a duplicate certificate for google.com to someone else, apparently mistakenly.
- In March 2011, Comodo, a popular Certificate Authority, was hacked to issue fraudulent certificates for popular domains, including mail.google.com, addons.mozilla.org, and login.yahoo.com.
- In the same year, the Dutch certificate authority DigiNotar was also compromised and issued massive amounts of fraudulent certificates.
- …

©Petr Hanáček CLACRYPT Slide 31

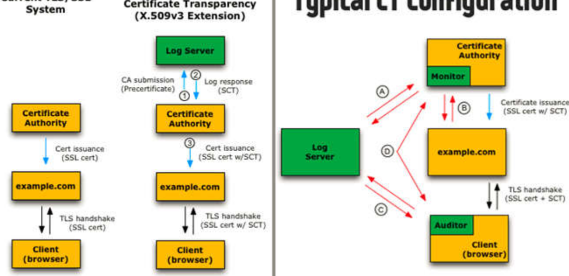
## Certificate Transparency




©Petr Hanáček CLACRYPT Slide 32
https://thehackernews.com/2016/04/ssl-certificate-transparency.html

## Certificate Transparency

- Certificate logs have three important qualities:
- 1. Append-only: Certificates records can only be added to a log. They can not be deleted, modified, or retroactively inserted into a log.
- 2. Cryptographically assured: Certificates Logs use a special cryptographic mechanism known as 'Merkle Tree Hashes' to prevent tampering.
- 3. Publicly auditable: Anyone can query a log and verify its behavior, or verify that an SSL certificate has been legitimately appended to the log.
- In CT, [...] T), which proves




©Petr Hanáček — CLACRYPT Slide 33

https://thehackernews.com/2016/04/ssl-certificate-transparency.html

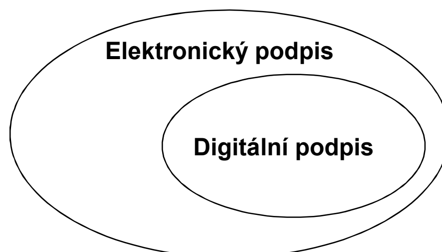
# Elektronický podpis

Petr Hanáček
Ústav informatiky a výpočetní techniky
Vysoké učení technické v Brně
Božetěchova 2
612 66 Brno
tel. 7275 216
e-mail: hanacek@dcse.fee.vutbr.cz

©Petr Hanáček — CLACRYPT Slide 34

# Elektronický vs. digitální

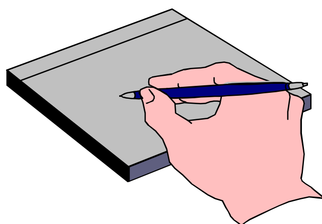
- **Elektronický podpis**
  - vložení textu podpisu do dokumentu
  - vložení naskenovaného obrázku vlastnoručního podpisu do dokumentu
  - ...
  - Digitální podpis
    - » založený na použití kryptografických mechanismů
      - v současnosti založený na použití kryptografie veřejným klíčem






# Funkce elektronického podpisu

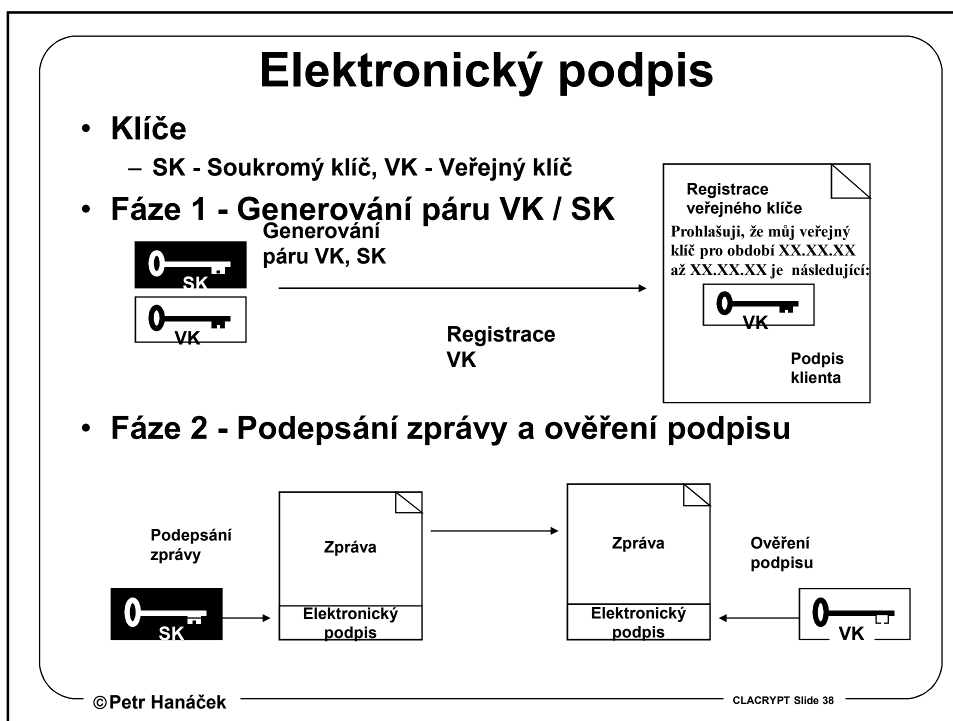
- **Zajišťuje autenticitu dokumentu**
  - Příjemce dokumentu bezpečně ví, kdo je autorem dokumentu.
- **Zajišťuje integritu dokumentu**
  - Příjemce dokumentu má jistotu, že obsah dokumentu dokument nebyl během přenosu nebo zpracování modifikován.
- **Zajišťuje nepopiratelnost autora dokumentu**
  - Autor dokumentu nemůže popřít autorství dokumentu ani jeho obsah.



## Porovnání vlastností

| | Manuální podpis | Elektronick |
|---|---|---|
| **Autenticita** | Ano | An |
| **Integrita** | Nedostatečně | An |
| **Neodmítnutelnost zodpovědnosti** | Ano | An |
| **Rozlišení originálu od kopie** | Ano | Ne |
| **Padělatelnost** | Ano | Při dodržení |
| **Vytvoření** | Fyzická přítomnost osoby | Znalost ta (soukromé |

©Petr Hanáček CLACRYPT Slide 37



## Obsah certifikátu

- Obsah vyžadovaný ze zákona
  - Jméno vlastníka podpisového klíče, který je pro případ záměny opatřen dodatkem, nebo nezaměnitelný pseudonym vlastníka podpisového klíče, který musí být jedinečný,
  - Veřejný podpisový klíč,
  - Názvy algoritmů, pomocí kterých lze používat veřejné klíče vlastníka podpisového klíče a veřejné klíče certifikační autority,
  - Pořadové (běžné) číslo certifikátu,
  - Začátek a konec platnosti certifikátu,
  - Jméno certifikační autority,
  - Údaje týkající se případného omezení použití podpisového klíče podle druhu a rozsahu specifických aplikací.



## Protokoly a aplikace

## Bezpečnostní služby ISO 7498-2

- Autentizace
  - Autentizace spojení
  - Autentizace odesílatele
- Řízení přístupu
- Důvěrnost
  - Důvěrnost spojení
  - Důvěrnost přenosu zpráv
  - Důvěrnost toku dat
- Integrita
  - Integrita spojení s opravou,
  - Integrita spojení bez opravy
  - Integrita přenosu zpráv
- Nepopiratelnost
  - Nepopiratelnost odesílatele
  - Nepopiratelnost doručení

©Petr Hanáček 

## PKCS: Public Key Cryptography Standards

- Počátek v roce 1991 jako implementační dohoda (průmyslový standard) mezi firmami, které začaly jako první používat kryptografii veřejným klíčem
  - Apple, Digital, Lotus, Microsoft, MIT, Northern Telecom, Novell, Sun
- Revidovány v roce 1993, Cryptoki (PKCS #11) vytvořeno 1995
- Stále aktualizovány

©Petr Hanáček 

## PKCS #1: RSA Cryptography

- **Šifrování a podpis algoritmem RSA**
- **v1.5 (1993) definovala základní schéma RSA, použité v SSL, S/MIME**
- **v2.0 (1998) přidala mechanismus Bellare-Rogaway OAEP** (Optimal Asymmetric Encryption )
- **v2.1 přidává B-R PSS (Probabilistic Signature Scheme)**



## PKCS #7: Cryptographic Message Syntax

- **Syntaxe zašiforvané a podepsané zprávy**
  - **Rozšíření oproti PEM (Privacy-Enhanced Mail)**
- **v1.5 (1993) definuje správu klíčů založenou na RSA, je základem pro S/MIME**
- **Dokument IETF RFC 2630 přidává správu klíčů pomocí algoritmu Diffie-Hellman**
- **v1.6bis obsahuje podporu pro protokol SET**

## PKCS #11: Cryptographic Token Interface (Cryptoki)

- Jednotné programátorské rozhraní pro čipové karty a jiná zařízení
- v1.0 (1995) základní metody
- v2.01 (1997) dodány další kryptografické mechanismy, zlepšená správa zařízení
- v2.1 Přehlednější rozhraní, další mechanismy

©Petr Hanáček 

## PKCS #15: Cryptographic Token Information Format

- Formát souborů pro kryptografická data na čipových kartách a jiných zařízeních
- v1.0 (1998) základní verze
- v1.1 přidává formáty pro uložení software

©Petr Hanáček 



©

## Další dokumenty PKCS

- PKCS #3: Diffie-Hellman Key-Agreement
- PKCS #5: Password-Based Cryptography
- PKCS #8: Private-Key Information Syntax
- PKCS #9: Selected Attribute Types
- PKCS #10: Certification Request Syntax
- PKCS #12: Personal Information Exchange Syntax

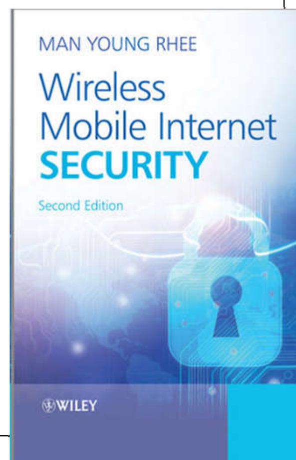
- #2, #4, #6 jsou již zastaralé, #13, #14 jsou ve vývoji

©Petr Hanáček CLACRYPT Slide 47

## Učebnice

5

- Man Young Rhee: Wireless Mobile Internet Security, 2nd Edition, ISBN: 978-1-118-49653-4XX
- Kapitoly
  - Kapitola 10 - Electronic Mail Security: PGP, S/MIME
    - » Zajímavá je pro nás celá kapitola v rozsahu slajdů.

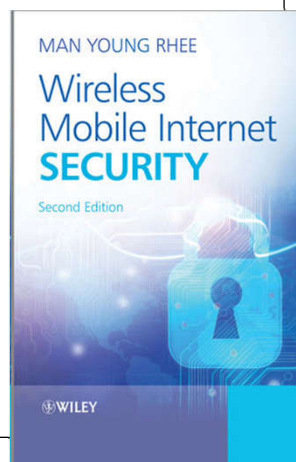



©Petr Hanáček

## Učebnice



- Man Young Rhee: Wireless Mobile Internet Security, 2nd Edition, ISBN: 978-1-118-49653-4XX
- Kapitoly
  - Kapitola 9 - Transport Layer Security: SSLv3 and TLSv1
    - Zajímavá je pro nás celá kapitola v rozsahu slajdů.




©Petr Hanáček

## Bezpečná elektronická pošta

Petr Hanáček
Ústav informatiky a výpočetní techniky
Vysoké učení technické v Brně
Božetěchova 2
612 66 Brno
tel. 7275 216
e-mail: hanacek@dcse.fee.vutbr.cz

©Petr Hanáček

## PEM a PGP

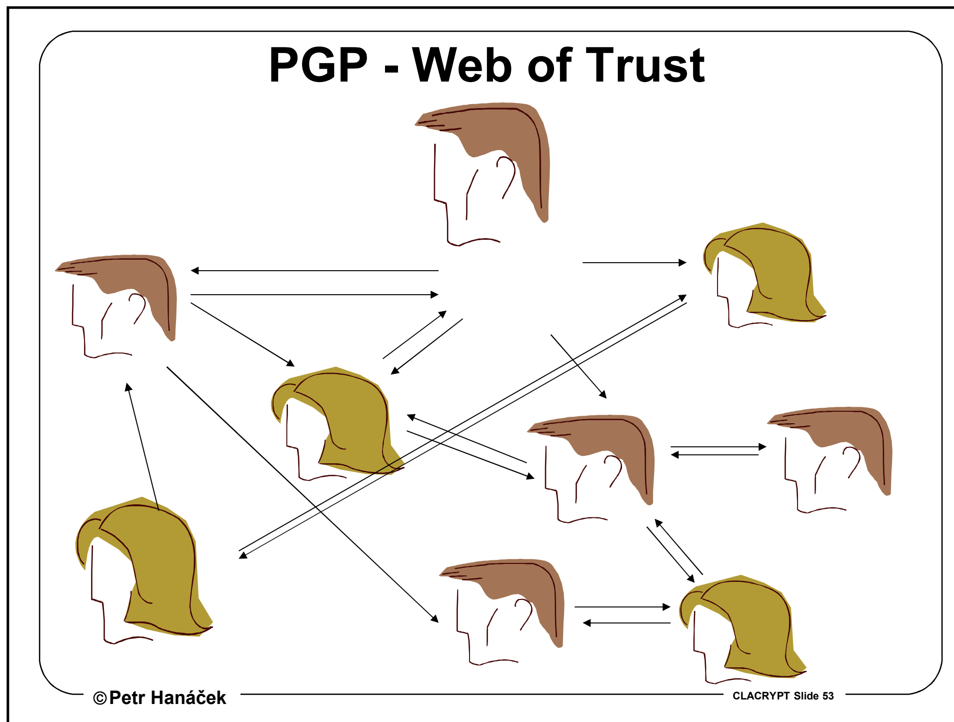
- **PEM (Privacy Enhanced Mail - bezpečná elektronická pošta)**
  - návrh standardu, definující procedury pro šifrování a autentizaci zpráv elektronické pošty při zlepšení bezpečnosti v rámci sítě Internet.
  - definována dokumenty RFC 1421 až 1424
  - implementace TIS/PEM (Privacy Enhanced Mail od firmy Trusted Information Systems) nebo RIPEM
- **PGP (Pretty Good Privacy)**
  - aplikační software s kryptografickými vlastnostmi, vykazujícími vysoký stupeň bezpečnosti
  - PGP verze 1 definoval a vyvinul Philip Zimmerman z Pretty Good Software
  - verze 2 byla vyvinuta pod jeho vedením

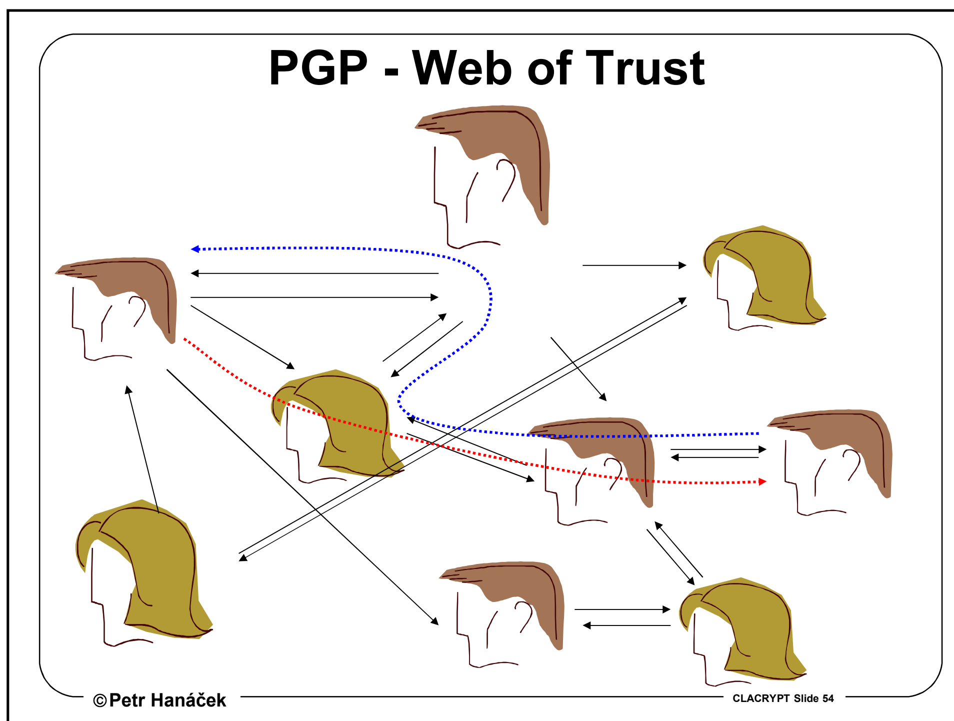
©Petr Hanáček CLACRYPT Slide 51

## PGP - Pretty Good Privacy

- **autor Philip Zimmerman**
- **vytvořen v r. 1991**
- **poskytuje**
  - zašifrování zpráv
    - » důvěrnost
  - elektronický podpis
    - » autentizace
    - » integrita
    - » neodmítnutelnost zodpovědnosti

©Petr Hanáček CLACRYPT Slide 52


PGP - Web of Trust
©Petr Hanáček
CLACRYPT Slide 53


PGP - Web of Trust
©Petr Hanáček
CLACRYPT Slide 54

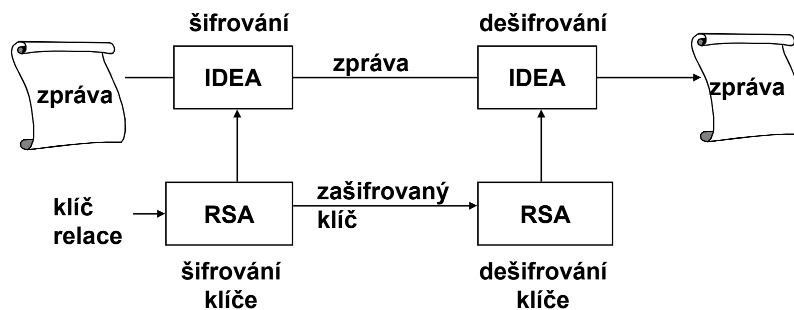
## Služby PGP

- **Šifrování zprávy**
  - IDEA, RSA
    - » Zpráva je zašifrovaná algoritmem IDEA náhodným klíčem relace. Klíč relace je zašifrován veřejným klíčem příjemce pomocí RSA a zaslán se zprávou.
- **Elektronický podpis**
  - RSA, MD5
    - » Pomocí MD5 je spočten hash zprávy a je zašifrován pomocí RSA soukromým klíčem odesílatele.
- **Komprese dat**
  - ZIP
- **Kódování do textové podoby**
  - Radix64
- **Segmentace**
    - » Dlouhá zpráva může být rozdělena na segmenty

©Petr Hanáček CLACRYPT Slide 55

## Důvěrnost

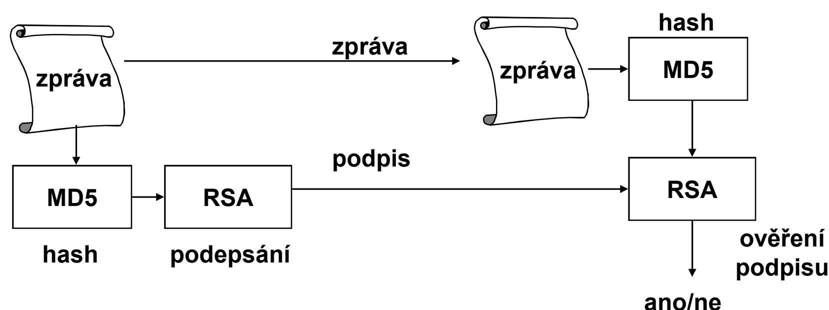
- **Zašifrování**
  - důvěrnost




©Petr Hanáček CLACRYPT Slide 56

### Autentizace, integrita a nepopiratelnost

- Elektronický podpis
  - autentizace odesílatele
  - integrita zprávy
  - nepopiratelnost odesílatele




©Petr Hanáček — CLACRYPT Slide 57

## Základní příkazy PGP

- Zašifrování textu VK příjemce:
  - pgp -e textfile her_userid
- Podepsání textu vlastním soukromým klíčem:
  - pgp -s textfile [-u your_userid]
- Podepsání textu vlastním soukromým klíčem a zašifrování textu VK příjemce:
  - pgp -es textfile her_userid [-u your_userid]
- Zašifrování textu heslem:
  - pgp -c textfile
- Dešifrování textu nebo ověření podpisu:
  - pgp ciphertextfile [-o plaintextfile]
- Zašifrování textu pro několik příjemců:
  - pgp -e textfile userid1 userid2 userid3

©Petr Hanáček — CLACRYPT Slide 58

# WinPGP

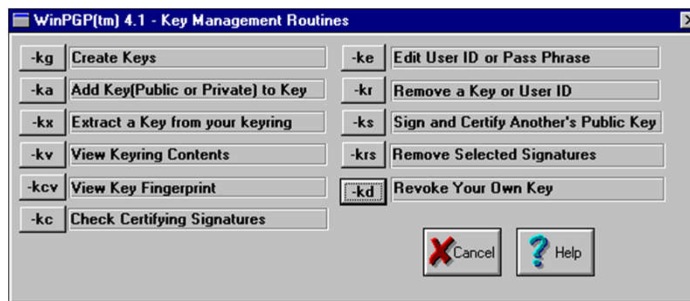
- WinPGP - grafické uživatelské rozhraní k PGP

# Správa klíčů PGP

- Generování dvojice klíčů RSA:
  » pgp -kg
- Přidání klíče do okruhu klíčů:
  » pgp -ka keyfile [keyring]
- Zobrazení obsahu okruhu klíčů:
  » pgp -kv[v] [userid] [keyring]
- Zobrazení kontrolního součtu veřejného klíče:
  » pgp -kvc [userid] [keyring]
- Zobrazení obsah okruhu klíčů a ověření podpisů:
  » pgp -kc [userid] [keyring]
- Změna parametrů důvěry veřejného klíče:
  » pgp -ke userid [keyring]
- Podepsání a certifikace něčího VK:
  » pgp -ks her_userid [-u your_userid] [keyring]
- Permanentní zrušení svého VK:
  » pgp -kd your_userid

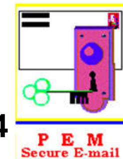
# WinPGP - správa klíčů

- Správa klíčů ve WinPGP




©Petr Hanáček CLACRYPT Slide 61

# PEM - Privacy Enhanced Mail




- standard Internet
- definován dokumenty RFC 1421 až RFC 1424
- podporuje symetrickou i asymetrickou správu klíčů
- certifikáty veřejných klíčů dle X.509, vlastní hierarchie certifikačních autorit
- implementace TIS/PEM (Privacy Enhanced Mail od firmy Trusted Information Systems) nebo RIPEM
- poskytuje
  - zašifrování zpráv
    - » důvěrnost
  - elektronický podpis
    - » autentizace
    - » integrita
    - » neodmítnutelnost zodpovědnosti (pokud je použita asymetrická správa klíčů)

©Petr Hanáček CLACRYPT Slide 62

## Služby PEM

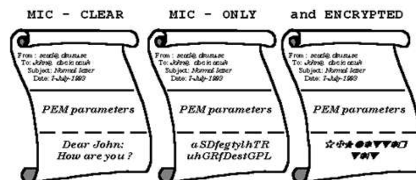
- **Šifrování zprávy**
  - » Zpráva je zašifrovaná algoritmem DES-CBC náhodným klíčem relace.
  - – DES-CBC, RSA
    - Klíč relace je zašifrován VK příjemce pomocí RSA
  - – DES-CBC, DES-ECB nebo DES-EDE
    - – Klíč relace je zašifrován domluveným tajným klíčem
- **Elektronický podpis (asymetrická kryptografie)**
  - – RSA, MD2 nebo MD5
    - » Pomocí MD2 nebo MD5 je spočten hash zprávy a je zašifrován pomocí RSA soukromým klíčem odesílatele.
- **Autentizace (symetrická kryptografie)**
  - » MD2 nebo MD5, DES-ECB nebo DES-EDE
- **Komprese dat - není**
- **Kódování do textové podoby - Radix64**
- **Segmentace - není**



## Způsoby zabezpečení

- **Fáze**
  - – 1) Konverze zprávy do kanonické formy
    - » 7bitový kód, konce řádků, délky řádků
  - – 2) Generování autentizačního a integritního zabezpečení
  - – 3) Zašifrování zprávy
  - – 4) Konverze do textové podoby
- **Typy zabezpečení**
  - – MIC-CLEAR - kroky 1, 2
  - – MIC-ONLY - kroky 1, 2, 4
  - – ENCRYPTED - kroky 1, 2, 3, 4

MIC - CLEAR MIC - ONLY and ENCRYPTED



## Příklad zprávy PEM

- Symetrická kryptografie

```
-----BEGIN PRIVACY-ENHANCED MESSAGE-----
   Proc-Type: 4,ENCRYPTED
   Content-Domain: RFC822
   DEK-Info: DES-CBC,F8143EDE5960C597
   Originator-ID-Symmetric: linn@zendia.enet.dec.com,,
   Recipient-ID-Symmetric: linn@zendia.enet.dec.com,ptf-kmc,3
   Key-Info: DES-ECB,RSA-MD2,9FD3AAD2F2691B9A,
             B70665BB9BF7CBCDA60195DB94F727D3
   Recipient-ID-Symmetric: pem-dev@tis.com,ptf-kmc,4
   Key-Info: DES-ECB,RSA-MD2,161A3F75DC82EF26,
             E2EF532C65CBCFF79F83A2658132DB47

   LLrHB0eJzyhP+/fSStdW8okeEnv47jxe7SJ/iN72ohNcUk2jHEUSoH1nvNSIWL9M
   8tEjmF/zxB+bATMtPjCUWbz8Lr9wloXIkjHUlBLpvXR0UrUzYbkNpk0agV2IzUpk
   J6UiRRGcDSvzrsoK+oNvqu6z7Xs5Xfz5rDqUcMlK1Z6720dcBWGGsDLpTpSCnpot
   dXd/H5LMDWnonNvPCwQUHt==
   -----END PRIVACY-ENHANCED MESSAGE-----
```

©Petr Hanáček CLACRYPT Slide 65

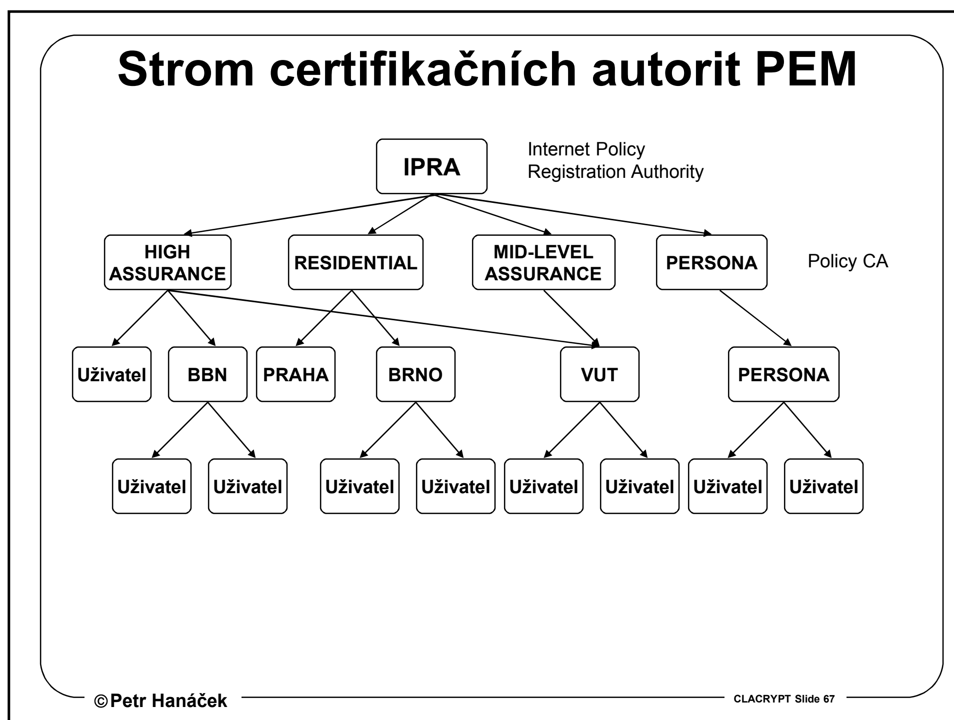
## Příklad zprávy PEM

- Asymetrická kryptografie

```
-----BEGIN PRIVACY-ENHANCED MESSAGE-----
   Proc-Type: 4,ENCRYPTED
   Content-Domain: RFC822
   DEK-Info: DES-CBC,BFF968AA74691AC1
   Originator-Certificate:
    MIIBlTCCAScCAWUwDQYJKoZIhvcNAQECBQAwUTELMAkGA1UEBhMCVVMxIDAeBgNV
    5XUXGx7qusDgHQGs7Jk9W8CW1fuSWUgN4w==
   Key-Info: RSA,
    I3rRIGXUGWAF8js5wCzRTkdhO34PTHdRZY9Tuvm03M+NM7fx6qc5udixps2Lng0+
    wGrtiUm/ovtKdinz6ZQ/aQ==
   Issuer-Certificate:
    MIIB3DCCAUgCAQowDQYJKoZIhvcNAQECBQAwTzELMAkGA1UEBhMCVVMxIDAeBgNV
    EREZd9++32ofGBIXaialnOgVUn0OzSYgugiQ077nJLDUj0hQehCizEs5wUJ35a5h
   MIC-Info: RSA-MD5,RSA,
    UdFJR8u/TIGhfH65ieewe2lOW4tooa3vZCvVNGBZirf/7nrgzWDABz8w9NsXSexv
    AjRFbHoNPzBuxwmOAFeA0HJszL4yBvhG
   Recipient-ID-Asymmetric:
    MFExCzAJBgNVBAYTAlVTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cml0eSwgSW5j
    LjEPMA0GA1UECxMGQmV0YSAxMQ8wDQYDVQQLEwZOT1RBUlk=,66
   Key-Info: RSA,
    O6BS1ww9CTyHPtS3bMLD+L0hejdvX6Qv1HK2ds2sQPEaXhX8EhvVphHYTjwekdWv
    7x0Z3Jx2vTAhOYHMcqqCjA==

   qeWlj/YJ2Uf5ng9yznPbtD0mYloSwIuV9FRYx+gzY+8iXd/NQrXHfi6/MhPfPF3d
   jIqCJAxvld2xgqQimUzoS1a4r7kQQ5c/Iua4LqKeq3ciFzEv/MbZhA==
   -----END PRIVACY-ENHANCED MESSAGE-----
```

©Petr Hanáček CLACRYPT Slide 66


Strom certifikačních autorit PEM
IPRA
Internet Policy
Registration Authority
HIGH ASSURANCE
RESIDENTIAL
MID-LEVEL ASSURANCE
PERSONA
Policy CA
Uživatel
BBN
PRAHA
BRNO
VUT
PERSONA
Uživatel
Uživatel
Uživatel
Uživatel
Uživatel
Uživatel
Uživatel
Uživatel
©Petr Hanáček
CLACRYPT Slide 67

S/MIME
• S/MIME (Secure/MIME)
– specifikace bezpečné výměny zpráv
– zastřešení průmyslového standardu MIME
– aplikováním standardů asymetrické kryptografie PKCS
• Internet Drafts
– S/MIME Message Specification
– S/MIME Certificate Handling
• Certifikáty veřejných klíčů dle X.509
• Poskytuje
– zašifrování zpráv
» důvěrnost
– elektronický podpis
» autentizace
» integrita
» neodmítnutelnost zodpovědnosti
S/MIME ENABLED
RSA
©Petr Hanáček
CLACRYPT Slide 68

# Služby S/MIME

- Šifrování zprávy
  - Zpráva je zašifrovaná symetrickým algoritmem náhodným klíčem relace.
    - » DES, 3DES nebo RC2
- Elektronický podpis
  - RSA, MD5 nebo SHA-1
    - » Pomocí MD5 nebo SHA-1 je spočten hash zprávy a je zašifrován pomocí RSA soukromým klíčem odesílatele.
- Formát dat
  - podle standardů PKCS
    - » "PKCS #1: RSA Encryption", [PKCS-1]
    - » "PKCS #7: Cryptographic Message Syntax", [PKCS-7]
    - » "PKCS #10: Certification Request Syntax", [PKCS-10]
- Kódování do textové podoby - MIME

©Petr Hanáček 

# Tvorba zabezpečovaných zpráv

- Podepsaná a zašifrovaná zpráva
  - podle standardu PKCS #7 šifrováním a/nebo podpisem, pak se přidají hlavičky podle normy MIME
  - Content–Type: Application/pkcs7-mime
  - výsledek se pomocí kódování BASE–64 připraví pro vyslání jako ASCII text
- Pouze podepsaná zpráva
  - čitelný text a podpis rozdělen např.
  - Content–Type: multipart/signed; protocol= Application/pkcs7-signature; micalg=rsa-md5; boundary= …
  - a vlastní čitelná zpráva se bude vypisovat samostatně oddělená pomocí … od podpisové části.

©Petr Hanáček 

## Typy zabezpečených zpráv

- **Standard PKCS #7 rozlišuje šest typů obsahu zpráv**
  - **Data (data) – obecná nešifrovaná a nepodepsaná data**
  - **Podepsaná data (signedData) – nešifrovaná digitálně podepsaná data**
  - **Šifrovaná data pro přenos (envelopedData) – šifrovaná data určená pro přenos.**
  - **Podepsaná šifrovaná data pro přenos (signedAndEnvelopedData) – šifrovaná digitálně podepsaná data určená pro přenos.**
  - **Data s charakteristikou (digestData)**
  - **Šifrovaná data pro uložení (encryptpedData) – šifrovaná data určená pro uložení např. na lokálním disku. Šifrovací klíč je odvozen z hesla. Neukládá se společně s šifrovanými daty. Předpokládá se, že správa šifrovacích klíčů je dělána off-line, jiným**

©Petr Hanáček 

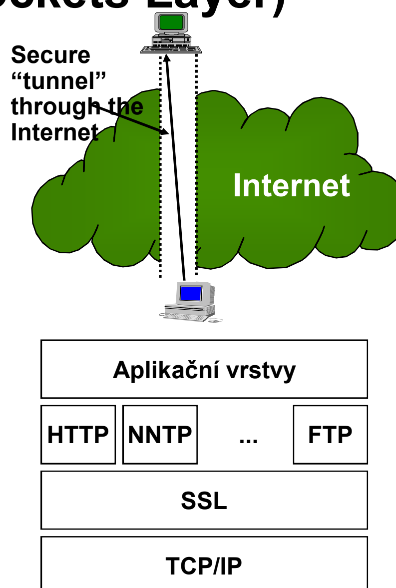
## SSL

©Petr Hanáček 

©

## SSL (Secure Sockets Layer)

- zabezpečení TCP/IP na transportní úrovni
  - lze pomocí něj implementovat různé služby
  - např. HTTPS = HTTP + SSL
- Podpora různých aplikací a protokolů
- Vyžaduje spolehlivou transportní vrstvu (TCP)
- SSL není skryto před aplikací (aplikaci je třeba modifikovat)
- SSL Handshake Protocol
  - Naváže šifrované spojení symetrickým algoritmem
- SSL Record Protocol
  - Stará se o skládání paketů, šifrování a dešifrování




| Aplikační vrstvy | | | |
|---|---|---|---|
| HTTP | NNTP | ... | FTP |
| SSL | | | |
| TCP/IP | | | |

©Petr Hanáček 

## SSL - Poskytované služby

- Autentizace
  - Kryptografické ověření identity obou komunikujících stran
  - Autentizace klient-server a server-klient
- Důvěrnost
  - Zašifrování dat, přenášených kanálem
- Integrita
  - Ochrana dat, přenášených kanálem, proti modifikaci
- ~~Nepopiratelnost - není poskytována~~
  - ~~Nepopiratelnost odeslání~~
  - ~~Nepopiratelnost příjmu~~

©Petr Hanáček 

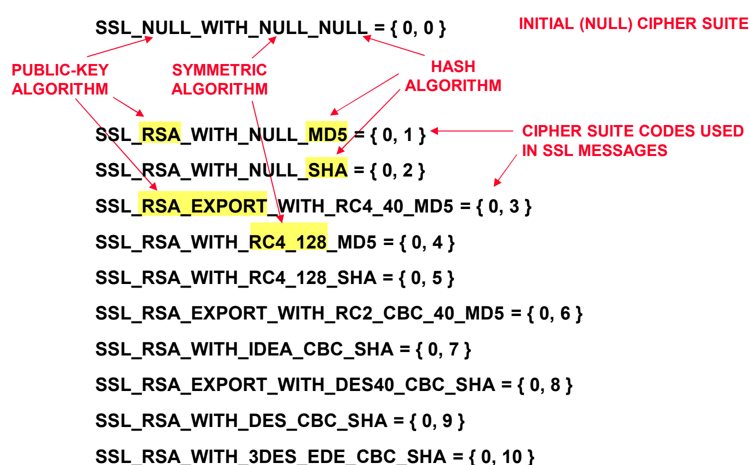
©

# SSL - Kryptografické mechanismy

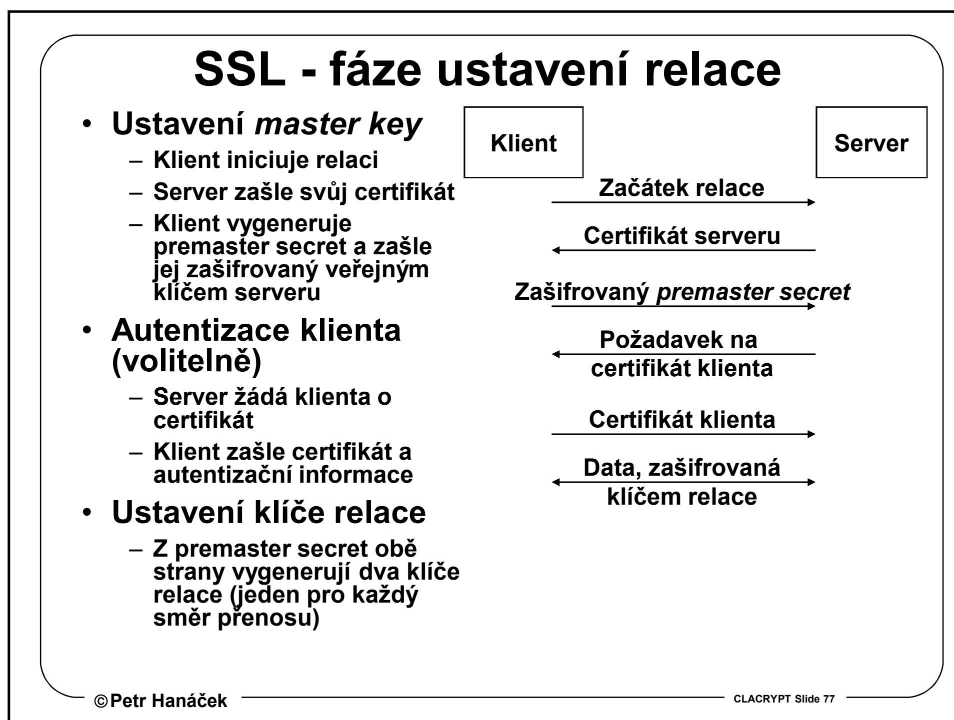
- Certifikáty
  - certifikáty ve formátu X.509
- Sada algoritmů (Cipher Suite)
  - Algoritmus s veřejným klíčem
    - » RSA
    - » Diffie-Hellman
  - Hašovací algoritmus
    - » MD5, SHA
  - Symetrická šifra
    - » RC2, RC4, IDEA-CBC, DES-CBC, 3DES-CBC

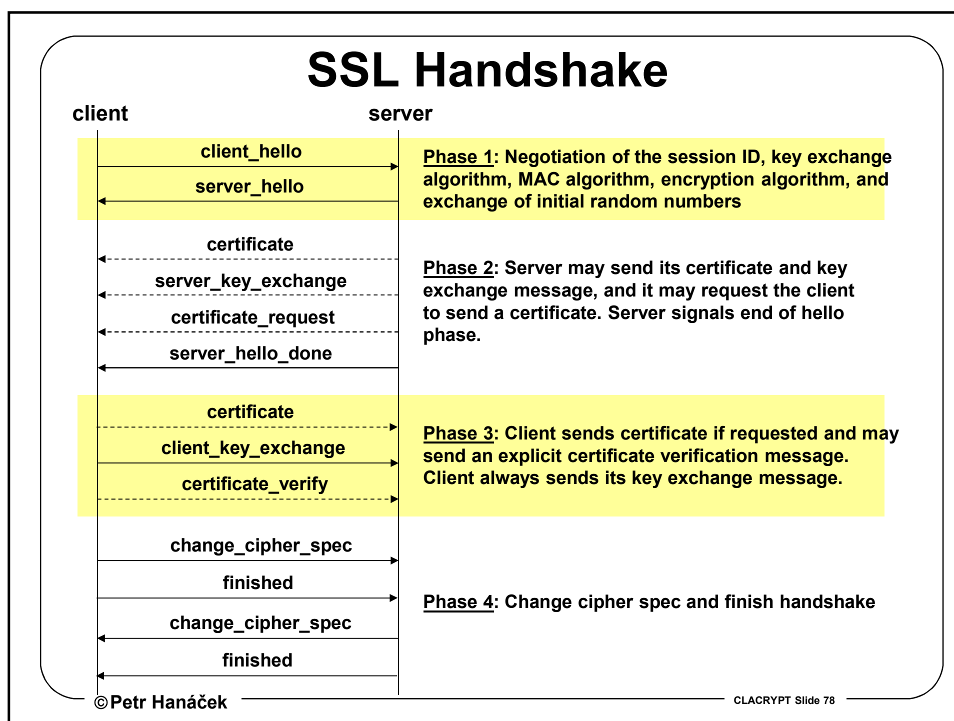
©Petr Hanáček CLACRYPT Slide 75

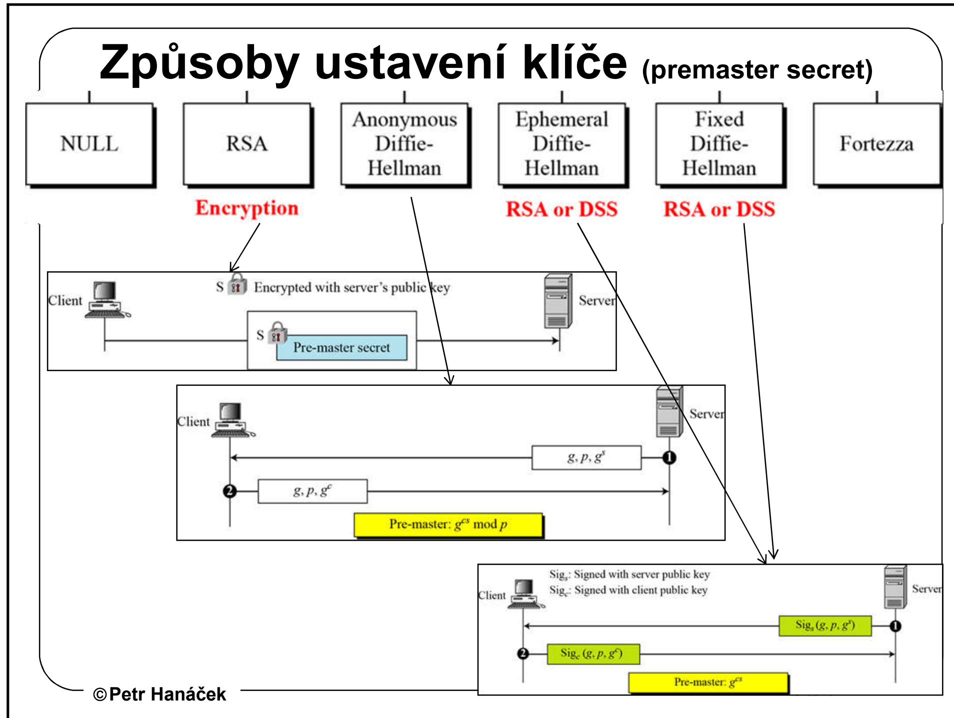
# Cipher Suite




©Petr Hanáček CLACRYPT Slide 76


SSL - fáze ustavení relace
• Ustavení master key
– Klient iniciuje relaci
– Server zašle svůj certifikát
– Klient vygeneruje premaster secret a zašle jej zašifrovaný veřejným klíčem serveru
• Autentizace klienta (volitelně)
– Server žádá klienta o certifikát
– Klient zašle certifikát a autentizační informace
• Ustavení klíče relace
– Z premaster secret obě strany vygenerují dva klíče relace (jeden pro každý směr přenosu)
Klient
Server
Začátek relace
Certifikát serveru
Zašifrovaný premaster secret
Požadavek na certifikát klienta
Certifikát klienta
Data, zašifrovaná klíčem relace
©Petr Hanáček
CLACRYPT Slide 77


SSL Handshake
client
server
client_hello
server_hello
Phase 1: Negotiation of the session ID, key exchange algorithm, MAC algorithm, encryption algorithm, and exchange of initial random numbers
certificate
server_key_exchange
certificate_request
server_hello_done
Phase 2: Server may send its certificate and key exchange message, and it may request the client to send a certificate. Server signals end of hello phase.
certificate
client_key_exchange
certificate_verify
Phase 3: Client sends certificate if requested and may send an explicit certificate verification message. Client always sends its key exchange message.
change_cipher_spec
finished
change_cipher_spec
finished
Phase 4: Change cipher spec and finish handshake
©Petr Hanáček
CLACRYPT Slide 78




# Cipher Suite SSL

| *Cipher suite* | *Key Exchange* | *Encryption* | *Hash* |
|---|---|---|---|
| SSL_NULL_WITH_NULL_NULL | NULL | NULL | NULL |
| SSL_RSA_WITH_NULL_MD5 | RSA | NULL | MD5 |
| SSL_RSA_WITH_NULL_SHA | RSA | NULL | SHA-1 |
| SSL_RSA_WITH_RC4_128_MD5 | RSA | RC4 | MD5 |
| SSL_RSA_WITH_RC4_128_SHA | RSA | RC4 | SHA-1 |
| SSL_RSA_WITH_IDEA_CBC_SHA | RSA | IDEA | SHA-1 |
| SSL_RSA_WITH_DES_CBC_SHA | RSA | DES | SHA-1 |
| SSL_RSA_WITH_3DES_EDE_CBC_SHA | RSA | 3DES | SHA-1 |
| SSL_DH_anon_WITH_RC4_128_MD5 | DH_anon | RC4 | MD5 |
| SSL_DH_anon_WITH_DES_CBC_SHA | DH_anon | DES | SHA-1 |
| SSL_DH_anon_WITH_3DES_EDE_CBC_SHA | DH_anon | 3DES | SHA-1 |
| SSL_DHE_RSA_WITH_DES_CBC_SHA | DHE_RSA | DES | SHA-1 |
| SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA | DHE_RSA | 3DES | SHA-1 |
| SSL_DHE_DSS_WITH_DES_CBC_SHA | DHE_DSS | DES | SHA-1 |
| SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA | DHE_DSS | 3DES | SHA-1 |
| SSL_DH_RSA_WITH_DES_CBC_SHA | DH_RSA | DES | SHA-1 |
| SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA | DH_RSA | 3DES | SHA-1 |
| SSL_DH_DSS_WITH_DES_CBC_SHA | DH_DSS | DES | SHA-1 |
| SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA | DH_DSS | 3DES | SHA-1 |
| SSL_FORTEZZA_DMS_WITH_NULL_SHA | Fortezza | NULL | SHA-1 |
| SSL_FORTEZZA_DMS_WITH_FORTEZZA_CBC_SHA | Fortezza | Fortezza | SHA-1 |
| SSL_FORTEZZA_DMS_WITH_RC4_128_SHA | Fortezza | RC4 | SHA-1 |

©Petr Hanáček 

# KRY

## Vytvoření klíčů

- **Premaster secret**
  - Vytvořený klientem
  - Jednoduchý: 2 bytes of SSL version + 46 random bytes
  - Zasílá se serveru zašifrovaný veřejným klíčem serveru
- **Master secret**
  - Vytvořený oběma účastníky z premaster secret a náhodných hodnot klienta a serveru (Client Random, Server Random)
- **Klíčový materiál (Key material)**
  - Vytvořený z master secret a náhodných hodnot klienta a serveru (Client Random, Server Random)
- **Klíče**
  - Vytvořené z klíčového materiálu



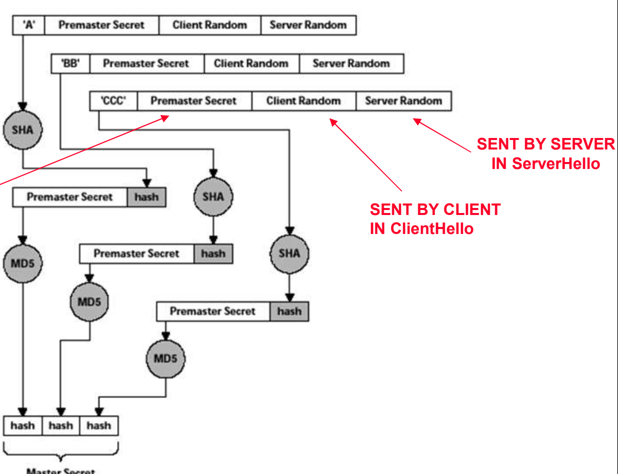
## Vytvoření Master Secret

SERVER'S PUBLIC KEY IS SENT BY SERVER IN ServerKeyExchange

CLIENT GENERATES THE PREMASTER SECRET

ENCRYPTS WITH PUBLIC KEY OF SERVER

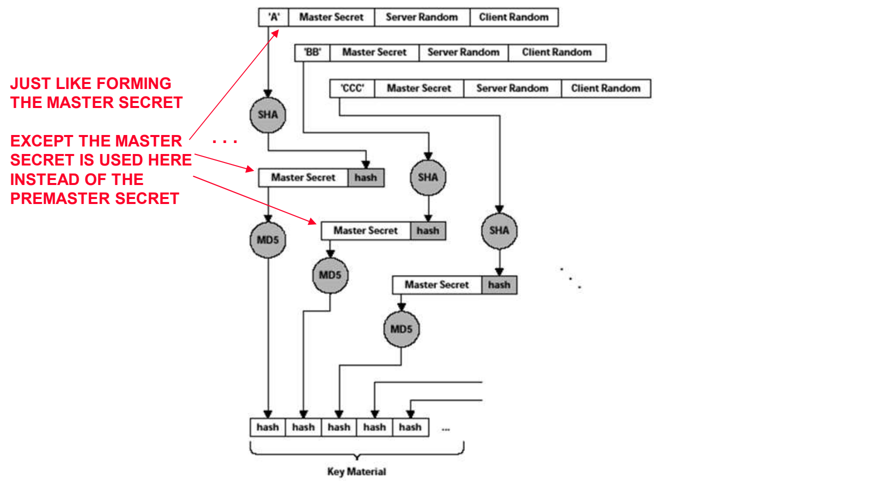
CLIENT SENDS PREMASTER SECRET IN ClientKeyExchange




MASTER SECRET IS 3 MD5 HASHES CONCATENATED TOGETHER = 384 BITS

SOURCE: THOMAS, *SSL AND TLS ESSENTIALS*

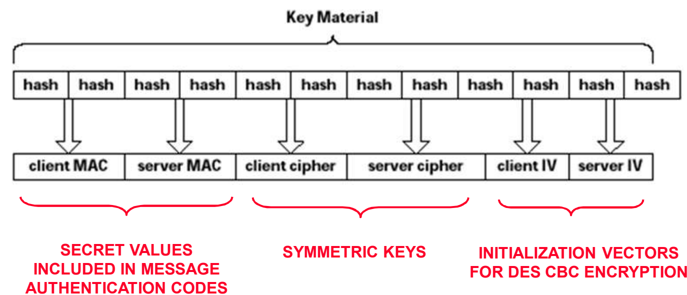
# Vytvoření Klíčového materiálu




SOURCE: THOMAS, *SSL AND TLS ESSENTIALS*

©Petr Hanáček 

# Vytvoření klíčů z klíčového materiálu




SECRET VALUES INCLUDED IN MESSAGE AUTHENTICATION CODES

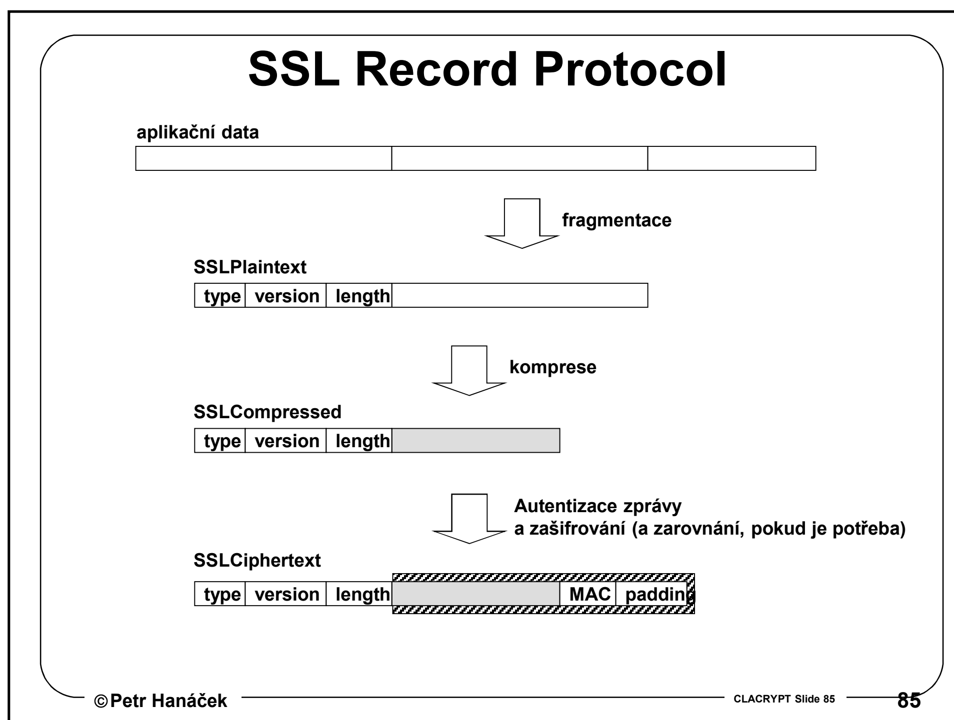
SYMMETRIC KEYS

INITIALIZATION VECTORS FOR DES CBC ENCRYPTION

SOURCE: THOMAS, *SSL AND TLS ESSENTIALS*

©Petr Hanáček 

©

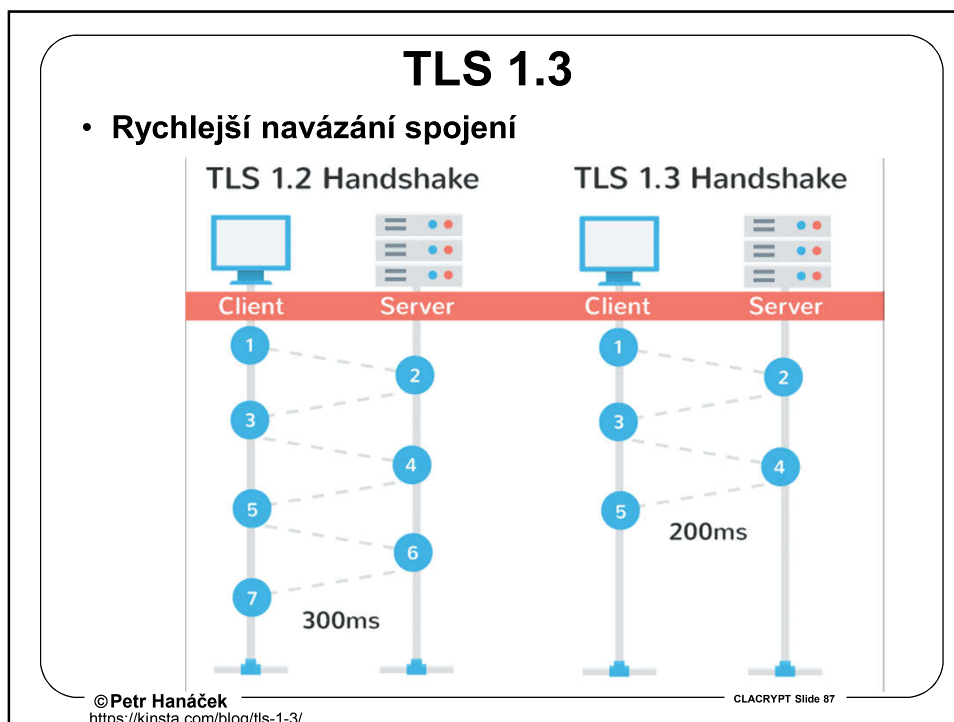


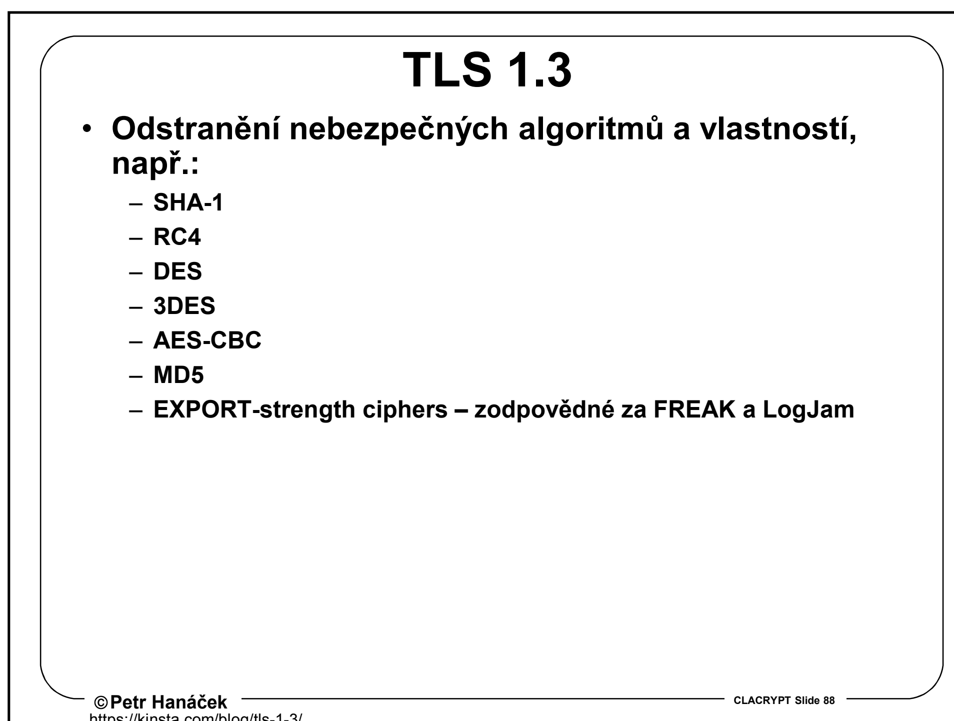

# SSL a TLS

- **SSL 1.0**
  - Interní návrh Netscape, někdy v roce 1994, nikdy nepoužívána
- **SSL 2.0**
  - Netscape, 1995, považována za nebezpečnou
- **SSL 3.0**
  - Netscape a Paul Kocher, 1996
  - podpora více algoritmů (např. Diffie-Hellman, Fortezza)
  - Lepší dohoda na certifikátu
- **TLS 1.0**
  - Internetový standard, vychází z SSL 3.0, RFC 2246, 1999
- **TLS 1.1**
  - Ochrana proti CBC útokům (zrušen implicitní IV)
- **TLS 1.2**
  - SHA256, AES

| Defined | |
|---|---|
| Protocol | Year |
| SSL 1.0 | n/a |
| SSL 2.0 | 1995 |
| SSL 3.0 | 1996 |
| TLS 1.0 | 1999 |
| TLS 1.1 | 2006 |
| TLS 1.2 | 2008 |
| TLS 1.3 | 2018 |

©Petr Hanáček CLACRYPT Slide


TLS 1.3
• Rychlejší navázání spojení
TLS 1.2 Handshake
TLS 1.3 Handshake
Client
Server
Client
Server
1
2
3
4
5
6
7
300ms
1
2
3
4
5
200ms
©Petr Hanáček
CLACRYPT Slide 87
https://kinsta.com/blog/tls-1-3/

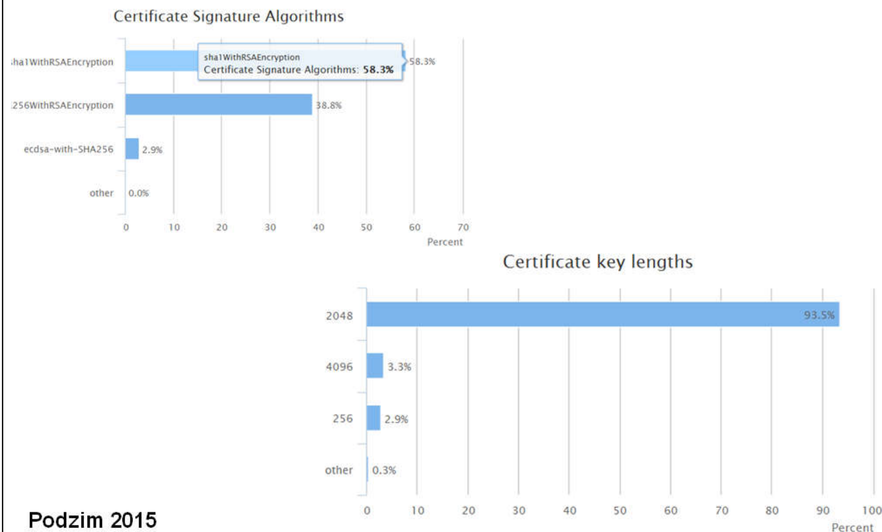

TLS 1.3
• Odstranění nebezpečných algoritmů a vlastností, např.:
– SHA-1
– RC4
– DES
– 3DES
– AES-CBC
– MD5
– EXPORT-strength ciphers – zodpovědné za FREAK a LogJam
©Petr Hanáček
CLACRYPT Slide 88
https://kinsta.com/blog/tls-1-3/

## Cipher Suite TLS

| *Cipher suite* | *Key Exchange* | *Encryption* | *Hash* |
|---|---|---|---|
| TLS_NULL_WITH_NULL_NULL | NULL | NULL | NULL |
| TLS_RSA_WITH_NULL_MD5 | RSA | NULL | MD5 |
| TLS_RSA_WITH_NULL_SHA | RSA | NULL | SHA-1 |
| TLS_RSA_WITH_RC4_128_MD5 | RSA | RC4 | MD5 |
| TLS_RSA_WITH_RC4_128_SHA | RSA | RC4 | SHA-1 |
| TLS_RSA_WITH_IDEA_CBC_SHA | RSA | IDEA | SHA-1 |
| TLS_RSA_WITH_DES_CBC_SHA | RSA | DES | SHA-1 |
| TLS_RSA_WITH_3DES_EDE_CBC_SHA | RSA | 3DES | SHA-1 |
| TLS_DH_anon_WITH_RC4_128_MD5 | DH_anon | RC4 | MD5 |
| TLS_DH_anon_WITH_DES_CBC_SHA | DH_anon | DES | SHA-1 |
| TLS_DH_anon_WITH_3DES_EDE_CBC_SHA | DH_anon | 3DES | SHA-1 |
| TLS_DHE_RSA_WITH_DES_CBC_SHA | DHE_RSA | DES | SHA-1 |
| TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA | DHE_RSA | 3DES | SHA-1 |
| TLS_DHE_DSS_WITH_DES_CBC_SHA | DHE_DSS | DES | SHA-1 |
| TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA | DHE_DSS | 3DES | SHA-1 |
| TLS_DH_RSA_WITH_DES_CBC_SHA | DH_RSA | DES | SHA-1 |
| TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA | DH_RSA | 3DES | SHA-1 |
| TLS_DH_DSS_WITH_DES_CBC_SHA | DH_DSS | DES | SHA-1 |
| TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA | DH_DSS | 3DES | SHA-1 |

©Petr Hanáček 

## Stav podzim 2015

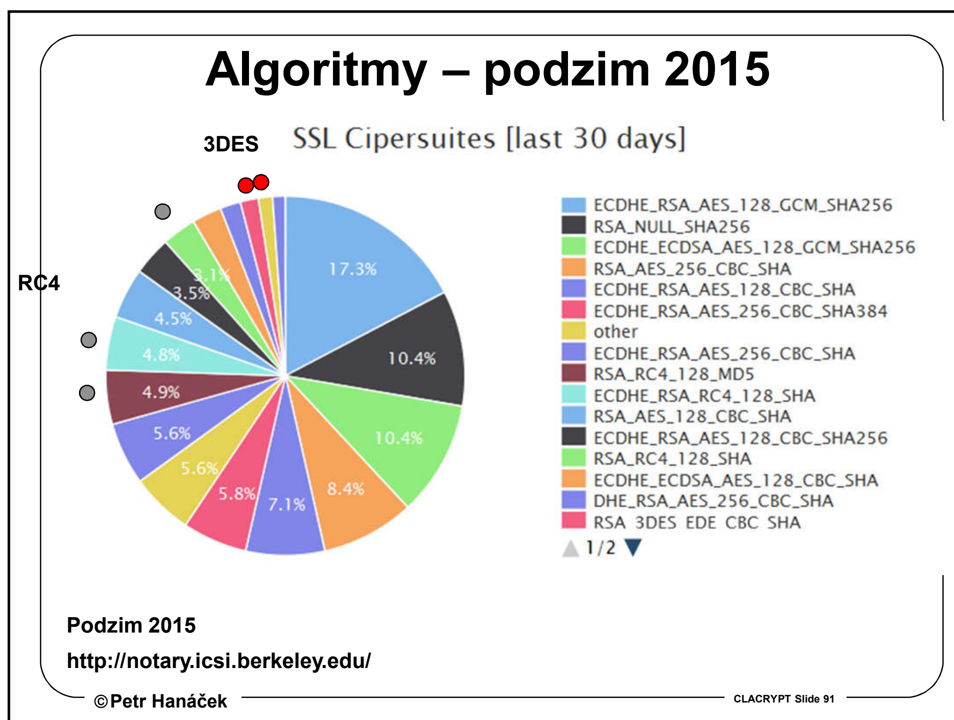


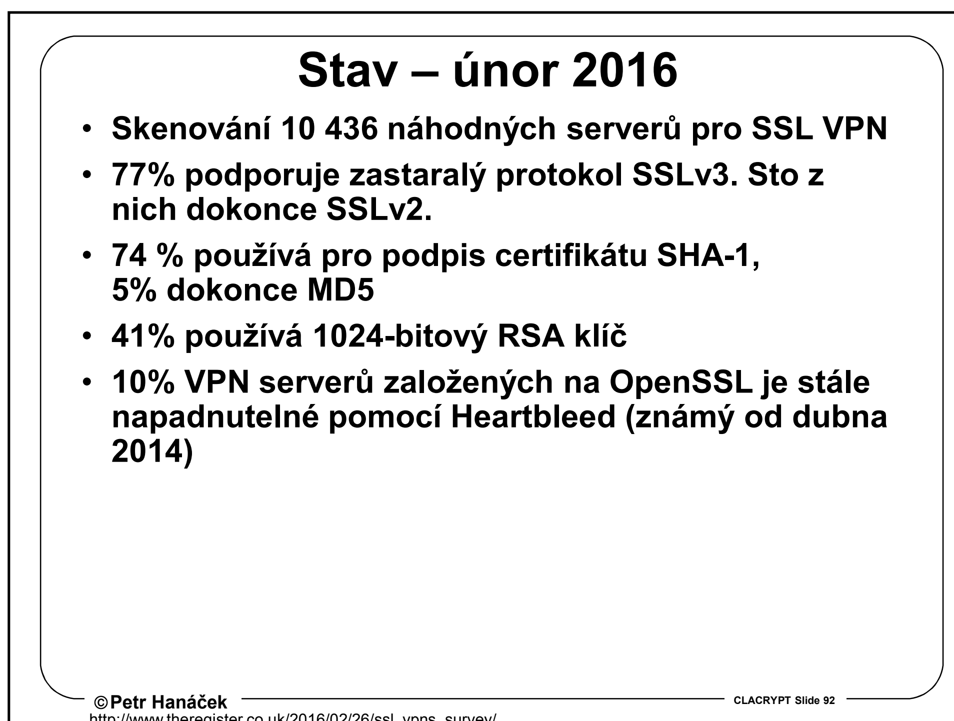

Podzim 2015

http://notary.icsi.berkeley.edu/

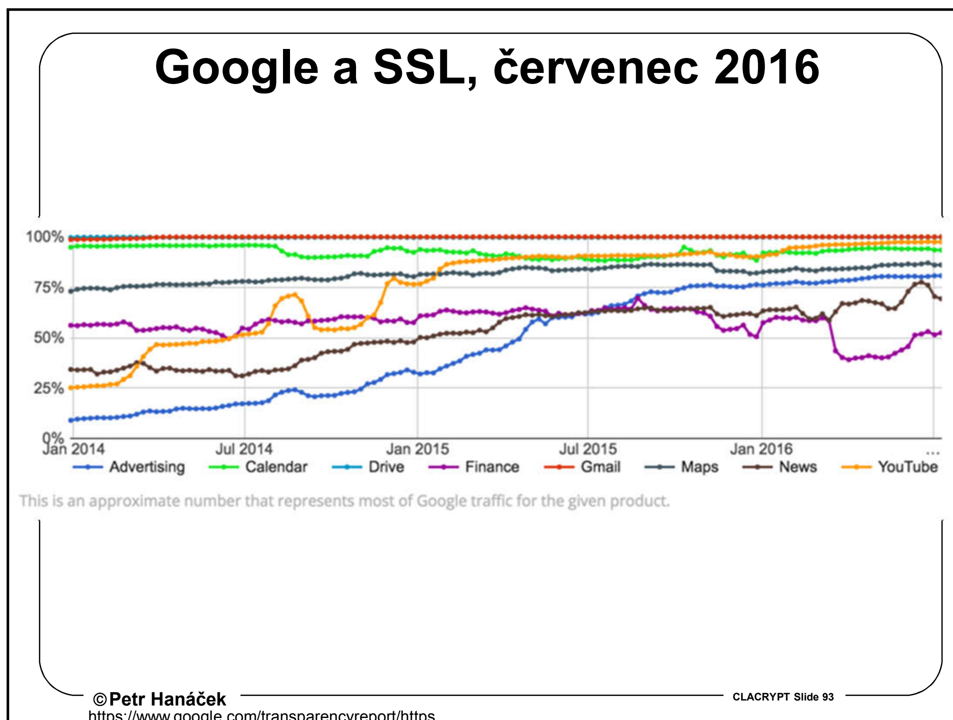
©Petr Hanáček 

© 


Algoritmy – podzim 2015
3DES
SSL Cipersuites [last 30 days]
RC4
17.3%
10.4%
10.4%
8.4%
7.1%
5.8%
5.6%
5.6%
4.9%
4.8%
4.5%
3.5%
3.1%
ECDHE_RSA_AES_128_GCM_SHA256
RSA_NULL_SHA256
ECDHE_ECDSA_AES_128_GCM_SHA256
RSA_AES_256_CBC_SHA
ECDHE_RSA_AES_128_CBC_SHA
ECDHE_RSA_AES_256_CBC_SHA384
other
ECDHE_RSA_AES_256_CBC_SHA
RSA_RC4_128_MD5
ECDHE_RSA_RC4_128_SHA
RSA_AES_128_CBC_SHA
ECDHE_RSA_AES_128_CBC_SHA256
RSA_RC4_128_SHA
ECDHE_ECDSA_AES_128_CBC_SHA
DHE_RSA_AES_256_CBC_SHA
RSA_3DES_EDE_CBC_SHA
1/2
Podzim 2015
http://notary.icsi.berkeley.edu/
©Petr Hanáček
CLACRYPT Slide 91


Stav – únor 2016
• Skenování 10 436 náhodných serverů pro SSL VPN
• 77% podporuje zastaralý protokol SSLv3. Sto z nich dokonce SSLv2.
• 74 % používá pro podpis certifikátu SHA-1, 5% dokonce MD5
• 41% používá 1024-bitový RSA klíč
• 10% VPN serverů založených na OpenSSL je stále napadnutelné pomocí Heartbleed (známý od dubna 2014)
©Petr Hanáček
CLACRYPT Slide 92
http://www.theregister.co.uk/2016/02/26/ssl_vpns_survey/

## Google a SSL, červenec 2016




©Petr Hanáček CLACRYPT Slide 93
https://www.google.com/transparencyreport/https

## TLS a útoky

- FREAK attack
  - Factoring RSA Export Keys
- Version rollback attacks
- BEAST attack
  - Browser Exploit Against SSL/TLS) - violate same origin policy
- CRIME and BREACH attacks
  - Compression Ratio Info-leak Made Easy
  - Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext
- Timing attacks on padding
- POODLE attack
  - Padding Oracle On Downgraded Legacy Encryption (SSL3 CBC)
- RC4 attacks
- Truncation attack
- Heartbleed Bug

©Petr Hanáček CLACRYPT Slide 94

# Další poznámky k bezpečnosti

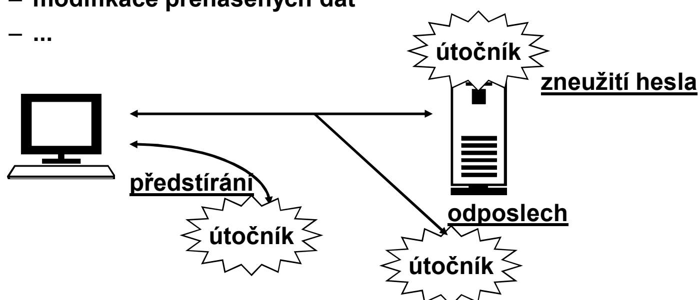
- **MITM attacks**
  - **Certificate pinning**
- **Forward secrecy**
  - **Ephemeral DH**
  - **Wiki:**
  - Dopředná bezpečnost (anglicky forward secrecy) je pojem z oboru kryptografie označující žádoucí vlastnost zabezpečených komunikačních protokolů, totiž že prozrazení soukromého klíče či hesla neohrozí dávnou komunikaci, respektive vyjádřeno naopak, že komunikace „dopředu“ v budoucnosti a její případná kompromitace nepředstavuje ohrožení pro současnou komunikaci. U protokolů, které postrádají dopřednou bezpečnost, si útočník při budoucím zjištění soukromého klíče může rozšifrovat všechnu starší komunikaci, k jejímuž šifrování byl klíč použit a kterou si útočník odposlechl a uložil.
  - Typickou metodou, jak zajistit dopřednou bezpečnost, je používat pro jednotlivé rozhovory vždy nové jednorázové klíče, které jsou po ukončení rozhovoru zahozeny a nelze je zpětně ze znalosti dlouhodobých klíčů odvodit. K dohodě na jednorázovém klíči lze použít nějakou variantu Diffieho-Hellmanovy výměny klíčů, přičemž zprávy dohadující jednorázový klíč jsou podepsány dlouhodobým klíčem k zabránění útoku člověka uprostřed.



# SSH

# SSH

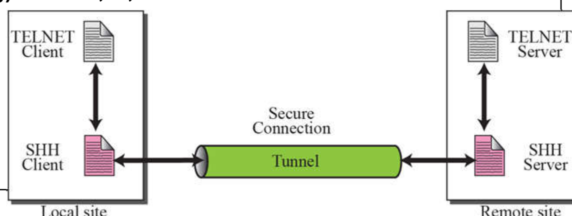
- **Motivace**
  - klasické služby vzdáleného terminálu (*telnet, rsh, rlogin*) nejsou bezpečné
- **autentizace heslem - hrozby**
  - odposlech hesla - heslo prochází sítí nezašifrované
  - odposlech přenášených dat
  - modifikace přenášených dat
  - ...






# SSH

- **SSH - Secure Shell**
  - bezpečný (šifrovaný) shell
  - nahrazuje *rsh, rlogin, rcp*
- **Služby**
  - Utajení: end-to-end encryption šifrování - DES, IDEA, Blowfish
  - Integrita: 32 bitový Cyclic Redundancy Check (CRC-32)
  - Autentizace: server pomocí „host key“, klient obvykle heslem nebo veřejným klíčem
  - Autorizace: buďto práva na celý server nebo podle uživatelských účtů na serveru
  - Přesměrování spojení (port forwarding), zapouzdření jiného protokolu (tunneling) - telnet, X, …



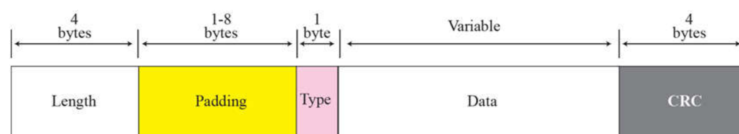
## Navázání spojení

- SSH server zašle klientovi – klíč serveru, seznam podporovaných šifrovací, autentizačních a kompresních algoritmů, a 8 náhodných bajtů
- Klient zkontroluje identitu serveru pomocí klíče serveru oproti databázi známých serverů
- Klient vygeneruje klíč relace (session key) a zašifruje jej klíčem serveru
- Zašle zašifrovaný klíč relace spolu s kontrolními bajty a přijatelnými algoritmy

©Petr Hanáček CLACRYPT Slide 99

## Autentizace

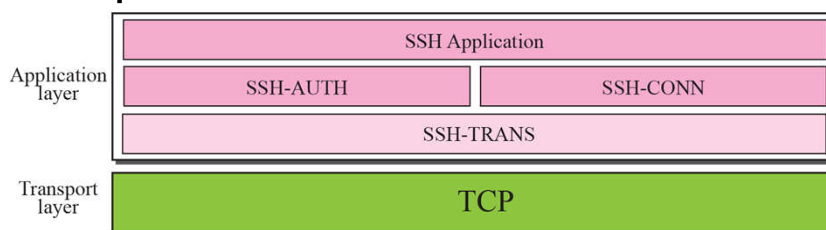
- Server dešifruje přijatý zašifrovaný klíč relace a pošle potvrzení, zašifrované klíčem relace
- Klient pošle potvrzení, že autentizoval server
- Klient se autentizuje obvykle buď heslem nebo veřejným klíčem
  - Password Authentication
  - Public key Authentication
    - » Server vygeneruje 256 bitovou náhodnou výzvu, zašifruje ji klientovým veřejným klíčem a pošle klientovi
    - » Klient dešifruje výzvu, a pošle serveru haš výzvy s identifikátorem relace a pošle serveru
    - » Server si vygeneruje stejný haš, porovná s přijatým, pokud sedí, relace je autentizovaná
- Server potvrdí, že autentizoval klienta




©Pet

## SSH2 vs. SSH1

- SSH2 má oddělený transportní, autentizační a připojovací protokol. SSH1 je monolitický.
- SSH2 kontroluje integritu pomocí MAC, SSH1 používá CRC-32
- SSH2 podporuje několik souběžných kanálů, SSH1 právě jeden
- SSH-2 server také obvykle podporuje SSH-1 kvůli kompatibilitě




©Petr Hanáček CLACRYPT Slide 101

http://www.root.cz/clanky/utoky-na-ssh-je-jeste-bezpecne/

©Petr Hanáček CLACRYPT Slide 102

KONEC

©Petr Hanáček


©

CHAPTER 16

# Obtaining Authentic Public Keys

## Chapter Goals

- To describe the notion of digital certificates.
- To explain the notion of a PKI.
- To examine different approaches such as X509, PGP and SPKI.
- To show how an implicit certificate scheme can operate.
- To explain how identity based cryptographic schemes operate.

### 1. Generalities on Digital Signatures

Digital signatures have a number of uses which go beyond the uses of handwritten signatures. For example we can use digital signatures to

- control access to data,
- allow users to authenticate themselves to a system,
- allow users to authenticate data,
- sign 'real' documents.

Each application has a different type of data being bound, a different length of the lifetime of the data to be signed, different types of principals performing the signing and verifying and different awareness of the data being bound.

For example an interbank payment need only contain the two account numbers and the amount. It needs to be signed by the payee and verified only by the computer which will carry out the transfer. The lifetime of the signature is only until the accounts are reconciled, for example when the account statements are sent to the customers and a suitable period has elapsed to allow the customers to complain of any error.

As another example consider a challenge response authentication mechanism. Here the user, to authenticate itself to the device, signs a challenge provided by the device. The lifetime of the signature may only be a few seconds. The user of course assumes that the challenge is random and is not a hash of an interbank payment. Hence, it is probably prudent that we use different keys for our authentication tokens and our banking applications.

As a final example consider a digital will or a mortgage contract. The length of time that this signature must remain valid may (hopefully in the case of a will) be many years. Hence, the security requirements for long-term legal documents will be very different from those of an authentication token.

You need to remember however that digital signatures are unlike handwritten signatures in that they are

- NOT necessarily on a document: Any piece of digital stuff can be signed.
- NOT transferable to other documents: Unlike a handwritten signature, a digital signature is different on each document.

- NOT modifiable after they are made: One cannot alter the document and still have the digital signature remaining valid.
- NOT produced by a person: A digital signature is never produced by a person, unless the signature scheme is very simple (and weak) or the person is a mathematical genius.

All they do is bind knowledge of an unrevealed private key to a particular piece of data.

## 2. Digital Certificates and PKI

When using a symmetric key system we assume we do not have to worry about which key belongs to which principle. It is tacitly assumed, see for example the chapter dealing with symmetric key agreement protocols and the BAN logic, that if Alice holds a long-term secret key $K_{ab}$ which she thinks is shared with Bob, then Bob really does have a copy of the same key. This assurance is often achieved using a trusted physical means of long-term key distribution, using for example armed couriers.

In a public key system the issues are different. Alice may have a public key which she thinks is associated with Bob, but we usually do not assume that Alice is 100 percent certain that it really belongs to Bob. This is because we do not, in the public key model, assume a physically secure key distribution system. After all, that was the point of public key cryptography in the first place: to make key management easier. Alice may have obtained the public key she thinks belongs to Bob from Bob's web page, but how does she know the web page has not been spoofed?

The process of linking a public key to an entity or principal, be it a person, machine or process, is called binding. One way of binding, common in many applications where the principal really does need to be present, is by using a physical token such as a smart card. Possession of the token, and knowledge of any PIN/password needed to unlock the token, is assumed to be equivalent to being the designated entity. This solution has a number of problems associated with it, since cards can be lost or stolen, which is why we protect them using a PIN (or in more important applications by using biometrics). The major problem is that most entities are non-human, they are computers and computers do not carry cards. In addition many public key protocols are performed over networks where physical presence of the principal (if it is human) is not something one can test.

Hence, some form of binding is needed which can be used in a variety of very different applications. The main binding tool in use today is the digital certificate. In this a special trusted third party, or TTP, called a certificate authority, or CA, is used to vouch for the validity of the public keys.

A CA based system works as follows:

- All users have a trusted copy of the public key of the CA. For example these come embedded in your browser when you buy your computer, and you 'of course' trust the vendor of the computer and the manufacturer of the software on your computer.
- The CA's job is to digitally sign data strings containing the following information

  (Alice, Alice's public key).

  This data string, and the associated signature is called a digital certificate. The CA will only sign this data if it truly believes that the public key really does belong to Alice.
- When Alice now sends you her public key, contained in a digital certificate, you now trust that the purported key really is that of Alice, since you trust the CA to do its job correctly.

This use of a digital certificate binds the name 'Alice' with the 'Key', it is therefore often called an identity certificate. Other bindings are possible, we shall see some of these later related to authorizations.

Public key certificates will typically (although not always) be stored in repositories and accessed as required. For example, most browsers keep a list of the certificates that they have come across. The digital certificates do not need to be stored securely since they cannot be tampered with as they are digitally signed.

To see the advantage of certificates and CAs in more detail consider the following example of a world without a CA. In the following discussion we break with our colour convention for a moment and now use red to signal public keys which must be obtained in an authentic manner and blue to signal public keys which do not need to be obtained in an authentic manner.

In a world without a CA you obtain many individual public keys from each individual in some authentic fashion. For example

| | |
|---|---|
| 6A5DEF....A21 | Jim Bean's public key, |
| 7F341A....BFF | Jane Doe's public key, |
| B5F34A....E6D | Microsoft's update key. |

Hence, each key needs to be obtained in an authentic manner, as does every new key you obtain.

Now consider the world with a CA. You obtain a single public key in an authentic manner, namely the CA's public key. We shall call our CA Ted since he is trustworthy. You then obtain many individual public keys, signed by the CA, in possibly an unauthentic manner. For example they could be attached at the bottom of an email, or picked up whilst browsing the web.

| | |
|---|---|
| A45EFB....C45 | Ted's totally trustworthy key, |
| 6A5DEF....A21 | Ted says 'This is Jim Bean's public key', |
| 7F341A....BFF | Ted says 'This is Jane Doe's public key', |
| B5F34A....E6D | Ted says 'This is Microsoft's update key'. |

If you trust Ted's key and you trust Ted to do his job correctly then you trust all the public keys you hold to be authentic.

In general a digital certificate is not just a signature on the single pair

(Alice, Alice's public key),

one can place all sorts of other, possibly application specific, information into the certificate. For example it is usual for the certificate to contain the following information.

- user's name,
- user's public key,
- is this an encryption or signing key?
- name of the CA,
- serial number of the certificate,
- expiry date of the certificate,
- ....

Commercial certificate authorities exist who will produce a digital certificate for your public key, often after payment of a fee and some checks on whether you are who you say you are. The certificates produced by commercial CAs are often made public, so one can call them public 'public key certificates', in that there use is mainly over open public networks.

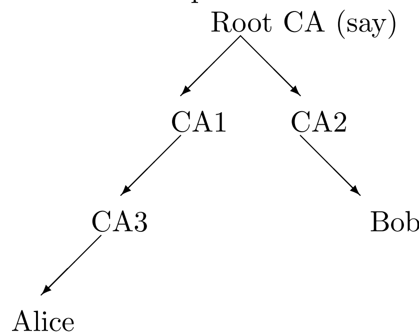
CAs are also used in proprietary systems, for example in debit/credit card systems or by large corporations. In such situations it may be the case that the end users do not want their public key certificates to be made public, in which case one can call them private 'public key certificates'. But one should bear in mind that whether the digital certificate is public or private should not effect the security of the private key associated to the public key contained in the certificate. The decision to make one's certificates private is often one of business rather than security.

It is common for more than one CA to exist. A quick examination of the properties of your web browser will reveal a large number of certificate authorities which your browser assumes you 'trust' to perform the function of a CA. As there are more than one CA it is common for one CA to sign a digital certificate containing the public key of another CA, and vice versa, a process which is known as cross-certification.

Cross-certification is needed if more than one CA exists, since a user may not have a trusted copy of the CA's public key needed to verify another user's digital certificate. This is solved by cross-certificates, i.e. one CA's public key is signed by another CA. The user first verifies the appropriate cross-certificate, and then verifies the user certificate itself.

With many CAs one can get quite long certificate chains, as Fig. 1 illustrates. Suppose Alice trusts the Root CA's public key and she obtains Bob's public key which is signed by the private key of CA2. She then obtains CA2's public key, either along with Bob's digital certificate or by some other means. CA2's public key comes in a certificate which is signed by the private key of the Root CA. Hence, by verifying all the signatures she ends up trusting Bob's public key.

FIGURE 1. Example certification hierarchy




Often the function of a CA is split into two parts. One part deals with verifying the user's identity and one part actually signs the public keys. The signing is performed by the CA, whilst the identity of the user is parcelled out to a registration authority, or RA. This can be a good practice, with the CA implemented in a more secure environment to protect the long-term private key.

The main problem with a CA system arises when a user's public key is compromised or becomes untrusted for some reason. For example

- a third party has gained knowledge of the private key,
- an employee leaves the company.

As the public key is no longer to be trusted all the associated digital certificates are now invalid and need to be revoked. But these certificates can be distributed over a large number of users, each one of which needs to be told to no longer trust this certificate. The CA must somehow inform all users that the certificate(s) containing this public key is/are no longer valid, in a process called certificate revocation.

One way to accomplish this is via a Certificate Revocation List, or CRL, which is a signed statement by the CA containing the serial numbers of all certificates which have been revoked by that CA and whose validity period has not expired. One clearly need not include in this the serial numbers of certificates which have passed their expiry date. Users must then ensure they have the latest CRL. This can be achieved by issuing CRLs at regular intervals even if the list has not changed. Such a system can work well in a corporate environment when overnight background jobs are often used to make sure each desktop computer in the company is up to date with the latest software etc. For other situations it is hard to see how the CRLs can be distributed, especially if there are a large number of CAs trusted by each user.

In summary, with secret key cryptography the main problems were ones of

- key management,
- key distribution.

These problems resulted in keys needing to be distributed via secure channels. In public key systems we replace these problems with those of

- key authentication,

in other words which key belongs to who. Hence, keys needed to be distributed via authentic channels. The use of digital certificates provides the authentic channels needed to distribute the public keys.

The whole system of CAs and certificates is often called the Public Key Infrastructure or PKI. This essentially allows a distribution of trust; the need to trust the authenticity of each individual public key in your possession is replaced by the need to trust a body, the CA, to do its job correctly.

In ensuring the CA does its job correctly you can either depend on the legal system, with maybe a state sponsored CA, or you can trust the business system in that it would not be in the CA's business interests to not act properly. For example, if it did sign a key in your name by mistake then you could apply publicly for exemplary restitution.

We end this section by noting that we have now completely solved the key distribution problem: For two users to agree on a shared secret key, they first obtain authentic public keys from a CA. Then secure session keys are obtained using, for example, signed Diffie–Hellman,

$$\begin{array}{ccc} \text{Alice} & & \text{Bob} \\ (g^a, \text{Sign}_{\text{Alice}}(g^a)) & \longrightarrow & \\ & \longleftarrow & (g^b, \text{Sign}_{\text{Bob}}(g^b)) \end{array}$$

## 3. Example Applications of PKI

In this section we shall look at some real systems which distribute trust via digital certificates. The examples will be

- PGP,
- SSL,
- X509 (or PKIX),
- SPKI.

**3.1. PGP.** The email encryption program Pretty Good Privacy, or PGP, takes a bottom-up approach to the distribution of trust. The design goals of PGP were to give a low-cost encryption/signature system for all users, hence the use of an expensive top-down global PKI would not fit this model. Instead the system makes use of what it calls a 'Web of Trust'.

The public key management is done from the bottom up by the users themselves. Each user acts as their own CA and signs other user's public keys. So Alice can sign Bob's public key and then Bob can give this signed 'certificate' to Charlie, in which case Alice is acting as a CA for Bob. If Charlie trusts Alice's judgement with respect to signing people's keys then she will trust that Bob's key really does belong to Bob. It is really up to Charlie to make this decision. As users keep doing this cross-certification of each other's keys, a web of trusted keys grows from the bottom up.

PGP itself as a program uses RSA public key encryption for low-volume data such as session keys. The block cipher used for bulk transmission is called IDEA. This block cipher has a 64-bit block size and a 128-bit key size and is used in CFB mode. Digital signatures in PGP can be produced either with the RSA or the DSA algorithm, after a message digest is taken using either MD5 or SHA-1.

The keys that an individual trusts are held in a so-called key ring. This means users have control over their own local public key store. This does not rule out a centralized public key store, but means one is not necessarily needed.

Key revocation is still a problem with PGP as with all such systems. The ad-hoc method adopted by PGP is that if your key is compromised then you should tell all your friends, who have a copy of your key, to delete the key from their key ring. All your friends should then tell all their friends and so on.

**3.2. Secure Socket Layer.** Whilst the design of PGP was driven by altruistic ideals, namely to provide encryption for the masses, the Secure Socket Layer, or SSL, was driven by commercial requirements, namely to provide a secure means for web based shopping and sales. Essentially SSL adds security to the TCP level of the IP stack. It provides security of data and not parties but allows various protocols to be transparently layered on top, for example HTTP, FTP, TELNET, etc.

The primary objective was to provide channel security, to enable the encrypted transmission of credit card details or passwords. After an initial handshake all subsequent traffic is encrypted. The server side of the communication, namely the website or the host computer in a Telnet session, is always authenticated for the benefit of the client. Optionally the client may be authenticated to the user, but this is rarely done in practice for web based transactions.

As in PGP, bulk encryption is performed using a block or stream cipher (usually either DES or an algorithm from the RC family). The choice of precise cipher is chosen between the client and server during the initial handshake. The session key to be used is derived using standard protocols such as the Diffie–Hellman protocol, or RSA based key transport.

The server is authenticated since it provides the client with an X509 public key certificate. This, for web shopping transactions, is signed by some global CA whose public key comes embedded into the user's web browser. For secure Telnet sessions (often named SSH after the program which runs them) the server side certificate is usually a self-signed certificate from the host computer.

The following is a simplified overview of how SSL can operate.

- The client establishes connection with the server on a special port number so as to signal this will be a secure session.
- The server sends a certified public key to the client.
- The client verifies the certificate and decides whether it trusts this public key.
- The client chooses a random secret.
- The client encodes this with the server's public key and sends this back to the server.
- The client and server now securely share the secret.
- The server now authenticates itself to the client by responding using the shared secret.

The determination of session keys can be a costly operation for both the server and the client, especially when the data may come in bursts, as when one is engaged in shopping transactions, or performing some remote access to a computer. Hence, there is some optimization made to enable reuse of session keys. The client is allowed to quote a previous session key, the server can either accept it or ask for a new one to be created. So as to avoid any problems this ability is limited by two rules. Firstly a session key should have a very limited lifetime and secondly any fatal error in any part of the protocols will immediately invalidate the session key and require a new one to be determined. In SSL the initial handshake is also used for the client and the server to agree on which bulk encryption algorithm to use, this is usually chosen from the list of RC4, RC5, DES or Triple DES.

**3.3. X509 Certificates.** When discussing SSL we mentioned that the server uses an X509 public key certificate. X509 is a standard which defines a structure for public key certificates, currently it is the most widely deployed certificate standard. A CA assigns a unique name to each

user and issues a signed certificate. The name is often the URL or email address. This can cause problems since, for example, many users may have different versions of the same email address. If you send a signed email containing your certificate for your email 'address'

N.P.Smart@some.where.com

but your email program sends this from the 'address'

Nigel.Smart@some.where.com

then, even though you consider both addresses to be equivalent, the email client of the recipient will often complain saying that the signature is not to be trusted.

The CAs are connected in a tree structure, with each CA issuing a digital certificate for the one beneath it. In addition cross-certification between the branches is allowed. The X509 certificates themselves are defined in standards using a language called ASN.1, or Abstract Syntax Notation. This can be rather complicated at first sight and the processing of all the possible options often ends up with incredible 'code bloat'.

The basic X509 certificate structure is very simple, but can end up being very complex in any reasonable application. This is because some advanced applications may want to add additional information into the certificates which enable authorization and other capabilities. However, the following records are always in a certificate.

- The version number of the X509 standard which this certificate conforms to.
- The certificate serial number.
- The CA's signing algorithm identifier. This should specify the algorithm and the domain parameters used, if the CA has multiple possible algorithms or domain parameters.
- The issuer's name, i.e. the name of the issuing CA.
- The validity period in the form of a not-before and not-after date.
- The subject's name, i.e. whose public key is being signed. This could be an email address or a domain name.
- The subject's public key. This contains the algorithm name and any associated domain parameters plus the actual value of the public key
- The issuer's signature on the subject's public key and all data that is to be bound to the subject's public key, such as the subject's name.

**3.4. SPKI.** In response to some of the problems associated with X509, another type of certificate format has been proposed called SPKI, or Simple Public Key Infrastructure. This system aims to bind authorizations as well as identities, and also tries to deal with the issue of delegation of authorizations and trust. Thus it may be suitable for business to business e-commerce transactions. For example, when managers go on holiday they can delegate their authorizations for certain tasks to their subordinates.

SPKI does not assume the global CA hierarchy which X509 does. It assumes a more ground-up approach like PGP. However, it is currently not used much commercially since PKI vendors have a lot of investment in X509 and are probably not willing to switch over to a new system (and the desktop applications such as your web browser would also need significant alterations).

Instead of using ASN.1 to describe certificates, SPKI uses S-expressions. These are LISP-like structures which are very simple to use and describe. In addition S-expressions can be made very simple even for humans to understand, as opposed to the machine-only readable formats of X509 certificates. S-expressions can even come with display hints to enable greater readability. The current draft standard specifies these display hints as simple MIME-types.

Each SPKI certificate has an issuer and a subject both of which are public keys (or a hash of a public key), and not names. This is because SPKI's authors claim that it is a key which does something and not a name. After all it is a key which is used to sign a document etc. Focusing on the keys also means we can concentrate more on the functionality. There are two types of

SPKI certificates: ones for binding identities to keys and ones for binding authorizations to keys. Internally these are represented as tuples of 4 and 5 objects, which we shall now explain.

3.4.1. *SPKI 4-Tuples.* To give an identity certificate and bind a name with a key, like X509 does, SPKI uses a 4-tuple structure. This is an internal abstraction of what the certificate represents and is given by:

(Issuer, Name, Subject, Validity).

In real life this would consist of the following five fields:

- issuer's public key,
- name of the subject,
- subject's public key,
- validity period,
- signature of the issuer on the triple (Name, Subject, Validity).

Anyone is able to issue such a certificate, and hence become a CA.

3.4.2. *SPKI 5-Tuples.* 5-tuples are used to bind keys to authorizations. Again this is an internal abstraction of what the certificate represents and is given by

(Issuer, Subject, Delegation, Authorization, Validity).

In real life this would consist of the following six fields:

- issuer's public key.
- subject's public key.
- delegation. A 'Yes' or 'No' flag, saying whether the subject can delegate the permission or not.
- authorization. What the subject is being given permission to do
- validity. How long the authorization is for.
- signature of the issuer on the quadruple (S,D,A,V).

One can combine an authorization certificate and an identity certificate to obtain an audit trail. This is needed since the authorization certificate only allows a key to perform an action. It does not say who owns the key. To find out who owns a key you need to use an identity certificate.

When certificate chains are eventually checked to enable some authorization, a 5-tuple reduction procedure is carried out. This can be represented by the following rule

$$\begin{aligned}(I_1, S_1, D_1, A_1, V_1) &+ (I_2, S_2, D_2, A_2, V_2) \\ &= (I_1, S_2, D_2, A_1 \cap A_2, V_1 \cap V_2).\end{aligned}$$

This equality holds only if

- $S_1 = I_2$
- $D_1 = \mathbf{true}$.

This means the first two certificates together can be interpreted as the third. This third 5-tuple is not really a certificate, it is the meaning of the first two when they are presented together.

As an example we will show how combining two 5-tuples is equivalent to delegating authority. Suppose our first 5-tuple is given by:

- $I_1 =$ Alice
- $S_1 =$ Bob
- $D_1 = \mathbf{true}$
- $A_1 =$ Spend up to £100 on Alice's account
- $V_1 =$ forever.

So Alice allows Bob to spend up to £100 on her account and allows Bob to delegate this authority to anyone he chooses.

Now consider the second 5-tuple given by

- $I_2$ = Bob
- $S_2$ = Charlie
- $D_2$ = **false**
- $A_2$ = Spend between £50 and £200 on Alice's account
- $V_2$ = before tomorrow morning.

So Bob is saying Charlie can spend between £50 and £200 of Alice's money, as long as it happens before tomorrow morning.

We combine these two 5-tuples, using the 5-tuple reduction rule, to form the new 5-tuple

- $I_3$ = Alice
- $S_3$ = Charlie
- $D_3$ = **false**
- $A_3$ = Spend between £50 and £100 on Alice's account
- $V_3$ = before tomorrow morning.

Since Alice has allowed Bob to delegate she has in effect allowed Charlie to spend between £50 and £100 on her account before tomorrow morning.

## 4. Other Applications of Trusted Third Parties

In some applications it is necessary for signatures to remain valid for a long time. Revocation of a public key, even long after the legitimate creation of the signature, potentially invalidates all digital signatures made using that key, even those in the past. This is a major problem if digital signatures are to be used for documents of long-term value such as wills, life insurance and mortgage contracts. We essentially need methods to prove that a digital signature was made prior to the revocation of the key and not after it. This brings us to the concept of time stamping.

A time stamping service is a means whereby a trusted entity will take a signed message, add a date/timestamp and sign the result using its own private key. This proves when a signature was made (like a notary service in standard life insurance). However, there is the requirement that the public key of the time stamping service must never be revoked. An alternative to the use of a time stamping service is the use of a secure archive for signed messages.

As another application of a trusted third-party consider the problem associated with keeping truly secret keys for encryption purposes.

- What if someone loses or forgets a key? They could lose all your encrypted data.
- What if the holder of the key resigns from the company or is killed? The company may now want access to the encrypted data.
- What if the user is a criminal? Here the government may want access to the encrypted data.

One solution is to deposit a copy of your key with someone else in case you lose yours, or something untoward happens. On the other hand, simply divulging the key to anybody, even the government, is very insecure.

A proposed solution is key escrow implemented via a secret sharing scheme. Here the private key is broken into pieces, each of which can be verified to be correct. Each piece is then given to some authority. At some later point if the key needs to be recovered then a subset of the authorities can come together and reconstruct it from their shares. The authorities implementing this escrow facility are another example of a Trusted Third Party, since you really have to trust them. In fact the trust required is so high that this solution has been a source of major debate within the cryptographic and governmental communities in the past. The splitting of the key between the various escrow authorities can be accomplished using the trick of a secret sharing scheme which we will discuss in Chapter 23.

## 5. Implicit Certificates

One issue with digital certificates is that they can be rather large. Each certificate needs to at least contain both the public key of the user and the signature of the certificate authority on that key. This can lead to quite large certificate sizes, as the following table demonstrates:

| | RSA | DSA | EC-DSA |
|---|---|---|---|
| User's key | 1024 | 1024 | 160 |
| CA sig | 1024 | 320 | 320 |

This assumes for RSA keys one uses a 1024-bit modulus, for DSA one uses a 1024-bit prime $p$ and a 160-bit prime $q$ and for EC-DSA one uses a 160-bit curve. Hence, for example, if the CA is using 1024-bit RSA and they are signing the public key of a user using 1024-bit DSA then the total certificate size must be at least 2048 bits. An interesting question is whether this can be made smaller.

Implicit certificates enable this. An implicit certificate looks like

$$X|Y$$

where

- $X$ is the data being bound to the public key,
- $Y$ is the implicit certificate on $X$.

From $Y$ we need to be able to recover the public key being bound to $X$ and implicit assurance that the certificate was issued by the CA. In the system we describe below, based on a DSA or EC-DSA, the size of $Y$ will be 1024 or 160 bits respectively. Hence, the size of the certificate is reduced to the size of the public key being certified.

**5.1. System Setup.** The CA chooses a public group $G$ of known order $n$ and an element $P \in G$. The CA then chooses a long-term private key $c$ and computes the public key

$$Q = P^c.$$

This public key should be known to all users.

**5.2. Certificate Request.** Suppose Alice wishes to request a certificate and the public key associated to the information $ID$, which could be her name. Alice computes an ephemeral secret key $t$ and an ephemeral public key

$$R = P^t.$$

Alice sends $R$ and $ID$ to the CA.

**5.3. Processing of the request.** The CA checks that he wants to link $ID$ with Alice. The CA picks another random number $k$ and computes

$$g = P^k R = P^k P^t = P^{k+t}.$$

Then the CA computes

$$s = cH(ID \| g) + k \pmod{n}.$$

Then the CA sends back to Alice the pair

$$(g, s).$$

The implicit certificate is the pair

$$(ID, g).$$

We now have to convince you that

- Alice can recover a valid public/private key pair,
- any other user can recover Alice's public key from this implicit certificate

**5.4. Alice's Key Discovery.** Alice knows the following information

$$t, s, R = P^t.$$

From this she can recover her private key

$$a = t + s \pmod{n}.$$

Note, Alice's private key is known only to Alice and not to the CA. In addition Alice has contributed some randomness $t$ to her private key, as has the CA who contributed $k$. Her public key is then

$$P^a = P^{t+s} = P^t P^s = R \cdot P^s.$$

**5.5. User's Key Discovery.** Since $s$ and $R$ are public, a user, say Bob, can recover Alice's public key from the above message flows via

$$R \cdot P^s.$$

But this says nothing about the linkage between the CA, Alice's public key and the $ID$ information. Instead Bob recovers the public key from the implicit certificate

$$(ID, g)$$

and the CA's public key

$$Q$$

via the equation

$$P^a = Q^{H(ID\|g)} g.$$

As soon as Bob sees Alice's key used in action, say he verifies a signature purported to have been made by Alice, he knows implicitly that it must have been issued by the CA, since otherwise Alice's signature would not verify correctly.

There are a number of problems with the above system which mean that implicit certificates are not used much in real life. For example,

(1) What do you do if the CA's key is compromised? Usually you pick a new CA key and re-certify the user's keys. But you cannot do this since the user's public key is chosen interactively during the certification process.
(2) Implicit certificates require the CA and users to work at the same security level. This is not considered good practice, as usually one expects the CA to work at a higher security level (say 2048-bit DSA) than the user (say 1024-bit DSA).

However for devices with restricted bandwidth they can offer a suitable alternative where traditional certificates are not available.

## 6. Identity Based Cryptography

Another way of providing authentic public keys, without the need for certificates, is to use a system whereby the user's key is given by their identity. Such a system is called an identity based encryption scheme or an identity based signature scheme. Such systems do not remove the need for a trusted third-party to perform the original authentication of the user, but they do however remove the need for storage and transmission of certificates.

The first scheme of this type was a signature scheme invented by Shamir in 1984. It was not until 2001 however that an identity based encryption scheme was given by Boneh and Franklin. We shall only describe the original identity based signature scheme of Shamir, which is based on the RSA problem.

A trusted third-party first calculates an RSA modulus $N$, keeping the two factors secret. The TTP also publishes a public exponent $e$, keeping the corresponding private exponent $d$ to themselves. In addition there is decided a mapping

$$I : \{0,1\}^* \longrightarrow (\mathbb{Z}/N\mathbb{Z})^*$$

which takes bit strings to elements of $(\mathbb{Z}/N\mathbb{Z})^*$. Such a mapping could be implemented by a hash function.

Now suppose Alice wishes to obtain the private key $g$ corresponding to her name 'Alice'. This is calculated for her by the TTP using the equation

$$g = I(\text{Alice})^d \pmod{N}.$$

To sign a message $m$, Alice generates the pair $(t, s)$ via the equations

$$t = r^e \pmod{N},$$
$$s = g \cdot r^{H(m\|t)} \pmod{N},$$

where $r$ is a random integer and $H$ is a hash function.

To verify the signature $(t, s)$ on the message $m$ another user can do this, simply by knowing the TTP's public data and the identity of Alice, by checking that the following equation holds modulo $N$,

$$\begin{aligned} I(\text{Alice}) \cdot t^{H(m\|t)} &= g^e \cdot r^{e \cdot H(m\|t)} \\ &= \left(g \cdot r^{H(m\|t)}\right)^e \\ &= s^e. \end{aligned}$$

## Chapter Summary

- Digital certificates allow us to bind a public key to some other information, such as an identity.
- This binding of key with identity allows us to solve the problem of how to distribute authentic public keys.
- Various PKI systems have been proposed, all of which have problems and benefits associated with them.
- PGP and SPKI work from the bottom up, whilst X509 works in a top-down manner.
- SPKI contains the ability to delegate authorizations from one key to another.
- Other types of trusted third-party applications exist such as time stamping and key escrow.
- Implicit certificates aim to reduce the bandwidth requirements of standard certificates, however they come with a number of drawbacks.
- Identity based cryptography helps authenticate a user's public key by using their identity as the public key, but it does not remove the need for trusted third parties.

## Further Reading

A good overview of the issues related to PKI can be found in the book by Adams and Lloyd. For further information on PGP and SSL look at the books by Garfinkel and Rescorla.

C. Adams and S. Lloyd. *Understanding Public-Key Infrastructure: Concepts, Standards and Deployment Considerations.* New Riders Publishing, 1999.

S. Garfinkel. *PGP: Pretty Good Privacy.* O'Reilly & Associates, 1994.

E. Rescorla. *SSL and TLS: Design and Building Secure Systems.* Addison-Wesley, 2000.

# 10

# Electronic Mail Security: PGP, S/MIME

Pretty Good Privacy (PGP) was invented by Philip Zimmermann who released version 1.0 in 1991. Subsequent versions 2.6.x and 5.x (or 3.0) of PGP have been implemented by an all-volunteer collaboration under the design guidance of Zimmermann. PGP is widely used in the individual and commercial versions that run on a variety of platforms throughout the computer community. PGP uses a combination of symmetric secret-key and asymmetric public-key encryption to provide security services for electronic mail and data files. It also provides data integrity services for messages and data files by using digital signature, encryption, compression (zip), and radix-64 conversion (ASCII Armor). With the explosively growing reliance on e-mail and file storage, authentication and confidentiality services have become increasing demands.

MIME is an extension to the RFC 2822 framework which defines a format for text messages being sent using e-mail. MIME is actually intended to address some of the problems and limitations of the use of SMTP. Secure/Multipurpose Internet Mail Extension (S/MIME) is a security enhancement to the MIME Internet e-mail format standard, based on technology from RSA Data Security.

Although both PGP and S/MIME are on an IETF standards track, it appears likely that PGP will remain the choice for personnel e-mail security for many users, while S/MIME will emerge as the industry standard for commercial and organizational use. Two schemes of PGP and S/MIME are discussed in this chapter.

## 10.1 PGP

Before looking at the operation of PGP in detail, it is convenient to confirm the notation. In the forthcoming analyses for security and data integrity services, the following symbols are generally used:

*Wireless Mobile Internet Security*, Second Edition. Man Young Rhee.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

$K_s$ = session key
$KP_a$ = public key of user A
$KS_a$ = private key of user A
$E$ = conventional encryption
$E_p$ = public-key encryption
$Z$ = compression using zip algorithm
$\|$ = concatenation
$H$ = hash function
$KP_b$ = public key of user B
$KS_b$ = private key of user B
$D$ = conventional decryption
$D_p$ = public-key decryption
$Z^{-1}$ = decompression

### 10.1.1 Confidentiality via Encryption

PGP provides confidentiality by encrypting messages to be transmitted or data files to be stored locally using a conventional encryption algorithm such as IDEA, 3DES, or CAST-128. In PGP, each symmetric key, known as a *session key*, is used only once. A new session key is generated as a random 128-bit number for each message. Since it is used only once, the session key is bound to the message and transmitted with it. To protect the key, it is encrypted with the receiver's public key. Figure 10.1 illustrates the sequence, which is described as follows:

- The sender creates a message.
- The sending PGP generates a random 128-bit number to be used as a session key for this message only.
- The session key is encrypted with RSA, using the recipient's public key.
- The sending PGP encrypts the message, using CAST-128 or IDEA or 3DES, with the session key. Note that the message is also usually compressed.
- The receiving PGP uses RSA with its private key to decrypt and recover the session key.
- The receiving PGP decrypts the message using the session key. If the message was compressed, it will be decompressed.

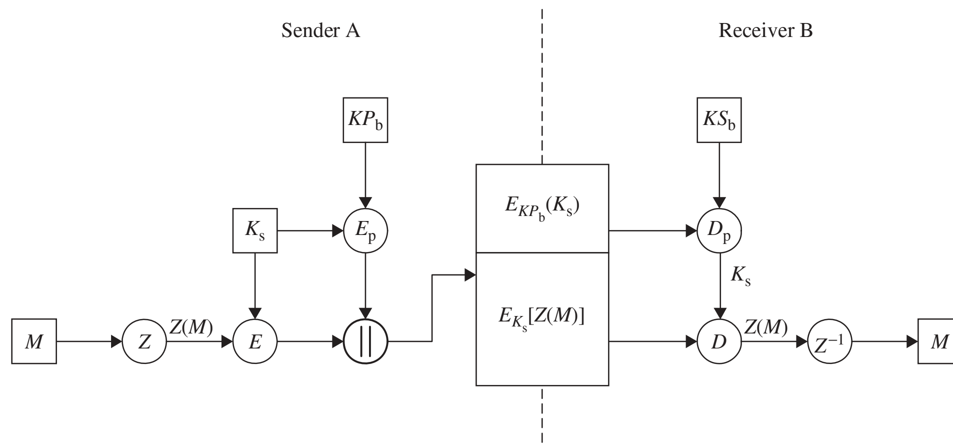



**Figure 10.1** PGP confidentiality computation scheme with compression/decompression Algorithms.

Instead of using RSA for key encryption, PGP may use a variant of Diffie–Hellman (known as *ElGamal*) that does provide encryption/decryption. In order for the encryption time to reduce, the combination of conventional and public-key encryption is used in preference to simply using RSA or ElGamal to encrypt the message directly. In fact, CAST-128 and other conventional algorithms are substantially faster than RSA or ElGamal. Since the recipient is able to recover the session key that is bound to the message, the use of the public-key algorithms solves the session key exchange problem. Finally, to the extent that the entire scheme is secure, PGP should provide the user with a range of key size options from 768 to 3072 bits.

Both digital signature and confidentiality services may be applied to the same message. First, a signature is generated from the message and attached to the message. Then the message plus signature are encrypted using a symmetric session key. Finally, the session key is encrypted using public-key encryption and prefixed to the encrypted block.

### 10.1.2 Authentication via Digital Signature

The digital signature uses a hash code of the message digest algorithm and a public-key signature algorithm. Figure 10.2 illustrates the digital signature service provided by PGP. The sequence is as follows:

- The sender creates a message.
- SHA-1 is used to generate a 160-bit hash code of the message.
- The hash code is encrypted with RSA using the sender's private key and a digital signature is produced.
- The binary signature is attached to the message.
- The receiver uses RSA with the sender's public key to decrypt and recover the hash code.
- The receiver generates a new hash code for the received message and compares it with the decrypted hash code. If the two match, the message is accepted as authentic.

The combination of SHA-1 and RSA provides an effective digital signature scheme. As an alternative, signatures can be generated using DSS/SHA-1. The National Institute

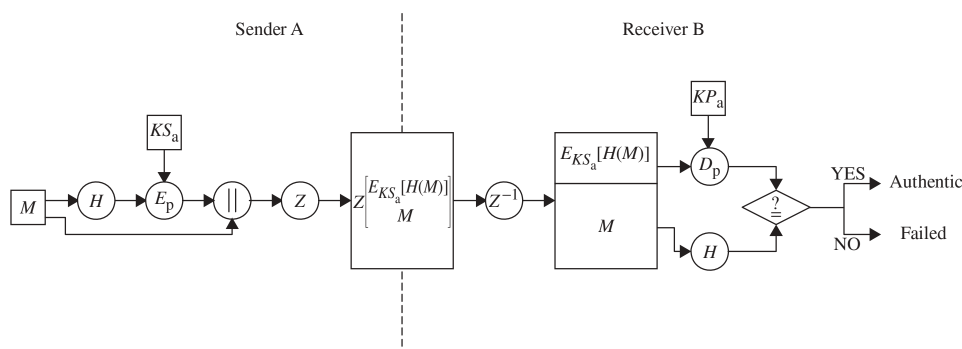



**Figure 10.2** PGP authentication computation scheme using compression algorithm.

of Standards and Technology (NIST) has published FIPS PUB 186, known as the *Digital Signature Standard* (DSS). The DSS uses an algorithm that is designed to provide only the digital signature function. Although DSS is a public-key technique, it cannot be used for encryption or key exchange. The DSS approach for generating digital signatures was fully discussed in Chapter 5. The DSS makes use of the secure hash algorithm (SHA-1) described in Chapter 4 and presents a new digital signature algorithm (DSA).

### 10.1.3 Compression

As a default, PGP compresses the message after applying the signature but before encryption. The placement of $Z$ for compression and $Z^{-1}$ for decompression is shown in Figures 10.1 and 10.2. This compression algorithm has the benefit of saving space both for e-mail transmission and for file storage. However, PGP compression technique will present a difficulty.

Referring to Figure 10.1, message encryption is applied after compression to strengthen cryptographic security. In reality, cryptanalysis will be more difficult because the compressed message has less redundancy than the original message.

Referring to Figure 10.2, signing an uncompressed original message is preferable because the uncompressed message together with the signature are directly used for future verification. On the other hand, for a compressed message, one may consider two cases, either to store a compressed message for later verification or to recompress the message when verification is required. Even if a recompressed message were recovered, PGP's compression algorithm would present a difficulty due to the fact that different trade-offs in running speed versus compression ratio produce different compressed forms.

PGP makes use of a compression package called *ZIP* which is functionally equivalent to PKZIP developed by PKWARE, Inc. The zip algorithm is perhaps the most commonly used cross-platform compression technique.

Two main compression schemes, named after Abraham Lempel and Jakob Ziv, were first proposed by them in 1977 and 1978, respectively. These two schemes for text compression (generally referred to as *lossless compression*) are broadly used because they are easy to implement and also fast.

In 1982 James Storer and Thomas Szymanski presented their scheme, LZSS, based on the work of Lempel and Ziv. In LZSS, the compressor maintains a window of size $N$ bytes and a lookahead buffer. Sliding-window-based schemes can be simplified by numbering the input text characters mod $N$, in effect creating a circular buffer. Variants of sliding-window schemes can be applied for additional compression to the output of the LZSS compressor, which include a simple variable-length code (LZB), dynamic Huffman coding (LZH), and Shannon–Fano coding (ZIP 1.x). All of them result in a certain degree of improvement over the basic scheme, especially when the data is rather random and the LZSS compressor has little effect.

Recently an algorithm was developed which combines the idea behind LZ77 and LZ78 to produce a hybrid called *LZFG* . LZFG uses the standard sliding window, but stores the data in a modified tree data structure and produces as output the position of the text in the tree. Since LZFG only inserts complete phrases into the dictionary, it should run faster than other LZ77-based compressors.

Huffman compression is a statistical data compression technique which reduces the average code length used to represent the symbols of an alphabet. Huffman code is an example of a code which is optimal when all symbols probabilities are integral powers of 1/2. A technique related to Huffman coding is Shannon–Fano coding. This coding divides the set of symbols into two equal or almost equal subsets based on the probability of occurrence of characters in each subset. The first subset is assigned a binary 0, the second a binary 1. Huffman encoding always generates optimal codes, but Shannon–Fano sometimes uses a few more bits.

Decompression of LZ77-compressed text is simple and fast. Whenever a (position, length) pair is encountered, one goes to that *position* in that window and copies *length* bytes to the output.

### 10.1.4 Radix-64 Conversion

When PGP is used, usually part of the block to be transmitted is encrypted. If only the signature service is used, then the message digest is encrypted (with the sender's private key). If the confidentiality service is used, the message plus signature (if present) are encrypted (with a one-time symmetric key). Thus, part or all of the resulting block consists of a stream of arbitrary 8-bit octets. However, many electronic mail systems only permit the use of blocks consisting of ASCII text. To accommodate this restriction, PGP provides the service of converting the raw 8-bit binary octets to a stream of printable 7-bit ASCII characters, called *radix-64 encoding* or *ASCII Armor*. Therefore, to transport PGP's raw binary octets through unreliable channels, a printable encoding of these binary octets is needed.

The scheme used for this purpose is radix-64 conversion. Each group of 3 octets of binary data is mapped into four ASCII characters. This format also appends a Cyclic Redundancy Check (CRC) to detect transmission errors. This radix-64 conversion is a wrapper around the binary PGP messages and is used to protect the binary messages during transmission over nonbinary channels, such as Internet e-mail.

Table 10.1 shows the mapping of 6-bit input values to characters. The character set consists of the upper- and lower-case letters, the digits 0–9, and the characters '+' and '/'. The '=' character is used as the padding character. The hyphen "-" character is not used.

Thus, a PGP text file resulting from ASCII characters will be immune to the modifications inflicted by mail systems. It is possible to use PGP to convert any arbitrary file to ASCII Armor. When this is done, PGP tries to compress the data before it is converted to radix-64.

***Example 10.1*** Consider the mapping of a 24-bit input (a block of 3 octets) into a four-character output consisting of the 8-bit set in the 32-bit block.

Suppose the 24-bit raw text is:

10110010 01100011 00101001

The hexadecimal representation of this text sequence is b2 63 29.

**Table 10.1** Radix-64 encoding

| 6-Bit value | Character encoding | 6-Bit value | Character encoding | 6-Bit value | Character encoding | 6-Bit value | Character encoding |
|---|---|---|---|---|---|---|---|
| 0 | A | 16 | Q | 32 | g | 48 | w |
| 1 | B | 17 | R | 33 | h | 49 | x |
| 2 | C | 18 | S | 34 | i | 50 | y |
| 3 | D | 19 | T | 35 | j | 51 | z |
| 4 | E | 20 | U | 36 | k | 52 | 0 |
| 5 | F | 21 | V | 37 | l | 53 | 1 |
| 6 | G | 22 | W | 38 | m | 54 | 2 |
| 7 | H | 23 | X | 39 | n | 55 | 3 |
| 8 | I | 24 | Y | 40 | o | 56 | 4 |
| 9 | J | 25 | Z | 41 | p | 57 | 5 |
| 10 | K | 26 | a | 42 | q | 58 | 6 |
| 11 | L | 27 | b | 43 | r | 59 | 7 |
| 12 | M | 28 | c | 44 | s | 60 | 8 |
| 13 | N | 29 | d | 45 | t | 61 | 9 |
| 14 | O | 30 | e | 46 | u | 62 | + |
| 15 | P | 31 | f | 47 | v | 63 | / |
| | | | | | | (pad) | = |

Arranging this input sequence in blocks of 6 bits yields:

101100  100110  001100  101001

The extracted 6-bit decimal values are 44, 38, 12, and 41.
Referring to Table 10.1, the radix-64 encoding of these decimal values produces the following characters:

smMp

If these characters are stored in 8-bit ASCII format with zero parity, we have them in hexadecimal as follows:

73 6d 4d 70

In binary representation, this becomes:

01110110  01101101  01001101  01110000

### *ASCII Armor Format*

When PGP encodes data into ASCII Armor, it puts specific headers around the data, so PGP can construct the data later. PGP informs the user about what kind of data is encoded in ASCII Armor through the use of the headers.

Concatenating the following data creates ASCII Armor: an Armor head line, Armor headers, a blank line, ASCII-Armored data, Armor checksum, and Armor tail. Specifically, an explanation for each item is as follows:

- *An Armor head line.* This consists of the appropriate header line text surrounded by five dashes ("-", 0x2D) on either side of the header line text. The header line text is chosen based upon the type of data that is being encoded in Armor, and how it is being encoded. Header line texts include the following strings:
  - BEGIN PGP MESSAGE – used for signed, encrypted, or compressed files.
  - BEGIN PGP PUBLIC KEY BLOCK – used for armoring public keys.
  - BEGIN PGP PRIVATE KEY BLOCK – used for armoring private keys.
  - BEGIN PGP MESSAGE, PART X/Y – used for multipart messages, where the armour is divided among Y parts, and this is the Xth part out of Y.
  - BEGIN PGP MESSAGE, PART X – used for multipart messages, where this is the Xth part of an unspecified number of parts and requires the MESSAGE-ID Armor header to be used.
  - BEGIN PGP SIGNATURE – used for detached signatures, PGP/MIME signatures, and natures following clear-signed messages. Note that PGP 2.xs BEGIN PGP MESSAGE is used for detached signatures.

- *Armor headers.* There are pairs of strings that can give the user or the receiving PGP implementation some information about how to decode or use the message. The Armor headers are a part of the armor, not a part of the message, and hence are not protected by any signatures applied to the message. The format of an Armor header is that of a (key, value) pair. A colon (":" 0x38) and a single space (0x20) separate the key and value. PGP should consider improperly formatted Armor headers to be corruptions of ASCII Armor. Unknown keys should be reported to the user, but PGP should continue to process the message.

  Currently defined Armor header keys include:
  - *Version.* This states the PGP version used to encode the message.
  - *Comment.* This is a user-defined comment.
  - *MessageID.* This defines a 32-character string of printable characters. The string must be the same for all parts of a multipart message that uses the "PART X" Armor header. MessageID string should be unique enough that the recipient of the mail can associate all the parts of a message with each other. A good checksum or cryptographic hash function is sufficient.
  - *Hash.* This is a comma-separated list of hash algorithms used in the message. This is used only in clear-signed messages.
  - *Charset.* This is a description of the character set that the plaintext is in. PGP defines text to be in UTF-8 by default. An implementation will get the best results by translating into and out of UTF-8 (see RFC 2279). However, there are many instance where this is easier *said* than *done*. Also, there are communities of users who have no need for UTF-8 because they are all satisfied with a character set like ISO Latin-5 or a Japanese one. In such instances, an implementation may override the UTF-8 default by using this header key.

- *A blank line.* This indicates zero length or contains only white space.

- *ASCII-Armored data.* An arbitrary file can be converted to ASCII-Armored data by using Table 10.1.

- *Armor checksum.* This is a 24-bit CRC converted to four characters of radix-64 encoding by the same MIME base 64 transformation, preceded by an equals sign (=). The CRC is computed by using the generator 0x864cfb and an initialization of 0xb704ce. The accumulation is done on the data before it is converted to radix-64, rather than on the converted data. The checksum with its leading equals sign may appear on the first line after the base 64 encoded data.
- *Armor tail.* The Armor tail line is composed in the same manner as the Armor header line, except the string "BEGIN" is replaced by the string "END".

### *Encoding Binary in Radix-64*

The encoding process represents three 8-bit input groups as output strings of four encoded characters. These 24 bits are then treated as four concatenated 6-bit groups, each of which is translated into a single character in the radix-64 alphabet. Each 6-bit group is used as an index. The character referenced by the index is placed in the output string.

Special processing is performed if fewer than 24 bits are available at the end of the data being encoded. There are three possibilities:

1. The last data group has 24 bits (3 octets). No special processing is needed.
2. The last data group has 16 bits (2 octets). The first two 6-bit groups are processed as above. The third (incomplete) data group has two zero-value bits added to it, and is processed as above. A pad character (=) is added to the output.
3. The last data group has 8 bits (1 octet). The first 6-bit group is processed as above. The second (incomplete) data group has four zero-value bits added to it, and is processed as above. Two pad characters (=) are added to the output.

Radix-64 printable encoding of binary data is shown in Figure 10.3.

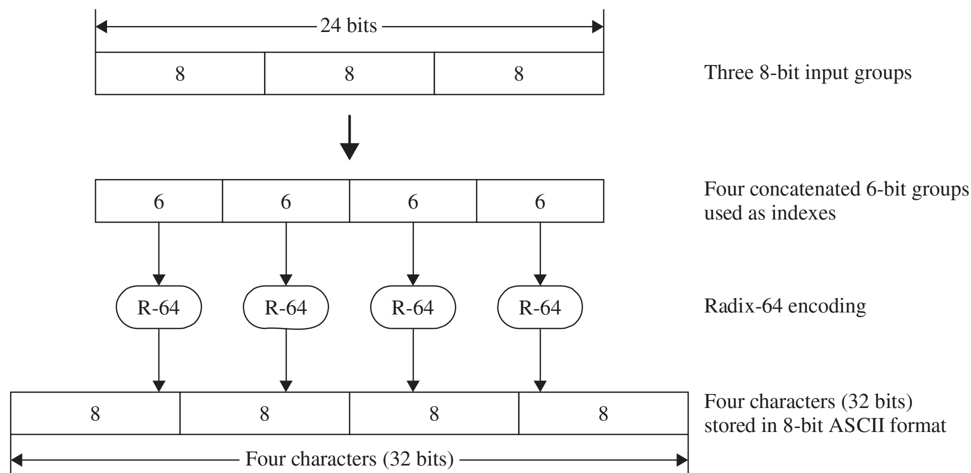



**Figure 10.3** Radix-64 printable encoding of binary data.

***Example 10.2*** Consider the encoding process from 8-bit input groups to the output character string in the radix-64 alphabet.

1. Input raw text: 0x 15 d0 2f 9e b7 4c

8-bit octets: 00010101 11010000 00101111 10011110 10110111 01001100
6-bit index: 000101 011101 000000 101111 100111 101011 011101 001100
Decimal: 5 29 0 47 39 43 29 12
Output character: F d A v n r d M
(radix-64 encoding)
ASCII format (0x): 46 64 41 76 6e 72 64 4d
Binary: 01000110 01100100 01000001 01110110
01101110 01110010 01100100 01001101

2. Input raw text: 0x 15 d0 2f 9e b7

8-bit octets: 00010101 11010000 00101111 10011110 10110111
6-bit index: 000101 011101 000000 101111 100111 101011 011100
Pad with 00 (=)
Decimal: 5 29 0 47 39 43 28
Output character: F d A v n r c =

3. Input raw text: 0x 15 d0 2f 9e

8-bit octets: 00010101 11010000 00101111 10011110
6-bit index: 000101 011101 000000 101111 100111 100000
Pad with 0000 (==)
Decimal: 5 29 0 47 39 32
Output character: F d A v n g ==

### 10.1.5 Packet Headers

A PGP message is constructed from a number of packets. A packet is a chunk of data which has a tag specifying its meaning. Each packet consists of a packet header of variable length, followed by the packet body.

The first octet of the packet header is called the *packet tag* as shown in Figure 10.4. The MSB is “bit 7” (the leftmost bit) whose mask is 0x80 (10000000) in hexadecimal. PGP 2.6.x only uses old format packets. Hence, software that interoperates with PGP 2.6.x must only use old format packets. These packets have 4 bits of content tags, but new format packets have 6 bits of content tags.

#### *Packet Tags*

The packet tag denotes what type of packet the body holds. The defined tags (in decimal) are:

0 –Reserved
1 –Session key packet encrypted by public key

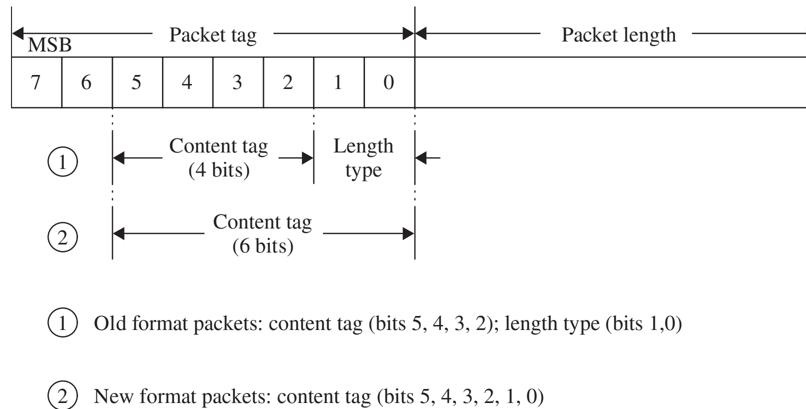



**Figure 10.4** Packet header.

2 –Signature packet
3 –Session key packet encrypted by symmetric key
4 –One-pass signature packet
5 –Secret-key packet
6 –Public-key packet
7 –Secret-subkey packet
8 –Compressed data packet
9 –Symmetrically encrypted data packet
10 –Marker packet
11 –Literal data packet
12 –Trust packet
13 –User ID packet
14 –Public subkey packet
60 ~ 63–Private or experimental values.

### *Old-format Packet Lengths*

The meaning of the length type in old-format packets is:

0 –The packet has a 1-octet length. The header is 2 octets long.
1 –The packet has a 2-octet length. The header is 3 octets long.
2 –The packet has a 4-octet length. The header is 5 octets long.
3 –The packet is of indeterminate length. An implementation should not use indeterminate length packets except where the end of data will be clear from the context. It is better to use a new-format header described below.

### *New-format Packet Lengths*

New-format packets have four possible ways of encoding length.

- *One-octet lengths.* A 1-octet body length header encodes packet lengths from 0 to 191 octets. This type of length header is recognized because the 1-octet value is less than 192. The body length is equal to:

  $\text{bodyLen} = \text{1st_octet}$

- *Two-octet lengths.* A 2-octet body length header encodes a length from 192 to 8383 octets. It is recognized because its first octet is in the range 192–223. The body length is equal to:

  $\text{bodyLen} = ((\text{1st_octet} - 192) \ll 8) + (\text{2nd_octet}) + 192$

- *Five-octet lengths.* A 5-octet body length header encodes packet lengths of up to 4 294 967 295(0xffffffff) octets in length. This header consists of a single octet holding the value 255, followed by a 4-octet scalar. The body length is equal to:

  $\text{bodyLen} = (\text{2nd_octet} \ll 24)|(\text{3rd_octet} \ll 16)|(\text{4th_octet} \ll 8)|\text{5th_octet}$

- *Partial body lengths.* A partial body length header is 1 octet long and encodes the length of only part of the data packet. This length is a power of 2, from 1 to 1 073 741 824 (2 to the 30th power). It is recognized by its 1-octet value that is greater than or equal to 224 and less than 255. The partial body length is equal to:

  $\text{partialBodyLen} = 1 \ll (\text{1st_octet} \quad \text{0x1f})$

Each partial body length header is followed by a portion of the packet body data. The header specifies this portion's length. Another length header (of one of the three types: 1 octet, 2 octet, or partial) follows that portion. The last length header in the packet *must not* be a partial body length header. The latter headers may only be used for the nonfinal parts of the packet.

***Example 10.3*** Consider a packet with length 100. Compute its length encoded in 1 octet.
Now:

$100\ (\text{decimal}) = 2^6 + 2^5 + 2^2 = 01100100(\text{binary}) = \text{0x64}\ (\text{hex})$

Thus, a packet with length 100 may have its length encoded in 1 octet: 0x64. This header is followed by 100 octets of data. Similarly, a packet with length 1723 may have its length encoded in 2 octets: 0xc5 and 0xfb. This header is followed by the 1723 octets of data. A packet with length 100 000 may have its length encoded in 5 octets: 0xff, 0x00, 0x01, 0x86, and 0xa0.

### 10.1.6 PGP Packet Structure

A PGP file consists of a message packet, a signature packet, and a session key packet.

### *Message Packet*

This packet includes the actual data to be transmitted or stored as well as a header that includes control information generated by PGP such as a filename and a timestamp. A timestamp specifies the time of creation. The message component consists of a single literal data packet.

### *Signature Packet* (*Tag 2*)

This packet describes a binding between some public key and some data. The most common signatures are a signature of a file or a block of text and a signature that is a certification of a user ID.

Two versions of signature packets are defined. PGP 2.6.x only accepts version 3 signature. Version 3 provides basic signature information, while version 4 provides an expandable format with subpackets that can specify more information about the signature. It is reasonable to create a v3 signature if an implementation is creating an encrypted and signed message that is encrypted with a v3 key.

At first, version 3 for basic signature information will be presented in the following. The signature packet is the signature of the message component, formed using a hash code of the message component and sender a's public key. The signature component consists of single signature packet.

The signature includes the following components:

- *Timestamp.* This is the time at which the signature was created.
- *Message digest* (*or hash code*). A hash code represents the 160-bit SHA-1 digest, encrypted with sender a's private key. The hash code is calculated over the signature timestamp concatenated with the data portion of the message component. The inclusion of the signature timestamp in the digest protects against replay attacks. The exclusion of the filename and timestamp portion of the message component ensures that detached signatures are exactly the same as attached signatures prefixed to the message. Detached signatures are calculated on a separate file that has none of the message component header fields.

If the default option of compression is chosen, then the block consisting of the literal data packet and the signature packet is compressed to form a compressed data packet:

- *Leading 2 octets of hash code.* These enable the recipient to determine if the correct public key was used to decrypt the hash code for authentication, by comparing the plaintext copy of the first 2 octets with the first 2 octets of the decrypted digest. Two octets also serve as a 16-bit frame-check sequence for the message.
- *Key ID of sender's public key.* This identifies the public key that should be used to decrypt the hash code and hence identifies the private key that was used to encrypt the hash code.

The message component and signature component (optional) may be compressed using ZIP and may be encrypted using a session key.

There are a number of possible meanings of a signature, which are specified in signature-type octets as shown below:

0x00: Signature of a binary document
0x01: Signature of a canonical text document
0x02: Stand-alone signature
0x10: Generic certification of a user ID and public-key packet
(All PGP key signatures are of this type of certification.)
0x11: Personal certification of a user ID and public-key packet
(The issuer has not carried out any verification of the claim.)
0x12: Casual certification of a user ID and public-key packet
(The issuer has carried out some casual verification of the identity claim.)
0x13: Positive certification of a user ID and public-key packet
(The issuer has carried out substantial verification of the identity claim.)
0x18: Subkey binding signature
(This signature is a statement by the top-level signing key indicating that it owns the subkey.)
0x1f: Signature directly on a key
(This signature is calculated directly on a key. It binds the information in the signature subpackets to the key.)
0x20: Key revocation signature
(This signature is calculated directly when the key is revoked. A revoked key is not to be used.)
0x28: Subkey revocation signature
(This signature is calculated directly when the subkey is revoked. A revoked subkey is not to be used.)
0x30: Certification revocation signature
(This signature revokes an earlier user ID certification signature. It should be issued by the same key that issued the revoked signature or an authorised revocation key.)
0x40: Timestamp signature
(This signature is only meaningful for the timestamp contained in it.)

The contents of the signature packets of version 3 (v3) and version 4 (v4) are illustrated in Table 10.2.

The signature calculation for version 4 signature is based on a hash of the signed data. The data being signed is hashed, and then the signature data from the version number to the hashed subpacket data is hashed. The resulting hash value is what is signed. The left 16 bits of the hash are included in the signature packet to provide a quick test to reject some invalid signatures.

### *Session Key Packets (Tag 1)*

This component includes the session key and the identifier of the receiver's public key that was used by the sender to encrypt the session key. A public-key-encrypted session key packet, $E_{KP_b}(K_s)$, holds the session key used to encrypt a message. The symmetrically encrypted data packets are preceded by one public-key-encrypted session key packet for

**Table 10.2** Signature packet format of version 3 and version 4

| Content | Length in octets | |
|---|---|---|
| | v3 | v4 |
| Version number: v3(3), v4(4) | 1 | 1 |
| Signature type | 1 | 1 |
| Creation time | 4 | – |
| Signer's key ID | 8 | – |
| Public-key algorithm | 1 | 1 |
| Hash algorithm | 1 | 1 |
| Field holding left 16 bits of signed hash value | 2 | 2 |
| One or more MPIs comprising the signature | Algorithm specific * | Algorithm specific |
| Scalar octet count for hashed subpacket data | – | 2 |
| Hashed subpacket data | – | Zero or more subpackets |
| Scalar octet count for all of the unhashed subpackets | – | 2 |
| Unhashed subpacket data | – | Zero or more subpackets |

*Algorithm-specific fields for RSA signature: MPI of RSA signature value $m^d$; algorithm-specific fields for DSA signature: MPI of DSA value $r$, MPI of DSA value $s$. (MPI = Multiprecision Integer)

each PGP 5.x key to which the message is encrypted. The message is encrypted with the session key, and the session key is itself encrypted and stored in the encrypted session key packet. The recipient of the message finds a session key that is encrypted to its public key, decrypts the session key, and then uses the session key to decrypt the message.

The body of this session key component consists of:

- A 1-octet version number which is 3.
- An 8-octet key ID of the public key that the session key is encrypted to.
- A 1-octet number giving the public key algorithm used.
- A string of octets that is the encrypted session key. This string's contents are dependent on the public-key algorithm used:
  - Algorithm-specific fields for RSA encryption: multiprecision integer (MPI) of RSA encrypted value $m^e$-mod $n$.
  - Algorithm-specific fields for ElGamal encryption: MPI of ElGamal value $g^k$ mod $p$; MIP of ElGamal value $my^k$ mod $p$. The value '$m$' is derived from the session key.

If compression has been used, then conventional encryption is applied to the compressed data packet format from the compression of the signature packet and the literal data packet. Otherwise, conventional encryption is applied to the block consisting of the signature packet and the literal data packet. In either case, the ciphertext is referred to as a *conventional-key-encrypted data packet*.

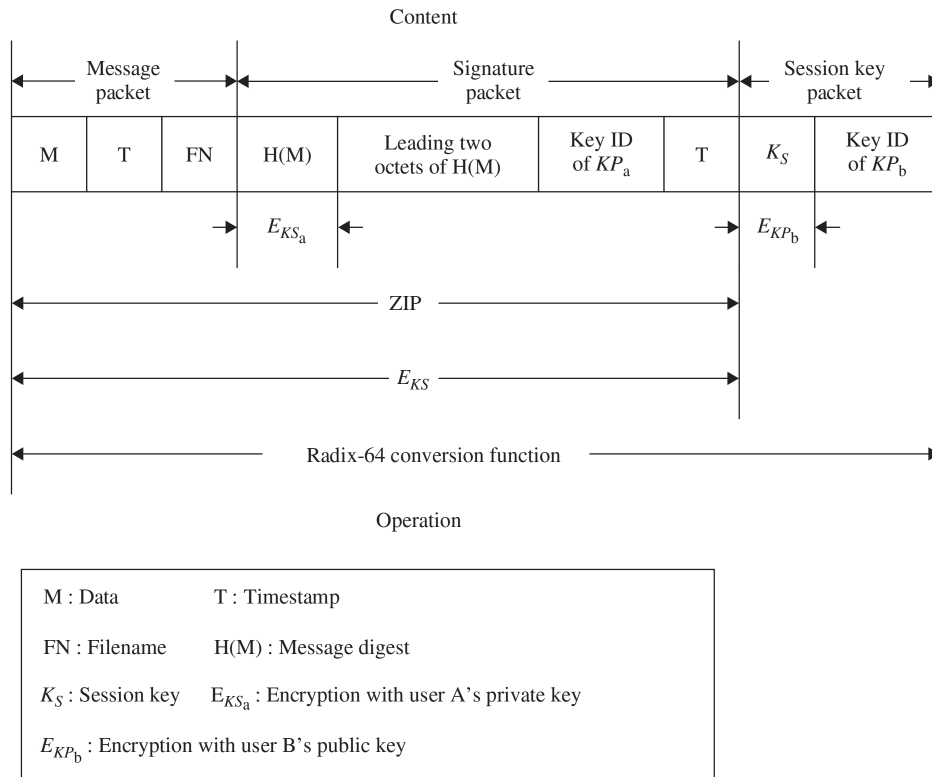



**Figure 10.5** PGP message format.

As shown in Figure 10.5, the entire block of PGP message is usually encoded with radix-64 encoding.

### 10.1.7 Key Material Packet

A key material packet contains all the information about a public or private key. There are four variants of this packet type and two versions.

#### *Key Packet Variants*

There are:

- *Public-key packet* (*tag 6*). This packet starts a series of packets that forms a PGP 5.x key.
- *Public subkey packet* (*tag 14*). This packet has exactly the same format as a public-key packet, but denotes a subkey. One or more subkeys may be associated with a

top-level key. The top-level key provides signature services, and the subkeys provide encryption services. PGP 2.6.x ignores public-subkey packets.

- *Secret-key packet* (*tag 5*). This packet contains all the information that is found in a public-key packet, including not only the public-key materials but also the secret-key material after all the public-key fields.
- *Secret-subkey packet* (*tag 7*). A secret-subkey packet is the subkey analogous to the secret-key packet and has exactly the same format.

### *Public-key Packet Formats*

There are two variants of version 3 packets and version 2 packets. Version 3 packets were originally generated by PGP 2.6. Version 2 packets are identical in format to version 3 packets, but are generated by PGP 2.5. However, v2 keys are deprecated and they must not be generated. PGP 5.0 introduced version 4 packets, with new fields and semantics. PGP 2.6.x will not accept key-material packets with versions greater than 3. PGP 5.x (or PGP3) implementation should create keys with version 4 format, but v4 keys correct some security deficiencies in v3 keys.

A v3 key packet contains:

- A 1-octet version number (3).
- A 4-octet number denoting the time that the key was created.
- A 2-octet number denoting the time in days that this key is valid.
- A 1-octet number denoting the public-key algorithm of this key.
- A series of MPIs comprising the key material: an MPI of RSA public module $n$ and an MPI of RSA public encryption exponent $e$.

A key ID is an 8-octet scalar that identifies a key. For a v3 key, the 8-octet key ID consists of the low 64 bits of the public modulus of the RSA key. The *fingerprint* of a v3 key is formed by hashing the body (excluding the 2-octet length) of the MPIs that form the key material with MD5.

Note that MPIs are unsigned integers. An MPI consists of two parts: a 2-octet scalar that is the length of the MPI in bits followed by a string of octets that contain the actual integer.

***Example 10.4*** Suppose the string of octets [0009 01ff] forms an MPI. The length of the MPI in bits is [00000000 00001001] or 9 ($= 2^3 + 2^0$) in octets. The actual integer value of the MPI is:

$$[01ff] = 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 511$$

The MPI size is:

$$((\text{MPI.length} + 7)/8) + 2 = ((9 + 7)/8) + 2 = 4 \text{ octets}$$

which checks the given size of the MPI string.

The v4 format is similar to the v3 format except for the absence of a validity period. Fingerprints of v4 keys are calculated differently from v3 keys. A v4 fingerprint is the

160-bit SHA-1 hash of the 1-octet packet tag, followed by the 2-octet packet length, followed by the entire public-key packet starting with the version field. The key ID is the low-order 64 bits of the fingerprint.

A v4 key packet contains:

- A 1-octet version number (4).
- A 4-octet number denoting the time that the key was created.
- A 1-octet number denoting the public-key algorithm of this key.
- A series of MPIs comprising the key material:
  - Algorithm-specific fields for RSA public keys: MPI of RSA public modulus $n$ and MPI of RSA public encryption exponent $e$.
  - Algorithm-specific fields for DSA public keys: MPI of DSA prime $p$; MPI of DSA group order $q$ ($q$ is a prime divisor of $p - 1$); MPI of DSA group generator $g$; and MPI of DSA public key value $y = g^x$ where $x$ is secret.
  - Algorithm-specific fields for ElGamal public keys: MPI of ElGamal prime $p$; MPI of ElGamal group generator $g$; and MPI of ElGamal public key value $y = g^x$ where $x$ is secret.

### *Secret-key Packet Formats*

The secret-key and secret-subkey packets contain all the data of public-key and public-subkey packets in encrypted form, with additional algorithm-specific key data appended.

The secret-key packet contains:

- A public-key or public-subkey packet, as described above.
- One octet indicating string-to-key (S2K) usage conventions: 0 indicates that the secret-key data is not encrypted; 255 indicates that an S2K specifier is being given. Any other value specifies a symmetric-key encryption algorithm.
- If the S2K usage octet was 255, a 1-octet symmetric encryption algorithm (optional).
- If the S2K usage octet was 255, an S2K specifier (optional). The length of the S2K specifier is implied by its type, as described above.
- If secret data is encrypted, an 8-octet IV (optional).
- Encrypted MPIs comprising the secret-key data. These algorithm-specific fields are as described below.
- A 2-octet checksum of the plaintext of the algorithm-specific portion (sum of all octets, mod $2^{16}$ = mod 65 536):
  - Algorithm-specific fields for RSA secret keys: MPI of RSA secret exponent $d$; MPI of RSA secret prime value $p$; MPI of RSA secret prime value $q$ ($p < q$); and MPI of $u$, the multiplicative inverse of $p$, mod $q$.
  - Algorithm-specific fields for DSA secret keys: MPI of DSA secret exponent $x$.
  - Algorithm-specific fields for ElGamal secret keys: MPI of ElGamal secret exponent $x$.

Simple S2K directly hashes the string to produce the key data:

Octet 0: 0x00
Octet 1: hash algorithm

It also hashes the *passphrase* to produce the session key. The hashing process to be done depends on the size of the session key and the size of the hash algorithm's output. If the hash size is greater than or equal to the session key size, the higher-order (leftmost) octets of the hash are used as the key. If the hash size is less than the key size, multiple instances are preloaded with 0, 1, 2, ... octets of zeros in order to produce the required key data.

S2K specifiers are used to convert passphrase strings into symmetric-key encryption/decryption keys. They are currently used in two ways: to encrypt the secret part of private keys in the private *keyring* and to convert passphrases to encryption keys for symmetrically encrypted messages.

Secret MPI values can be encrypted using a passphrase. If an S2K specifier is given, it describes the algorithm for converting the passphrase to a key, otherwise a simple MD5 hash of the passphrase is used. The cipher for encrypting the MPIs is specified in the secret-key packet.

Encryption/decryption of the secret data is done in CFB (Cipher Feedback) mode using the key created from the passphrase and IV from the packet. A different mode is used with v3 keys (which are only RSA) than with other key formats. With v3 keys, the prefix data (the first two octets) of the MPI is not encrypted; only the MPI nonprefix data is encrypted. Furthermore, the CFB state is resynchronized at the beginning of each new MPI value, so that the CFB block boundary is aligned with the start of the MPI data. With v4 keys, a simpler method is used: all secret MPI values are encrypted in CFB mode, including the MPI bitcount prefix.

The 16-bit checksum that follows the algorithm-specific portion is the algebraic sum, mod 65 536, of the plaintext of all the algorithm-specific octets (including the MPI prefix and data). With v4 keys, the checksum is encrypted like the algorithm-specific data. This value is used to check that the passphrase was correct.

Besides simple S2K, there are two more S2K specifiers currently supported:

- *Salted S2K.* This includes a *salt* value in the simple S2K specifier that hashes the passphrase to help prevent dictionary attacks:

  Octet 0: 0x01
  Octet 1: hash algorithm
  Octets 2–9: 8-octet salt value

  Salted S2K is exactly like simple S2K, except that the input to the hash function consists of the 8 octets of salt from the S2K specifier, followed by the passphrase.

- *Iterated and salted S2K.* This includes both a salt and an octet count. The salt is combined with the passphrase and the resulting value is hashed repeatedly. This further increases the amount of work an attacker would have to do.

  Octet 0: 0x03
  Octet 1: hash algorithm
  Octets 2–9: 8-octet salt value
  Octet 10: count, a 1-octet, coded value. (The count is coded into a 1-octet number.)

  Iterated–salted S2K hashes the passphrase and salt data multiple times. The total number of octets to be hashed is given in the encoded count in the S2K specifier. But

the resulting count value is an octet count of how many octets will be hashed, not an iteration count. The salt followed by the passphrase data is repeatedly hashed until the number of octets specified by the octet count has been hashed. Implementations should use salted or iterated–salted S2K specifiers because simple S2K specifiers are more vulnerable to dictionary attacks.

### 10.1.8 Algorithms for PGP 5.x

This section describes the algorithms used in PGP 5.x.

*Public-key Algorithms*

| ID | Algorithm |
|---|---|
| 1 | RSA (encrypt or sign) |
| 2 | RSA encryption only |
| 3 | RSA sign only |
| 16 | ElGamal (encrypt only) |
| 17 | DSA (DSS) |
| 18 | Reserved for elliptic curve |
| 19 | Reserved for ECDSA |
| 20 | ElGamal (encrypt or sign) |
| 21 | Reserved for Diffie–Hellman |
| 100–110 | Private/experimental algorithm |

*Symmetric-key Algorithms*

| ID | Algorithm |
|---|---|
| 0 | Plaintext or unencrypted data |
| 1 | IDEA |
| 2 | Triple DES (DES–EDE) |
| 3 | CAST 5 (128-bit key) |
| 4 | Blowfish (128-bit key, 16 rounds) |
| 5 | SAFER-SK128 (13 rounds) |
| 6 | Reserved for DES/SK |
| 7 | Reserved for AES (128-bit key) |
| 8 | Reserved for AES (192-bit key) |
| 9 | Reserved for ASE (256-bit key) |
| 100–110 | Private/experimental algorithm |

*Compression Algorithm*

| ID | Algorithm |
|---|---|
| 0 | Uncompressed |
| 1 | ZIP (RFC 1951) |
| 2 | ZLIB (RFC 1950) |
| 100–110 | Private/experimental algorithm |

*Hash Algorithms*

| ID | Algorithm |
|---|---|
| 1 | MD5 |
| 2 | SHA-1 |
| 3 | RIPE-MD/160 |
| 4 | Reserved for double-width SAH (experimental) |
| 5 | MD2 |
| 6 | Reserved for TIGER/192 |
| 7 | Reserved for HAVAL (5 pass, 160-bit) |
| 100–110 | Private/experimental algorithm |

These tables are not an exhaustive list. An implementation may utilize an algorithm not on these lists.

## 10.2 S/MIME

S/MIME provides a consistent means to send and receive secure MIME data. S/MIME, based on the Internet MIME standard, is a security enhancement to cryptographic electronic messaging. Further, S/MIME not only is restricted to e-mail, but can be used with any transport mechanism that carries MIME data, such as HTTP. As such, S/MIME takes advantage of allowing secure messages to be exchanged in mixed-transport systems. Therefore, it appears likely that S/MIME will emerge as the industry standard for commercial and organizational use. This section describes a protocol for adding digital signature and encryption services to MIME data.

### 10.2.1 MIME

SMTP is a simple mail transfer protocol by which messages are sent only in NVT (Network Virtual Terminal) 7-bit ASCII format. NVT normally uses what is called *NVT ASCII*. This is an 8-bit character set in which the seven lowest-order bits are the same as ASCII and the highest-order bit is zero.

MIME was defined to allow transmission of non-ASCII data through e-mail. MIME allows arbitrary data to be encoded in ASCII and then transmitted in a standard e-mail message. It is a supplementary protocol that allows non-ASCII data to be sent through SMTP. However, MIME is not a mail protocol and cannot replace SMTP; it is only an extension to SMTP. In fact, MIME does not change SMTP or POP3, neither does it replace them.

The MIME standard provides a general structure for the content type of Internet messages and allows extensions for new content-type applications. To accommodate arbitrary data types and representations, each MIME message includes information that tells the recipient the type of the data and the encoding used. The MIME standard specifies that a content-type declaration must contain two identifiers, a content type and a subtype, separated by a slash.

### *MIME Description*

MIME transforms non-ASCII data at the sender's site to NVT ASCII data and delivers it to the client SMTP to be sent through the Internet. The server SMTP at the receiver's site receives the NVT ASCII data and delivers it to MIME to be transformed back to the original non-ASCII data. Figure 10.6 illustrates a set of software functions that transforms non-ASCII data to ASCII data and vice versa.

### *MIME Header*

MIME defines five headers that can be added to the original SMTP header section:

- MIME_Version
- Content_Type
- Content_Transfer_Encoding
- Content_Id
- Content_Description.

The MIMI header is shown in Figure 10.7 and described below.

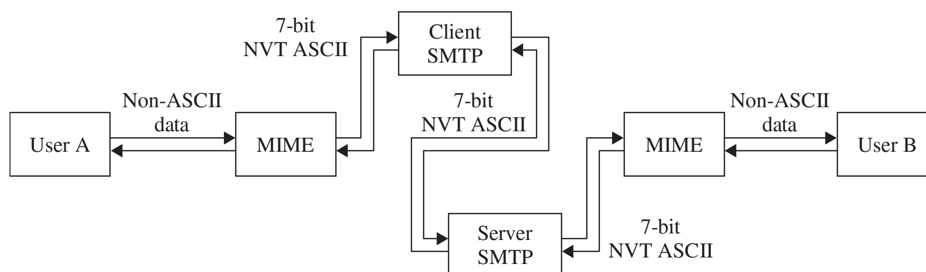



**Figure 10.6** MIME showing a set of transforming functions.

| Original header |
|---|
| MIME header<br>MIME Version: 1.1<br>Content Type: type/subtype<br>Content Transfer Encoding: encoding type<br>Content ID: message ID<br>Content Description: textual explanation of non-textual contents |
| Mail message body |

**Figure 10.7** MIME header.

### *MIME_Version*

This header defines the version of MIME used. The current version is 1.0.

### *Content_Type*

This header defines the type of data used in the message body. The content type and the content subtype are separated by a slash. MIME allows seven different types of data:

- *Text*. The original message is in 7-bit ASCII format.
- *Multipart*. The body contains multiple, independent parts. The multipart header needs to define the boundary between each part. Each part has a separate content type and encoding.

  The multipart/signed content type specifies how to support authentication and integrity services via digital signature.

  Definition of multipart/signed:
  - MIME type name: multipart
  - MIME subtype name: signed
  - Required parameters: boundary, protocol, and micalg
  - Optional parameters: none
  - Security considerations: must be treated as opaque while in transit.

  The multipart/signed content type contains exactly two body parts. The first body part is the one over which the digital signature was created, including its MIME headers. The second body part contains the control information necessary to verify the digital signature.

  Definition of multipart/encrypted:
  - MIME type name: multipart
  - MIME subtype name: encrypted
  - Required parameters: boundary and protocol
  - Optional parameters: none
  - Security considerations: none.

**Table 10.3** Five types of encoding

| Type | Description |
|---|---|
| 7 Bit | NVT ASCII characters and short lines |
| 8 Bit | Non-ASCII characters and short lines |
| Binary | Non-ASCII characters with unlimited-length lines |
| Base64 | 6-Bit blocks of data encoded into 8-bit ASCII characters |
| Quoted-printable | Non-ASCII characters encoded as an equals sign followed by an ASCII code |

The multipart/encrypted content type contains exactly two body parts. The first body part contains the control information necessary to decrypt the data in the second body part and is labeled according to the value of the protocol parameter. The second body part contains the data which was encrypted and is always labeled application/octet-stream.

- *Message.* In the message type, the body is itself a whole mail message, a part of a mail message, or a pointer to the message. Three subtypes are currently used: RFC 2822, partial body, or external body. The subtype RFC 2822 is used if the body is encapsulating another message. The subtype partial is used if the original message has been fragmented into different mail messages and this mail message is one of the fragments. The fragments must be reassembled at the destination by MIME. Three parameters must be added: ID, number, and total. The *id* identifies the message and is present in all the fragments. The *number* defines the sequence order of the fragment. The *total* defines the number of fragments that comprise the original message.
- *Image.* The original message is a stationary image, indicating that there is no animation. The two subtypes currently used are Joint Photographic Experts Group (JPEG), which uses image compression, and Graphics Interchange Format (GIF).
- *Video.* The original message is a time-varying image (animation). The only subtype is Motion Picture Experts Group (MPEG). If the animated image contains sound, it must be sent separately using the audio content type.
- *Audio.* The original message contains sound. The only subtype is basic, which uses 8-kHz standard audio data.
- *Application.* The original message is a type of data not previously defined. There are only two subtypes used currently: octet-stream and PostScript. Octet-stream is used when the data represents a sequence of binary data consisting of 8-bit bytes. PostScript is used when the data is in Adobe PostScript format for printers that support PostScript.

### *Content_Transfer_Encoding*

This header defines the method to encode the messages into ones and zeros for transport. There are the five types of encoding: 7 bit, 8 bit, binary, Base64, and Quoted-printable. Table 10.3 describes the Content_Transfer_Encoding by the five types.

Note that lines in the header identify the type of the data as well as the encoding used.

- *7 Bit.* This is 7-bit NVT ASCII encoding. Although no special transformation is needed, the length of the line should not exceed 1000 characters.
- *8 Bit.* This is 8-bit encoding. Non-ASCII characters can be sent, but the length of the line still should not exceed 1000 characters. Since the underlying SMTP is able to transfer 8-bit non-ASCII characters, MIME does not do any encoding here. Base64 (or radix-64) and quoted-printable types are preferable.
- *Binary.* This is 8-bit encoding. Non-ASCII characters can be sent, and the length of the line can exceed 1000 characters. MIME does not do any encoding here; the underlying SMTP must be able to transfer binary data. Therefore, it is not recommended. Base64 (or radix-64) and quoted-printable types are preferable.
- *Base64.* This is a solution for sending data made of bytes when the highest bit is not necessarily zero. Base64 transforms this type of data of printable characters which can be sent as ASCII characters.
- *Quoted-printable.* Base64 is a redundant encoding scheme. The 24-bit non-ASCII data becomes four characters consisting of 32 bits. We have an overhead of 25%. If the data consists of mostly ASCII characters with a small non-ASCII portion, we can use quoted-printable encoding. If a character is ASCII, it is sent as it is; if a character is not ASCII it is sent as three characters.

### Content_Id

This header uniquely identifies the whole message in a multiple message environment:

Content_Id: id =<content_id>

### Content_Description

This header defines whether the body is image, audio, or video:
Content_Description: <description>

***Example 10.5*** Consider an MIME message that contains a photograph in standard GIF representation. This GIF image is to be converted to 7-bit ASCII using Base64 encoding as follows:

From: myrhee@tsp.snu.ac.kr
To: kiisc2@kornet.net
MIME_Version: 1.1
Content_Type: image/gif
Content_Transfer_Encoding: Base64

. . . data for the gif image . . .

In this example, MIME_Version declares that the message was composed using version 1.1 of the MIME protocol. The MIME standard specifies that a Content_Type declaration

must contain two identifiers, a content type and a subtype, separated by a slash. In this example, *image* is the content type, and *gif* is the subtype. Therefore, the Content_Type declares that the data is a GIF image. For the Content_Transfer_Encoding, the header declares that Base64 encoding was used to convert the image to ASCII. To view the image, a receiver's mail system must first convert from Base64 encoding back to binary, and then run an application that displays a GIF image on the user's screen.

### *MIME Security Multiparts*

An Internet e-mail message consists of two parts: the headers and the body. The headers form a collection of field/value pairs, while the body is defined according to the MIME format. The basic MIME by itself does not specify security protection. Accordingly, a MIME agent must provide security services by employing a security protocol mechanism, by defining two security subtypes of the MIME multipart content type: signed and encrypted. In each of the security subtypes, there are exactly two related body parts: one for the protected data and one for the control information. The type and contents of the control information body parts are determined by the value of the protocol parameter of the enclosing multipart/signed or multipart/encrypted content type. A MIME agent should be able to recognize a security multipart body part and to identify its protected data and control information body part.

The multipart/signed content type specifies how to support authentication and integrity services via digital signature. The multipart/singed content type contains exactly two body parts. The first body part is the one over which the digital signature was created, including its MIME headers. The second body part contains the control information necessary to verify the digital signature. The Message Integrity Check (MIC) is the quantity computed over the body part with a message digest or hash function, in support of the digital signature service. The multipart/encrypted content type specifies how to support confidentiality via encryption. The multipart/encrypted content type contains exactly two body parts. The first body part contains the control information necessary to decrypt the data in the second body part. The second body part contains the data which was encrypted and is always labeled application/octet-stream.

### *MIME Security with OpenPGP*

This subsection describes how the OpenPGP message format can be used to provide privacy and authentication using the MIME security content type. The integrating work on PGP with MIME suffered from a number of problems, the most significant of which was the inability to recover signed message bodies without parsing data structures specific to PGP. RFC 1847 defines security multipart formats for MIME. The security multiparts clearly separate the signed message body from the signature.

PGP can generate either ASCII Armor or a stream of arbitrary 8-bit octets when encrypting data, generating a digital signature, or extracting public-key data. The ASCII Armor output is the required method for data transfer. When the data is to be transmitted in many parts, the MIME message/partial mechanism should be used rather than the multipart ASCII Armor OpenPGP format.

Agents treat and interpret multipart/signed and multipart/encrypted as opaque, which means that the data is not to be altered in any way. However, many existing mail gateways will detect if the next hop does not support MIME or 8-bit data and perform conversion to either quoted-printable or Base64. This presents serious problems for multipart/signed where the signature is invalidated when such an operation occurs. For this reason all data signed according to this protocol must be constrained to 7 bits.

Before OpenPGP encryption, the data is written in MIME canonical format (body and headers). OpenPGP encrypted data is denoted by the *multipart/encrypted* content type, described in the Section MIME Security Multiparts, and must have a protocol parameter value of "application/pgp-encrypted". The multipart/encrypted MIME body must consist of exactly two body parts, the first with content type "application/pgp-encrypted." This body contains the control information. The second MIME body part must contain the actual encrypted data. It must be labeled with a content type of "application/octet-stream."

OpenPGP signed messages are denoted by the multipart/signed content type, described in the Section MIME Security Multiparts, with a protocol parameter which must have a value of "application/pgp-signature". The *micalg* parameter for the "application/pgp-signature" protocol must contain exactly one hash symbol of the format "pgp-<hash-identifier>" where <hash-identifier>identifies the MIC algorithm used to generate the signature. Hash symbols are contracted from text names or by converting the text name to lower case and prefixing it with the four characters "pgp-". Currently defined values are "pgp-md5," "pgp-sha1." "pgp-ripemd160," "pgp-tiger192," and "pgp-haval-5-160." The multipart/signed body must consist of exactly two parts. The first part contains the signed data in MIME canonical format, including a set of appropriate content headers describing the data. The second part must contain the OpenPGP digital signature. It must be labeled with a content type of 'application/pgp-signature.'

When the OpenPGP digital signature is generated:

- The data to be signed must first be converted to its content-type specific canonical form.
- An appropriate Content_Transfer_Encoding is applied. In particular, line endings in the encoded data must use the canonical <CR><LF> sequence where appropriate.
- MIME content headers are then added to the body, each ending with the canonical <CR><LF> sequence.
- Any trailing white space must be removed from the signed material.
- The digital signature must be calculated over both the data to be signed and its set of content headers.
- The signature must be generated as detached from the signed data so that the process does not alter the signed data in any way.

Note that the accepted OpenPGP convention is for signed data to end with a <CR><LF> sequence.

Upon receipt of a signed message, an application must:

- Convert line endings to the canonical <CR><LF>sequence before the signature can be verified.
- Pass both the signed data and its associated content headers along with the OpenPGP signature to the signature verification service.

Sometimes it is desirable both to digitally sign and then to encrypt a message to be sent. This encrypted and signed data protocol allows for two ways of accomplishing this task:

- The data is first signed as a multipart/signature body, and then encrypted to form the final multipart/encrypted body. This is most useful for standard MIME-compliant message forwarding.
- The OpenPGP packet format describes a method for signing and encrypting data in a single OpenPGP message. This method is allowed in order to reduce processing overheads and increase compatibility with non-MIME implementations of OpenPGP. The resulting data is formatted as a "multipart/encrypted" object. Messages which are encrypted and signed in this combined fashion are required to follow the same canonicalization rules as multipart/singed object. It is explicitly allowed for an agent to decrypt a combined message and rewrite it as a multipart/signed object using the signature data embedded in the encrypted version.

A MIME body part of the content type "application/pgp-keys" contains ASCII-Armoured transferable public-key packets as defined in RFC 2440.

Signatures of a canonical text document as defined in RFC 2440 ignore trailing white space in signed material. Implementations which choose to use signatures of canonical text documents will not be able to detect the addition of white space in transit.

### 10.2.2 S/MIME

S/MIME provides a way to send and receive 7-bit MIME data. S/MIME can be used with any system that transports MIME data. It can also be used by traditional mail user agents (MUAs) to add cryptographic security services to mail that is sent, and to interpret cryptographic security services in mail that is received. In order to create S/MIME messages, an S/MIME agent has to follow the specifications discussed in this section, as well as the specifications listed in the cryptographic message syntax (CMS).

The S/MIME agent represents user software that is a receiving agent, a sending agent, or both. S/MIME version 3 agents should attempt to have the greatest interoperability possible with S/MIME version 2 agents. S/MIME version 2 is described in RFC 2311 to RFC 2315 inclusively.

Before using a public key to provide security services, the S/MIME agent must certify that the public key is valid. S/MIME agents must use the Internet X.509 Public-Key Infrastructure (PKIX) certificates to validate public keys as described in the PKIX certificate and CRL profile.

#### *Definitions*

The following definitions are to be applied:

- *ASN.1.* Abstract Syntax Notation One, as defined in ITU-T X.680–689.
- *BER.* Basic Encoding Rules for ASN.1, as defined in ITU-T X.690.
- *DER.* Distinguished Encoding Rules for ASN.1, as defined in ITU-T X.690.

- *Certificate.* A type that binds an entity's distinguished name to a public key with a digital signature. This type is defined in the PKIX certificate and CRL profile. The certificate also contains the distinguished name of the certificate issuer (the signer), an issuer-specific serial number, the issuer's signature algorithm identifier, a validity period, and extensions also defined in that certificate.
- *CRL.* The Certificate Revocation List that contains information about certificates whose validity the issuer has prematurely revoked. The information consists of an issuer name, the time of issue, the next scheduled time of issue, a list of certificate serial numbers and their associated revocation times, and extensions as defined in Chapter 6. The CRL is signed by the issuer.
- *Attribute certificate.* An X.509 AC is a separate structure from a subject's PKIX certificate. A subject may have multiple X.509 ACs associated with each of its PKIX certificates. Each X.509 AC binds one or more attributes with one of the subject's PKIXs.
- *Sending agent.* Software that creates S/MIME CMS objects, MIME body parts that contains CMS objects, or both.
- *Receiving agent.* Software that interprets and processes S/MIME CMS objects, MIME parts that contain CMS objects, or both.
- *S/MIME agent.* User software that is a receiving agent, a sending agent, or both.

### *Cryptographic Message Syntax (CMS) Options*

CMS allows for a wide variety of options in content and algorithm support. This subsection puts forth a number of support requirements and recommendations in order to achieve a base level of interoperability among all S/MIME implementations. CMS provides additional details regarding the use of the cryptographic algorithms.

#### DigestAlgorithmIdentifier

This type identifies a message digest algorithm which maps the message to the message digest. Sending and receiving agents must support SHA-1. Receiving agents should support MD5 for the purpose of providing backward compatibility with MD5-digested S/MIME v2 SignedData objects.

#### SignatureAlgorithmIdentifier

Sending and receiving agents must support id-dsa defined in DSS. Receiving agents should support rsaEncryption, defined in PRCS-1.

#### KeyEncryptionAlgorithmIdentifier

This type identifies a key encryption algorithm under which a content encryption key can be encrypted. A key-encryption algorithm supports encryption and decryption operations. The encryption operation maps a key string to another encrypted key string under the control of a key encryption key.

Sending and receiving agents must support Diffie–Hellman key exchange. Receiving agents should support rsaEncryption. Incoming encrypted messages contain symmetric keys which are to be decrypted with a user's private key. The size of the private key is determined during key generation. Sending agents should support rsaEncryption.

### General syntax

The syntax is to support six different content types: data, signed data, enveloped data, signed-and-enveloped data, digested data, and encrypted data. There are two classes of content types: base and enhanced. Content types in the base class contain just *data* with no cryptographic enhancement, categorized as the data content type. Content types in the enhanced class contain content of some type (possibly encrypted), and other cryptographic enhancements. These types employ encapsulation, giving rise to the terms *outer* content containing the enhancements and *inner* content being enhanced.

CMS defines multiple content types. Of these, only the data, signed data and enveloped data types are currently used for S/MIME.

- *Data content type.* This type is arbitrary octet strings, such as ASCII text files. Such strings need not have any internal structure.
  The data content type should have ASN.1 type Data:

  Data:: = OCTET STRING

  Sending agents must use the id-data content-type identifier to indicate the message content which has had security services applied to it.
- *Signed-data content type.* This type consists of any type and encrypted message digests of the content for zero or more signers. Any type of content can be signed by any number of signers in parallel. The encrypted digest for a signer is a digital signature on the content for that signer. Sending agents must use the signed-data content type to apply a digital signature to a message or in a degenerate case where there is no signature information to convey certificates. The syntax has a degenerate case in which there are no signers on the content. This degenerate case provides a means to disseminate certificates and certificate-revocation lists.

  The process to construct signed data is as follows. A message digest is computed on the content with a signer-specific message digest algorithm. A digital signature is formed by taking the message digest of the content to be signed and then encrypting it with the private key of the signer. The content plus signature are then encoded using Base64 encoding. A recipient verifies the signed-data message by decrypting the encrypted message digest for each signer with the signer's public key, then comparing the recovered message digest to an independently computed message digest. The signer's public key is either contained in a certificate included in the signer information, or referenced by an issuer distinguished name and an issuer-specific serial number that uniquely identify the certificate for the public key.
- *Enveloped-data content type.* An application/prcs7-mime subtype is used for the enveloped-data content type. This content type is used to apply privacy protection to a message. The type consists of encrypted content of any type and encrypted-content

encryption keys for one or more recipients. The combination of encrypted content and encrypted content-encryption key for a recipient is called a *digital envelope* for that recipient. Any type of content can be enveloped for any number of recipients in parallel. If a sending agent is composing an encrypted message to a group of recipients, that agent is forced to send more than one message.

The process by which enveloped data is constructed involves the following:

- A content-encryption key (a pseudorandom session key) is generated at random and is encrypted with the recipient's public key for each recipient.
- The content is encrypted with the content-encryption key. Content encryption may require that the content be padded to a multiple of some block size.
- The recipient-specific information values for all the recipients are combined with the encrypted content into an EnvelopedData value. This information is then encoded into Base64.

To cover the encrypted message, the recipient first strips off the Base64 encoding. The recipient opens the envelope by decrypting one of the encrypted content-encryption keys with the recipient's private key and decrypting the encrypted content with the recovered content-encryption key (the session key).

A sender needs to have access to a public key for each intended message recipient to use this service. This content type does not provide authentication.

- *Digested-data content type.* This type consists of content of any type and a message digest of the content. A typical application of the digested-data content type is to add integrity to content of the data content type, and the result becomes the content input to the enveloped-data content type. A message digest is computed on the content with a message digest algorithm. The message digest algorithm and the message digest are combined with the content into a DigestedData value.

  A recipient verifies the message digest by comparing the message digest to an independently computed message digest.
- *Encrypted-data content type.* This type consists of encrypted content of any type. Unlike the enveloped-data content type, the encrypted-data content type has neither recipients nor encrypted content-encryption keys. Keys are assumed to be managed by other means.

  It is expected that a typical application of the encrypted-data content type will be to encrypt content of the data content type for local storage, perhaps where the encryption key is a password.

### 10.2.3 Enhanced Security Services for S/MIME

The security services described in this section are extensions to S/MIME version 3. Some of the features of each service use the concept of a *triple wrapped* message. A triple wrapped message is one that has been signed, then encrypted, and then signed again. The signers of the inner and outer signatures may be different entities or the same entity. The S/MIME specification does not limit the number of nested encapsulations, so there may be more than three wrappings.

The inside signature is used for content integrity, nonrepudiation with proof of origin, and binding attributes to the original content. These attributes go from the originator to

the recipient, regardless of the number of intermediate entities such as mail list agents that process the message. Signed attributes can be used for access control to the inner body. The encrypted body provides confidentiality, including confidentiality of the attributes that are carried in the inside signature.

The outside signature provides authentication and integrity for information that is processed hop by hop, where each hop is an intermediate entity such as a mail list agent. The outer signature binds attributes to the encrypted body. These attributes can be used for access control and routing decisions.

### *Triple Wrapped Message*

The steps to create a triple wrapped message are as follows:

1. Start with the original content (a message body).
2. Encapsulate the original content with the appropriate MIME content-type headers.
3. Sign the inner MIME headers and the original content resulting from step 2.
4. Add an appropriate MIME construct to the signed message from step 3. The resulting message is called the *inside signature*.
    - If it is signed using multipart/signed, the MIME construct added consists of a content type of multipart/signed with parameters, the boundary, the step 2 result, a content type of application/pkcs7-signature, optional MIME headers, and a body part that is the result of step 3.
    - If it is instead signed using application/pkcs7-mime, the MIME construct added consists of a content type of application/pkcs7-mime with parameters, optional MIME headers, and the result of step 3.
5. Encrypt the step 4 result as a single block, turning it into an application/pkcs7-mime object.
6. Add the appropriate MIME headers: a content type of application/pkcs7-mime with parameters and optional MIME headers such as Content-Transfer-Encoding and Content-Disposition.
7. Sign the step 6 result (the MIME headers and the encrypted body) as a single block.
8. Using the same logic as in step 4, add an appropriate MIME construct to the signed message from step 7. The resulting message is called the *outside signature* and is also the triple wrapped message.

A triple wrapped message has many layers of encapsulation. The structure differs depending on the choice of format for the signed portions of the message. Because of the way that MIME encapsulates data, the layers do not appear in order, and the notion of layers becomes vague.

There is no need to use the multipart/signed format in an inner signature because it is known that the recipient is able to process S/MIME messages. A sending agent might choose to use the multipart/signed format in the outer layer so that a non-S/MIME agent could see that the next inner layer is encrypted. Because many sending agents always use multipart/signed structures, all receiving agents must be able to interpret either multipart/signed or application/pkcs7-mime signature structures.

### *Security Services with Triple Wrapping*

This subsection briefly describes the relationship of each service with triple wrapping. If a signed receipt is requested for a triple wrapped message, the receipt request must be in the inside signature, not in the outside signature. A secure mailing list agent may change the receipt policy in the outside signature of a triple wrapped message when the message is processed by the mailing list.

A security label is included in the signed attributes of any SignedData object. A security label attribute may be included in either the inner signature or the outer signature, or both.

The inner security label is used for access control decisions related to the original plaintext content. The inner signature provides authentication and cryptographically protects the integrity of the original signer's security label that is in the inside body. The confidentiality security service can be applied to the inner security label by encrypting the entire inner SignedData block within an EnvelopedData block. The outer security label is used for access control and routing decisions related to the encrypted message.

Secure mail list message processing depends on the structure of S/MIME layers present in the message sent to the mail list agent. The agent never changes the data that was hashed to form the inner signature, if such a signature is present. If an outer signature is present, then the agent will modify the data that was hashed to form that outer signature.

Contain attributes should be placed in the inner or outer SignedData message. Some attributes must be signed, while signing is optional for others, and some attributes must not be signed.

Some security gateways sign messages that pass through them. If the message is of any type other than a SignedData type, the gateway has only one way to sign the message by wrapping it with a SignedData block and MIME headers. If the message to be signed by the gateway is a SignedData message already, the gateway can sign the message by inserting SignerInfo into the SignedData block.

### *Signed Receipts*

Returning a signed receipt provides to the originator proof of delivery of a message and allows the originator to demonstrate to a third party that the recipient was able to verify the signature of the original message. This receipt is bound to the original message through the signature. Consequently, this service may be requested only if a message is signed. The receipt sender may optionally also encrypt a receipt to provide confidentiality between the sender and the recipient of the receipt.

The originator of a message may request a signed receipt from the message's recipients. The request is indicated by adding a receiptRequest attribute to the signedAttributes field of the SignerInfo object for which the receipt is requested. The receiving user agent software should automatically create a signed receipt when requested to do so, and return the receipt in accordance with mailing list expansion options, local security policies, and configuration options.

Receipts involve the interaction of two parties: the sender and the receiver. The sender is the agent that sent the original message that includes a request for a receipt. The receiver is the party that received that message and generated the receipt.

The interaction steps in a typical transaction are:

1. Sender creates a signed message including a receipt request attribute.
2. Sender transmits the resulting message to the recipient(s).
3. Recipient receives message and determines if there are a valid signature and receipt request in the message.
4. Recipient creates a signed receipt.
5. Recipient transmits the resulting signed receipt message to the sender.
6. Sender receives the message and validates that it contains a signed receipt for the original message.

### *Receipt Request Creation*

Multilayer S/MIME messages may contain multiple SignedData layers. Receipts are requested only for the innermost SignedData layer in a multilayer S/MIME message such as a triple wrapped message. Only one receipt request attribute can be included in the signedAttributes of SignerInfo.

# 9

# Transport Layer Security: SSLv3 and TLSv1

Secure Sockets Layer version 3 (SSLv3) was introduced by Netscape Communications Corporation in 1995. SSLeay implements both SSLv2 and SSLv3 and TLSv1 as of the release of SSLeay-0.9.0. SSLv3 was designed with public review and input from industry and was published as an Internet-Draft document. After reaching a consensus of opinion to Internet standardization, the Transport Layer Security (TLS) Working Group was formed within Internet Engineering Task Force (IETF) in order to develop an initial version of TLS as an Internet standard. The first version of TLS is very closely compatible with SSLv3. The TLSv1 protocol provides communications privacy and data integrity between two communicating parties over the Internet. Both the SSL and TLS protocols allow client/server applications to communicate in such a way that they prevent eavesdropping, tampering, or message forgery. The SSL (or TLS) protocol is composed of two layers: the SSL (or TLS) Record Protocol and the SSL (or TLS) Handshake Protocol.

This chapter is devoted to a full discussion of the protocols of both SSLv3 and TLSv1.

## 9.1 SSL Protocol

SSL is a layered protocol. It is not a single protocol but rather two layers of protocols. At the lower level, the SSL Record Protocol is layered on top of some reliable transport protocol such as TCP. The SSL Record Protocol is also used to encapsulate various higher-level protocols. A higher-level protocol can layer on top of the SSL protocol transparently. For example, the Hypertext Transfer Protocol (HTTP), which provides a transfer service for Web client/server interaction, can operate on top of the SSL Record Protocol.

The SSL Record Protocol takes the upper-layer application message to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies an

*Wireless Mobile Internet Security*, Second Edition. Man Young Rhee.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

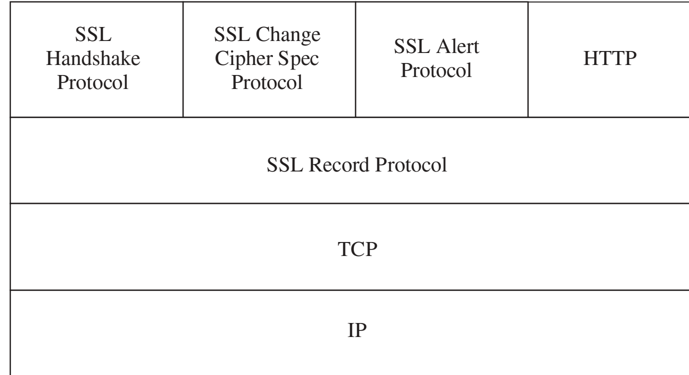



**Figure 9.1** Two-layered SSL protocols.

MAC, encrypts, adds a header, and transmits the result to TCP. The received data is decrypted, verified, decompressed, reassembled, and then delivered to higher-level clients. Figure 9.1 illustrates the overview of the SSL protocol stack.

## 9.1.1 Session and Connection States

There are two defined specifications: SSL session and SSL connection.

### SSL Session

An SSL session is an association between a client and a server. Sessions are created by the Handshake Protocol. They define a set of cryptographic security parameters, which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection. An SSL session coordinates the states of the client and the server. Logically, the state is represented twice as the current operating state and pending state. When the client or server receives a *change cipher spec* message, it copies the pending read state into the current read state. When the client or server sends a *change cipher spec* message, it copies the pending write state into the current write state. When the handshake negotiation is completed, the client and server exchange *change cipher spec* messages, and they then communicate using the newly agreed-upon cipher spec.

The session state is defined by the following elements:

- *Session identifier.* This is a value generated by a server that identifies an active or a resumable session state.
- *Peer certificate.* This is an X.509 v3 certificate of the peer. This element of the state may be null.
- *Compression method.* This is the algorithm used to compress data prior to encryption.

- *Cipher spec.* This specifies the bulk data encryption algorithm (such as null and DES) and a hash algorithm (such as MD5 or SHA-1) used for MAC computation. It also defines cryptographic attributes such as the hash size.
- *Master secret.* This is a 48-byte secret shared between the client and the server. It represents secure secret data used for generating encryption keys, MAC secrets, and IVs.
- *Is resumable.* This designates a flag indicating whether the session can be used to initiate new connections.

### SSL Connection

A connection is a transport (in the Open Systems Interconnect (OSI) layering model definition) that provides a suitable type of service. For SSL, such connections are peer-to-peer relationships. The connections are transient. Every connection is associated with one session.

The connection state is defined by the following elements:

- *Server and client random.* These are byte sequences that are chosen by the server and client for each connection.
- *Server write MAC secret.* This indicates the secret key used in MAC operations on data sent by the server.
- *Client write MAC secret.* This represents the secret key used in MAC operations on data sent by the client.
- *Server write key.* This is the conventional cipher key for data encrypted by the server and decrypted by the client.
- *Client write key.* This is the conventional cipher key for data encrypted by the client and decrypted by the server.
- *Initialization vectors.* When a block cipher in cipher block chaining (CBC) mode is used, an IV is maintained for each key. This field is first initialized by the SSL Handshake Protocol. Thereafter, the final ciphertext block from each record is preserved for use as the IV with the following record. The IV is XORed with the first plaintext block prior to encryption.
- *Sequence numbers.* Each party maintains separate sequence numbers for transmitted and received messages for each connection. When a party sends or receives a change cipher spec message, the appropriate sequence number is set to zero. Sequence numbers may not exceed $2^{64} - 1$.

## 9.1.2 SSL Record Protocol

The SSL Record Protocol provides basic security services to various higher-layer protocols. Three upper-layer protocols are defined as part of SSL: the Handshake Protocol, the Change Cipher Spec Protocol, and the Alert Protocol. Two layers of SSL protocols are shown in Figure 9.1. The SSL Record Layer receives data from higher layers in blocks of arbitrary size.

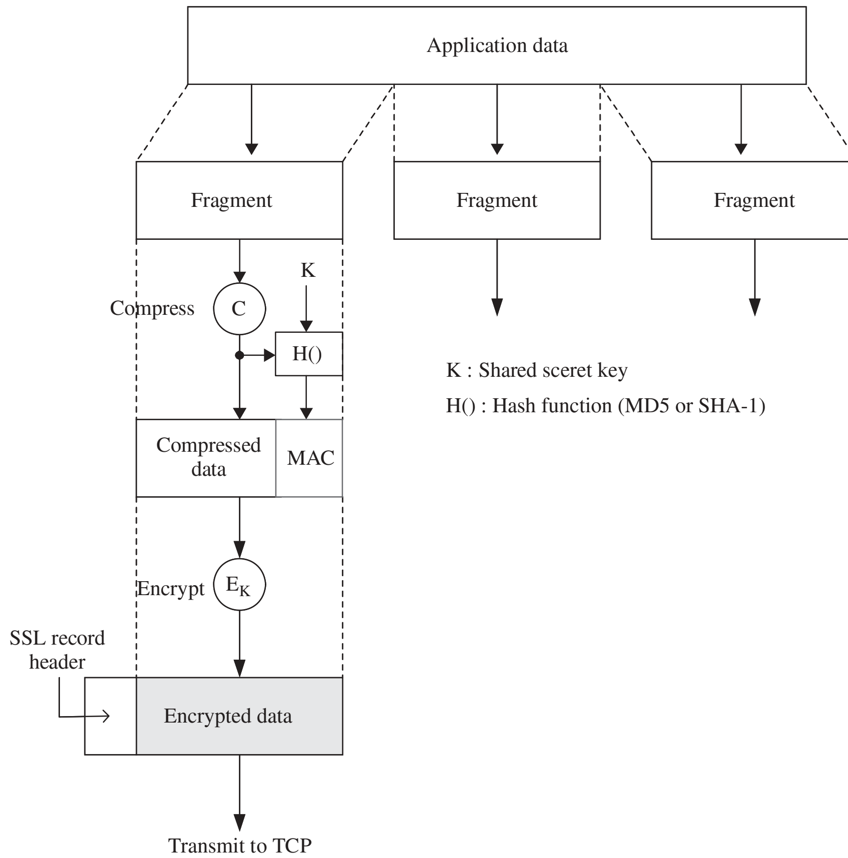



**Figure 9.2** The overall operation of the SSL Record Protocol.

The SSL Record Protocol takes an application message to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies an MAC, encrypts, adds a header, and transmits the result in a TCP segment. The received data is decrypted, verified, decompressed, reassembled, and then delivered to higher-level clients. The overall operation of the SSL Record Protocol is shown in Figure 9.2.

- *Fragmentation.* A higher-layer message is fragmented into blocks (SSLPlaintext records) of $2^{14}$ bytes or less.
- *Compression and decompression.* All records are compressed using the compression algorithm defined in the current session state. The compression algorithm translates an SSLPlaintext structure into an SSLCompressed structure. Compression must be lossless and may not increase the current length by more than 1024 bytes. If the decompression function encounters an SSLCompressed.fragment that would decompress to a length in excess of $2^{14} = 16\,348$ bytes, it should issue a fatal decompression-failure alert.

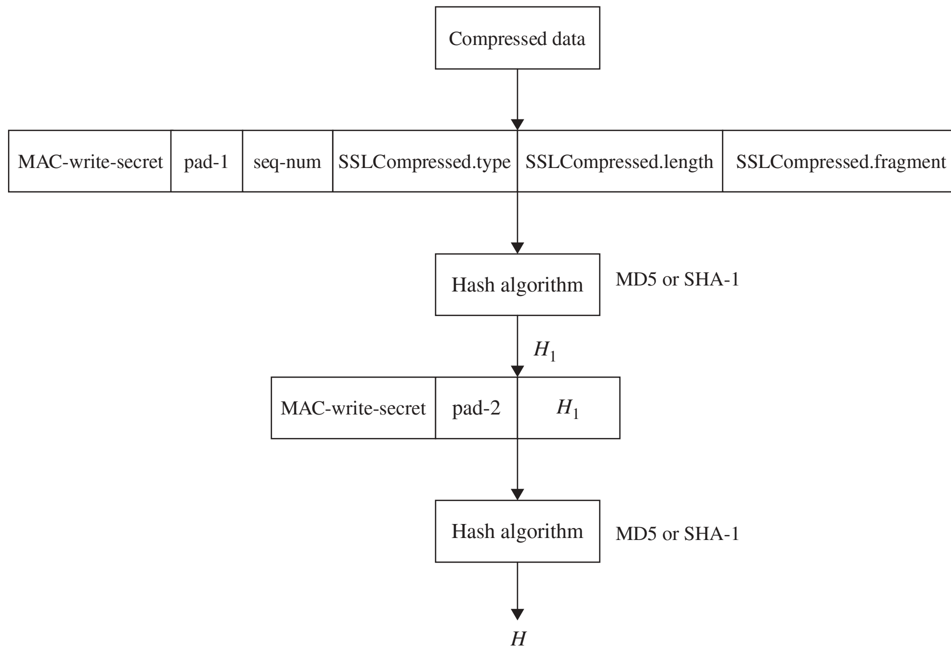



**Figure 9.3** Computation of MAC over the compressed data.

Compression is essentially useful when encryption is applied. If both compression and encryption are required, compression should be applied before encryption. The compression processing should ensure that an SSLPlaintext structure is identical after being compressed and decompressed. Compression is optionally applied in the SSL Record Protocol, but, if applied, it must be done before encryption and MAC computation.

- *MAC*. The MAC is computed before encryption. The computation of an MAC over the compressed data is illustrated in Figure 9.3. Using a shared secret key, the calculation is defined as follows:

$H_1$ = hash(MAC-write-secret ‖ pad-1 ‖ seq-num ‖ SSLCompressed.type ‖
SSLCompressed.length ‖ SSLCompressed.fragment)

$H$ = hash(MAC-write-secret‖ pad-2‖$H_1$)

where

| | |
|---|---|
| MAC-write-secret: | Shared secret key |
| Hash ($H_1$ and $H$): | Cryptographic hash algorithm; either MD5 or SHA-1 |
| Pad-1: | The byte 0x36 (0011 0110) repeated 48 times (384 bits) for MD5 and 40 times (320 bits) for SHA-1 |

| | |
|---|---|
| Pad-2: | The byte 0x5C (0101 1100) repeated 48 times for MD5 and 40 times for SHA-1 |
| Seq-num: | The sequence number for this message |
| SSLCompressed.type: | The higher-level protocol used to process this fragment |
| SSLCompressed.length: | The length of the compressed fragment |
| SSLCompressed.fragment: | The compressed fragment (the plaintext fragment if not compressed) |
| ‖: | concatenation symbol |

The compressed message plus the MAC are encrypted using symmetric encryption. The block ciphers being used as encryption algorithms are:

DES(56), Triple DES(168), IDEA(128),

RC5(variable), and Fortezza(80)

where the number inside the brackets indicates the key size. Fortezza is a Personal Computer Memory Card International Association (PCMCIA) card that provides both encryption and digital signing.

For block encryption, padding is added after the MAC prior to encryption. The total size of the data (plaintext plus MAC plus padding) to be encrypted must be a multiple of the cipher's block length. Padding is added to force the length of the plaintext to be a multiple of the block cipher's block length. Padding is formed by appending a single '1' bit to the end of the message and then '0' bits are added, as many as needed. The last 64 bits of the total size of padded data are reserved for the original message length.

For stream encryption, the compressed message plus the MAC are encrypted. Since the MAC is computed before encryption takes place, it is encrypted along with the compressed plaintext.

- *Append SSL record header*. The final processing of the SSL Record Protocol is to append an SSL record header. The composed fields consist of
  - *Content type* (*8 bits*). This field is the higher-layer protocol used to process the enclosed fragment.
  - *Major version* (*8 bits*). This field indicates the major version of SSL in use. For SSLv3, the value is 3.
  - *Minor version* (*8 bits*). This field indicates the minor version of SSL in use. For SSLv3, the value is 0.
  - *Compressed length* (*16 bits*). This field indicates the length in bytes of the plaintext fragment or compressed fragment if compression is required. The maximum value is $2^{14} + 2048$.

Figure 9.4 illustrates the SSL Record Protocol format.

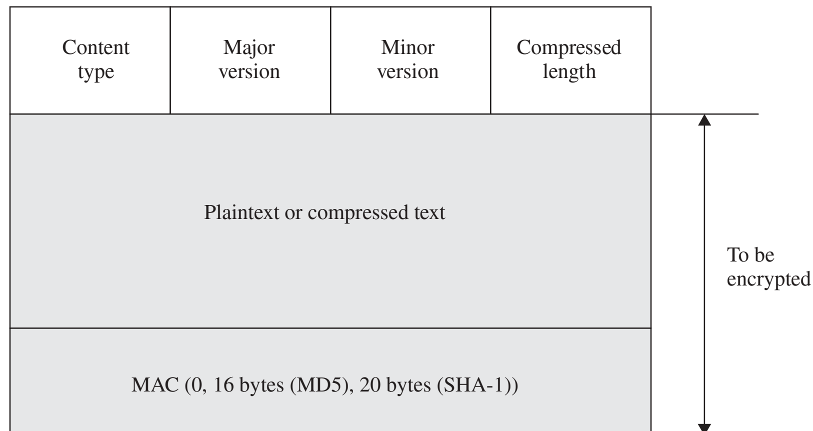



**Figure 9.4** SSL Record Protocol format.

The SSL-specific protocols consist of the Change Cipher Spec Protocol, the Alert Protocol, and the Handshake Protocol, as shown in Figure 9.1. The contents of these three protocols are presented in what follows.

## 9.1.3 SSL Change Cipher Spec Protocol

The Change Cipher Spec Protocol is the simplest of the three SSL-specific protocols. This protocol consists of a single message, which is compressed and encrypted under the current CipherSpec. The message consists of a single byte of value 1. The change cipher spec message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the just-negotiated CipherSpec and keys. Reception of this message causes the pending state to be copied into the current state, which updates the cipher suite to be used on this connection. The client sends a change cipher spec message following handshake key exchange and certificate verify messages (if any), and the server sends one after successfully processing the key exchange message it received from the client.

## 9.1.4 SSL Alert Protocol

One of the content types supported by the SSL Record Layer is the alert type. Alert messages convey the severity of the message and a description of the alert. Alert messages consist of 2 bytes. The first byte takes the value warning or fatal to convey the seriousness of the message. If the level is fatal, SSL immediately terminates the connection. In this case, other connections on the same session may continue, but the session identifiers must be invalidated, preventing the failed session from being used to establish new connections. The second byte contains a code that indicates the specific alert. As with other applications

that use SSL, alert messages are compressed and encrypted, as specified by the current connection state.

A specification of SSL-related alerts that are always fatal is listed in the following:

- *unexpected-message.* An inappropriate message was received. This alert is always fatal.
- *bad-record-mac.* This alert is returned if a record is received with an incorrect MAC. This message is always fatal.
- *decompression-failure.* The decompression function received improper input (i.e., data that would expand to a length that is greater than the maximum allowable length). This message is always fatal.
- *no-certificate.* This alert message may be sent in response to a certificate request if no appropriate certificate is available.
- *bad-certificate.* A received certificate was corrupt, that is, contained a signature that did not verify correctly.
- *unsupported certificate.* The type of the received certificate is not supported.
- *certificate-revoked.* A certificate has been revoked by its signer.
- *certificate-expired.* A certificate has expired or is not currently valid.
- *certificate-unknown.* This means some other unspecified issue arose in processing the certificate, rendering it unacceptable.
- *illegal-parameter.* A field in the handshake was out of range or inconsistent with other fields. This is always fatal.
- *close-notify.* This message notifies the recipient that the sender will not send any more messages on this connection. The session becomes unresumable if any connection is terminated without proper close-notify messages with level equal to warning. Each party is required to send a close-notify alert before closing the write side of the connection. Either party may initiate a close-notify alert. Any data received after a closure alert is ignored.

### 9.1.5 SSL Handshake Protocol

The SSL Handshake Protocol being operated on top of the SSL Record Layer is the most important part of SSL. This protocol provides three services for SSL connections between the server and client. The Handshake Protocol allows the client/server to agree on a protocol version, to authenticate each other by forming an MAC, and to negotiate an encryption algorithm and cryptographic keys for protecting data sent in an SSL record before the application protocol transmits or receives its first byte of data.

The Handshake Protocol consists of a series of messages exchanged by the client and server. Figure 9.5 shows the exchange of handshake messages needed to establish a logical connection between client and server. The contents and significance of each message are presented in detail in the following sections.

#### *Phase 1: Hello Messages for Logical Connection*

The client sends a client hello message to which the server must respond with a server hello message, or else a fatal error will occur and the connection will fail. The client

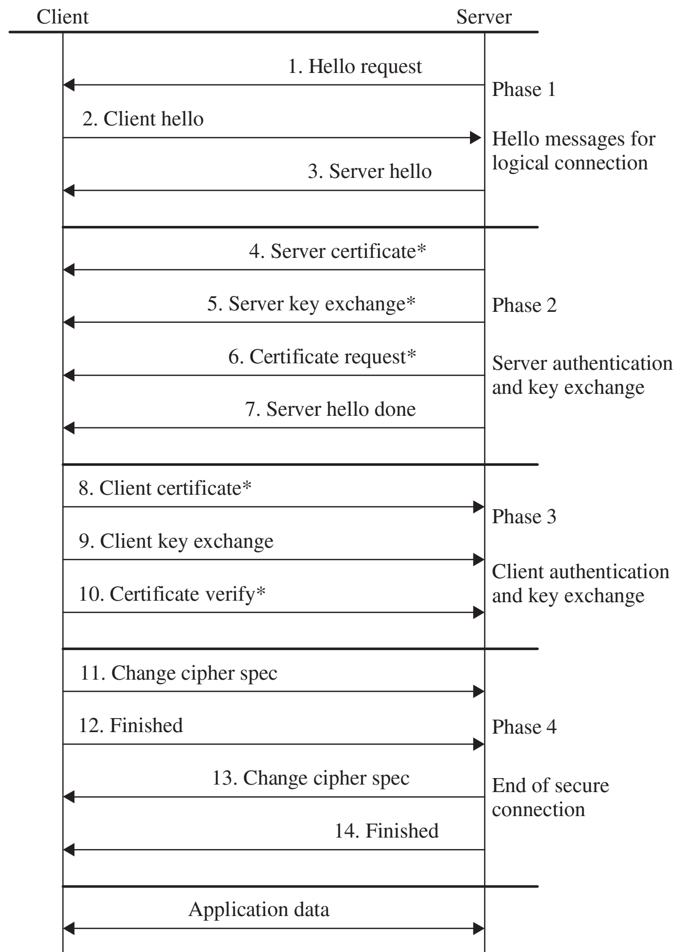



**Figure 9.5** SSL Handshake Protocol.

hello and server hello are used to establish security enhancement capabilities between client and server. The client hello and server hello establish the following attributes: protocol version, random values (ClientHello.random and ServerHello.random), session ID, cipher suite, and compression method.

### Hello messages

The hello phase messages are used to exchange security enhancement capabilities between client and server.

- *Hello request.* This message is sent by the server at any time, but may be ignored by the client if the Handshake Protocol is already underway. A client who receives a hello request while in a handshake negotiation state should simply ignore the message.
- *Client hello.* The exchange is initiated by the client. A client sends a client hello message using the session ID of the session to be resumed. The server then checks its session cache for a match. If a match is found, the server will send a server hello message with the same session ID value. The client sends a client hello message with the following parameters:

  - *Client version.* This is the version of the SSL protocol in which the client wishes to communicate during this session. This should be the most recent (highest-valued) version supported by the client. The value of this version will be 3.0.
  - *Random.* This is a client-generated random structure with 28 bytes generated by a secure random number generator.
  - *Session ID.* This is the identity of a session when the client wishes to use this connection. A nonzero value indicates that the client wishes to update the parameters of an existing connection or create a new connection in this session. A zero value indicates that the client wishes to establish a new connection in a new session.
  - *Cipher suites.* This is a list of the cryptographic options supported by the client, with the client's first preference first. The single cipher suite is an element of a list selected by the server from the list in ClientHello.cipher_suites. For a resumed session, this field is the value from the state of the session being resumed.
  - *Compression method.* This is a list of the compression methods supported by the client, sorted by client preference. If the session ID field is not empty, it must include the compression method from that session.

- *Server hello.* The server will send the server hello message in response to a client hello message when it has found an acceptable set of algorithms. If it is unable to find such a match, it will respond with a handshake failure alert. The structure of this message consists of server version, random, session ID, cipher suite, and compression method.

  - *Server version.* This field will contain the lower-valued version suggested by the client in the client hello and the highest-valued version supported by the server. The value of this version is 3.0.
  - *Random.* This structure is generated by the server and must be different from Client-breakHello.random.
  - *Session ID.* This field represents the identity of the session corresponding to this connection. If the ClientHello.session_id is nonempty, the server will look in its session cache for a match. If a match is found and the server is willing to establish the new connection using the specified session state, the server will respond with the same value as was supplied by the client. This indicates a resumed session and dictates that the parties must proceed directly to the finished messages.
  - *Cipher suite.* This is the single cipher suite selected by the server from the list in ClientHello.cipher_suites. For a resumed session, this field is the value from the state of the session being resumed.

- *Compression method.* This is the single compression algorithm selected by the server from the list in ClientHello.compression_methods. For resumed sessions, this field is the value from the resumed session state.

### *Phase 2: Server Authentication and Key Exchange*

Following the hello messages, the server begins this phase by sending its certificate if it needs to be authenticated. Additionally, a server key exchange message may be sent if it is required. If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected. Then the server will send the server hello done message, indicating that the hello message phase of the handshake is complete. The server will then wait for a client response. If the server has sent a certificate request message, the client must send the certificate message.

- *Server certificate.* If the server is to be authenticated, it must send a certificate immediately following the server hello message. The certificate type must be appropriate for the selected cipher suite's key exchange algorithm, and is generally an X.509 v3 certificate. It must contain a key which matches the key exchange method. The signing algorithm for the certificate must be the same as the algorithm for the certificate key.
- *Server key exchange message.* The server key exchange message is sent by the server only when it is required. This message is not used if the server certificate contains Diffie–Hellman (DH) parameters, or RSA key exchange is to be used for a signature-only RSA.
  - *params.* The server's key exchange parameters.
  - *signed-params.* For nonanonymous key exchange, a hash of the corresponding params value, with the signature appropriate to that hash applied.

  As usual, a signature is created by taking the hash of the message and encrypting it with the sender's public key. Hence, the hash is defined as:

  md5-hash : MD5(ClientHello.random‖ServerHello.random‖serverParams)

  sha-hash : SHA(ClientHello.random‖ServerHello.random‖serverParams)

  enum{anonymous, rsa, dsa}SignatureAlgorithm

  For a Digital Signature Standard (DSS) signature, the hash is performed using the SHA-1 algorithm. In the case of an RSA signature, both an MD5 and an SHA-1 hash are calculated, and the concatenation of the two hashes is encrypted with the server's public key.
- *Certificate request message.* A nonanonymous server can optionally request a certificate from the client, if appropriate for the selected cipher suite. This message includes two parameters, certificate type and certificate authorities. Its structure is as follows:

```
enum{
    rsa_sign(1), des_sign(2), rsa_fixed_dh(3),
```

```
    dss_fixed_dh(4),
    rsa_ephemeral_dh(5), dss_ephemeral_dh(6),
    fortezza_dms(20), (255)
} ClientCertificateType;
opaque DistinguishedName<1..2^16-1>;
struct {
        ClientCertificateType certificate_types<1..2^8-1>;
       DistinguishedName certificate_authorities<3..2^16-1>
} CertificateRequest;
```

- *certificate_types.* This field is a list of the types of certificates requested, sorted in order of the server's preference.
- *certificate_authorities.* This is a list of the distinguished names of acceptable certificate authorities. These distinguished names may specify a desired distinguished name for a root CA or for a subordinate CA; thus, this message can be used to describe both known roots and a desired authorization space.

Note that DistinguishedName is derived from X.509 and that it is a fatal handshake_ failure alert for an anonymous server to request client identification.

- *Server hello done message.* This message is sent by the server to indicate the end of the server hello and associated messages. After sending this message, the server will wait for a client response. This message means that server has finished sending messages to support the key exchange, and the client can proceed with its phase of the key exchange. Upon receipt of the server hello done message, the client should verify that the server provided a valid certificate if required and check that the server hello parameters are acceptable. If all is satisfactory, the client sends one or more messages back to the server.

### *Phase 3: Client Authentication and Key Exchange*

If the server has sent a certificate request message, the client must send the certificate message. The client key exchange message is then sent, and the content of that message will depend on the public key algorithm selected between the client hello and the server hello. If the client has sent a certificate with signing ability, a digitally signed certificate verify message is sent to explicitly verify the certificate.

- *Client certificate message.* This is the first message the client can send after receiving a server hello done message. This message is sent only when the server requests a certificate. If no suitable certificate is available, the client should send a certificate message containing no certificates. If client authentication is required by the server for the handshake to continue, it may respond with a fatal handshake failure alert. The same message type and structure will be used for the client's response to a certificate request message. Note that a client may send no certificates if it does not have an appropriate certificate to send in response to the server's authentication request. The client's DH certificates must match the server-specified DH parameters.

- *Client key exchange message.* This message is always sent by the client. It will immediately follow the client certificate message, if it is sent. Otherwise, it will be the first message sent by the client after it receives the server hello done message. With this message, the premaster secret is set, either through direct transmission of the RSA-encrypted secret, or by transmission of DH parameters which will allow each side to agree upon the same premaster secret. When the key exchange method is DH−RSA or DH−DSS, client certification has been requested, and the client was able to respond with a certificate which contained a DH public key whose parameters matched those specified by the server in its certificate; this message will not contain any data.
- *Certificate verify message.* This message is used to provide explicit verification of a client certificate. The message is only sent following any client certificate that has signing capability (i.e., all certificates except those containing fixed DH parameters). When sent, it will immediately follow the client key exchange message. This message signs a hash code based on the preceding messages, and its structure is defined as follows:

```
struct{
      Signature signature;
} CertificateVerify;
CertificateVerify.signature.md5_hash
     MD5(master_secret||pad2||MD5(handshake-message||
          master_secret||pad1))
Certificate.signature.sha_hash
    SHA(master_secret||pad2||SHA(handshake-message||
         master_secret||pad1))
```

where pad1 and pad2 are the values defined earlier for the MAC, handshake messages refer to all Handshake Protocol messages sent or received starting at client hello but not including this message, and master_secret is the calculated secret. If the user's private key is DSS, then it is used to encrypt the SHA-1 hash. If the user's private key is RSA, it is used to encrypt the concatenation of the MD5 and SHA-1 hashes.

### *Phase 4: End of Secure Connection*

At this point, a change cipher spec message is sent by the client, and the client copies the pending CipherSpec into the current CipherSpec. The client then immediately sends the finished message under the new algorithms, keys, and secrets. In response, the server will send its own change cipher spec message, transfer the pending CipherSpec to the current one, and then send its finished message under the new CipherSpec. At this point, the handshake is complete and the client and the server may begin to exchange application layer data (Figure 9.5).

- *Change cipher spec messages.* The client sends a change cipher spec message and copies the pending CipherSpec in the current CipherSpec. This message is immediately sent after the certificate verify message that is used to provide explicit verification of a

client certificate. It is essential that a change cipher spec message is received between the other handshake messages and the finished message. It is a fatal error if a change cipher spec message is not preceded by a finished message at the appropriate point in the handshake.

- *Finished message.* This is always sent immediately after a change cipher spec message to verify that the key exchange and authentication processes were successful. The content of the finished message is the concatenation of two hash values:

```
MD5(master_secret||pad2||MD5(handshake_messages||Sender||
     master_secret||pad1))
SHA(master_secret||pad2||SHA(handshake_messages||Sender||
     master_secret||pad1))
```

where 'Sender' is a code that identifies that the sender is the client and "handshake_ messages" is code that identifies the data from all handshake messages up to but not including this message.

The finished message is first protected with just-negotiated algorithms, keys, and secrets. Recipients of finished messages must verify that the contents are correct. Once a side has sent its finished message and received and validated the finished message from its peer, it may begin to send and receive application data over the connection. Application data treated as *transparent data* is carried by the Record Layer and is fragmented, compressed, and encrypted based on the current connection state.

## 9.2 Cryptographic Computations

The key exchange, authentication, encryption, and MAC algorithms are determined by the cipher suite selected by the server and revealed in the server hello message. The compression algorithm is negotiated in the hello messages, and the random values are exchanged in the hello messages. The creation of a shared master secret by means of the key exchange and the generation of cryptographic parameters from the master secrete are of interest to study as two further items.

### 9.2.1 Computing the Master Secret

For all key exchange methods, the same algorithm is used to convert the premaster secret into the master secret. In order to create the master secret, a premaster secret is first exchanged between two parties and then the master secret is calculated from it. The master secret is always exactly 48 bytes (384 bits) shared between the client and server. But the length of the premaster secret is not fixed and will vary depending on the key exchange method. There are two ways for the exchange of the premaster secret.

- *RSA.* When RSA is used for server authentication and key exchange, a 48-byte premaster secret is generated by the client, encrypted with the server's public key, and sent to the server. The server decrypts the ciphertext (of the premaster secret) using its

private key to recover the premaseter secret. Both parties then convert the premaster secret into the master secret as specified below.

- *Diffie–Hellman.* A conventional DH computation is performed. Both client and server generate a DH common key. This negotiated key is used as the premaster secret and is converted into the master secret, as specified below.

The client and server then compute the master secret as follows:

```
master_secret = MD5(pre_master_secret||SHA('A'||
                pre_master_secret||ClientHello.random||
                    ServerHello.random))||
                MD5(pre_master_secret||SHA('BB'||
                pre_master_secret||ClientHello.random||
                    ServerHello.random))||
                MD5(pre_master_secret||SHA('CCC'||
                pre_master_secret||ClientHello.random||
                    ServerHello.random))
```

where ClientHello.random and ServerHello.random are the two nonce values exchanged in the initial hello messages.

The generation of the master secret from the premaster secret is shown in Figure 9.6.

### 9.2.2 Converting the Master Secret into Cryptographic Parameters

CipherSpec specifies the bulk data encryption algorithm and a hash algorithm used for MAC computation, and defines cryptographic attributes such as the hash size.

To generate the key material, the following is computed.

```
key_block = MD5(master_secret||SHA('A'||master_secret||
             ServerHello.random||ClientHello.random))||
            MD5(master_secret||SHA('BB'||master_secret||
             ServerHello.random||ClientHello.random))||
            MD5(master_secret||SHA('CCC'||master_secret||
             ServerHello.random||ClientHello.random))||...
```

until enough output has been generated. Note that the generation of the key block from the master secret uses the same format for generation of the master secret from the premaster secret. Figure 9.7 illustrates the steps for generation of the key block from the master secret.

## 9.3 TLS Protocol

The TLSv1 protocol itself is based on the SSLv3 protocol specification as published by Netscape. Many of the algorithm-dependent data structures and rules are very close so that the differences between TLSv1 and SSLv3 are not dramatic. The current work on

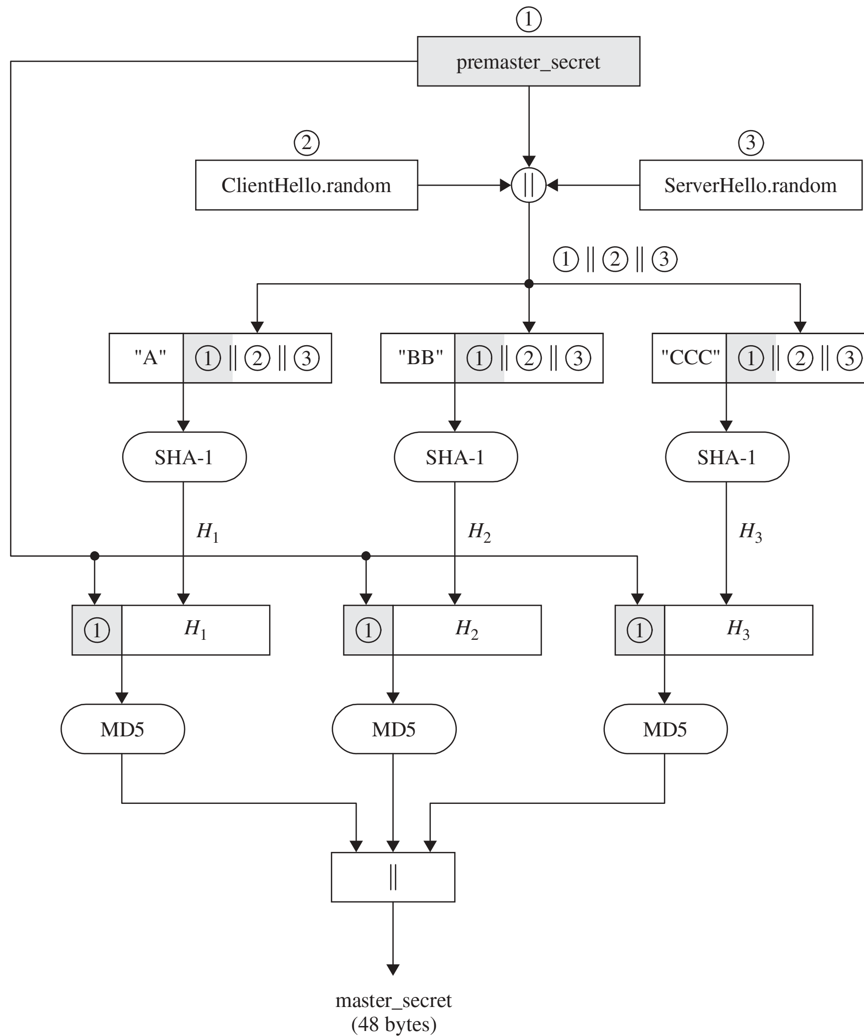



**Figure 9.6** Computation of the master secret.

TLS is aimed at producing an initial version as an Internet standard. It is recommended that readers examine the comparative studies between the TLSv1 of RFC 2246 and SSLv3 of Netscape. In this section, we will not repeat every detailed step of identical protocol contents, but only highlight the differences.

## 9.3.1 HMAC Algorithm

A Keyed-hashing Message Authentication Code (HMAC) is a secure digest of some data protected by a secret. Forging the HMAC is infeasible without knowledge of the MAC

secret. HMAC can be used with a variety of different hash algorithms, namely, MD5 and SHA-1, denoting these as HMAC_MD5(secret, data) and HMAC_SHA-1(secret, data).

There are two differences between the SSLv3 and TLSMAC schemes. TLS makes use of the HMAC algorithm defined in RFC 2104. HMAC was fully discussed in Chapters 4 and 7 and defined as

$$\text{HMAC} = H[(K \oplus \text{opad})\|H[(K \oplus \text{ipad})\|M]]$$

where

ipad = 00110110 (0x36) repeated 64 times (512 bits)
opad = 01011100 (0x5c) repeated 64 times (512 bits)
$H$ = one-way hash function for TLS (either MD5 or SHA-1)
$M$ = message input to HMAC
$K$ = padded secret key equal to the block length of the hash code (512 bits for MD5 and SHA-1)

The following explains the HMAC equation.

1. Append zeros to the end of $K$ to create a $b$-byte string (i.e., if $K = 160$ bits in length and $b = 512$ bits, then $K$ will be appended with 352 zero bits or 44 zero bytes 0x00).
2. XOR (bitwise exclusive OR) $K$ with ipad to produce the $b$-bit block computed in step 1.
3. Append $M$ to the $b$-byte string resulting from step 2.
4. Apply $H$ to the stream generated in step 3.
5. XOR (bitwise exclusive OR) $K$ with opad to produce the $b$-byte string computed in step 1.
6. Append the hash result $H$ from step 4 to the $b$-byte string resulting from step 5.
7. Apply $H$ to the stream generated in step 6 and output the result.

Figure 9.8 illustrates the overall operation of HMAC−MD5 or HMAC−SHA-1.

***Example 9.1*** HMAC−SHA-1 computation using RFC method:

Data : 0x 7104f218 a3192e65 1cf7025d 8011bf79 4a19
Key : 0x 31fa7062 c45113e3 2679fd13 53b71264

| – | A | B | C | D | E |
|---|---|---|---|---|---|
| IV | 67452301 | efcdab89 | 98badcfe | 10325476 | c3d2e1f0 |
| $H[(K \oplus \text{ipad})\|M]$ | 8efeef30 | f64b360f | 77fd8236 | 273f0784 | 613bbd4b |
| $H[(K \oplus \text{opad})\|H[(K \oplus \text{ipad})\|M]]$ | 31db10b8 | ed346850 | d0f0b7dd | 50fd71f4 | 2dacd24c |

HMAC−SHA-1 = 0x 31 db10b8 ed346850 d0f0b7dd 50fd71f4 2dacd24c

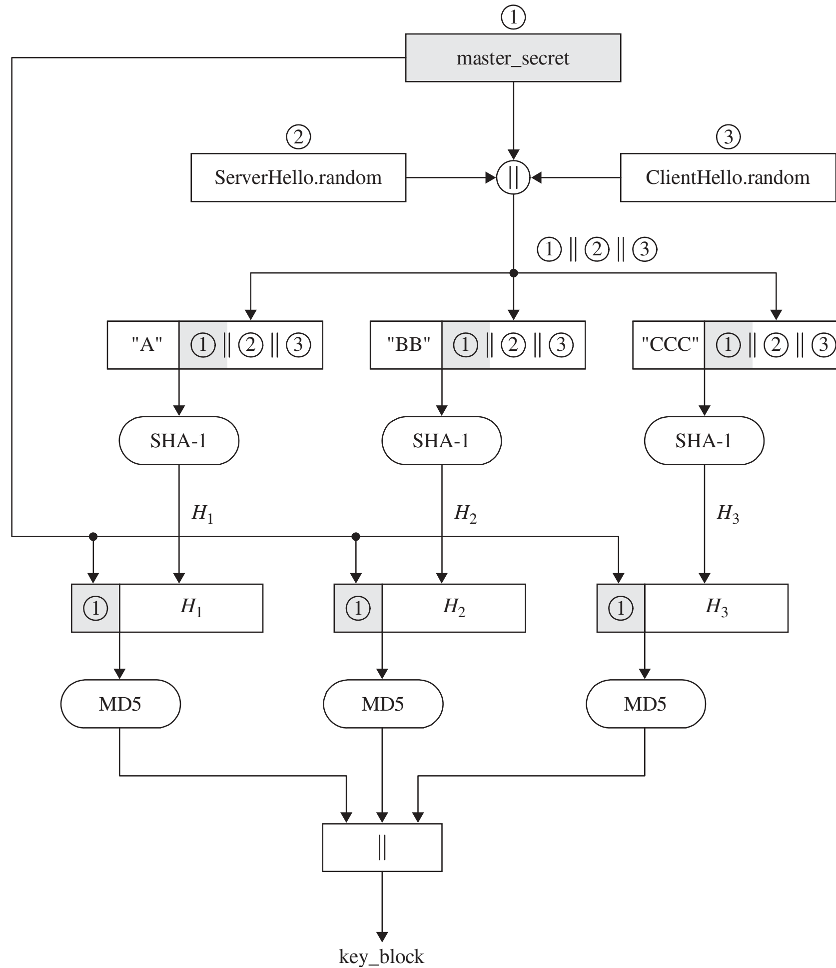



**Figure 9.7** Generation of key block.

The alternative operation for computation of either HMAC–MD5 or HMAC–SHA-1 is described in the following.

1. Append zeros to $K$ to create a $b$-bit string $K'$, where $b = 512$ bits.
2. XOR $K'$ (padding with zero) with ipad to produce the $b$-bit block.
3. Apply the compression function f(IV, $K' \oplus$ ipad) to produce $(\text{IV})_i = 128$ bits.
4. Compute the hash code $h$ with $(\text{IV})_i$ and $M_i$.
5. Raise the hash value computed from step 4 to a $b$-bit string.
6. XOR $K'$ (padded with zeros) with opad to produce the $b$-bit block.
7. Apply the compression function f(IV, $K' \oplus$ opad) to produce $(\text{IV})_o = 128$ bits.
8. Compute the HMAC with $(\text{IV})_o$ and the raised hash value resulting from step 5.

Figure 9.9 illustrates the overall operation of HMAC–MD5.

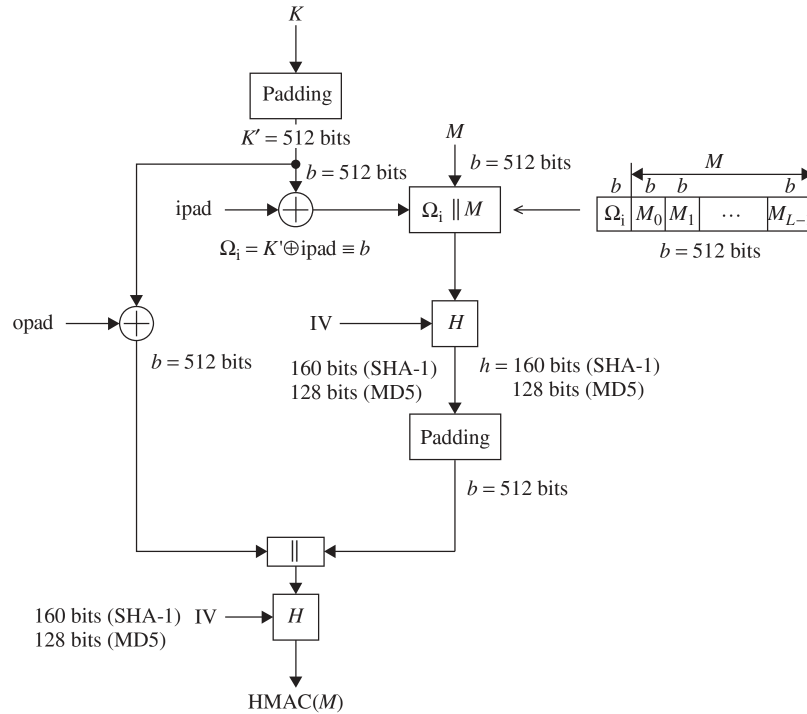



**Figure 9.8** Overall operation of HMAC computation using either MD5 or SHA-1 (message length computation based on $\Omega_i \| M$.)

***Example 9.2*** HMAC−MD5 computation using alternative method:

Data: 0x 2143f501 f014a713 c1059e23 7123fd68
Key: 0x 31fa7062 c45113e3 2679fd13 53b71264

| – | A | B | C | D |
|---|---|---|---|---|
| IV | 67452301 | efcdab89 | 98badcfe | 10325476 |
| $f[(K \oplus \text{ipad}), IV] = (IV)_i$ | 13fbaf34 | 034879ab | 35e73505 | 526a8d28 |
| $H[M, (IV)_i]$ | 90c6d9b0 | 0f281bc8 | 94d04b33 | 7f0f4265 |
| $f[(K \oplus \text{opad}), IV] = (IV)_o$ | 5f8647d7 | fa8e9afa | bffa4989 | 3cd471d1 |
| $H[H[M, (IV)_i], (IV)_o]$ | 2c47cd5b | 68830268 | 7d255059 | 45c7bef0 |

HMAC−MD5 = 0x 2c47cd5b 68830268 7d255059 45c7bef0

For TLS, the MAC computation encompasses the fields indicated in the following expression.

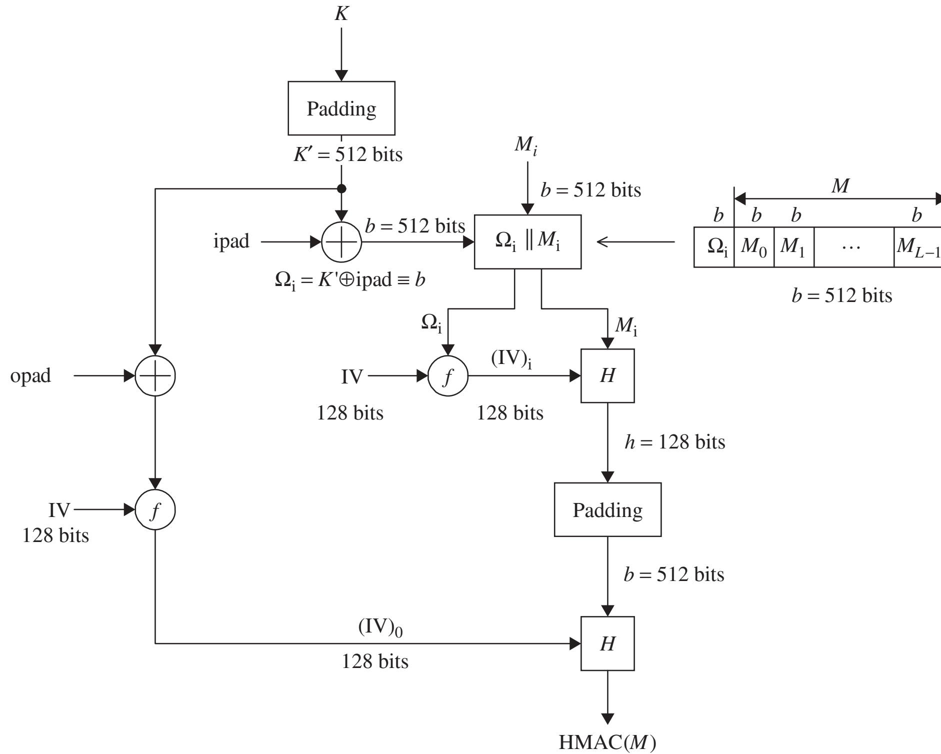



**Figure 9.9** Alternative operation of HMAC computation using MD5 (message length computation is based on *M* only).

```
HMAC_hash(MAC_write_secret, seq_num||TLScompressed.type||
 TLSCompressed.version||TLSCompressed.length||
 TLSCompressed.fragment)
```

Note that the MAC calculation includes all of the fields covered by the SSLv3 computation, plus the field TLSCompressed.version, which is the version of the protocol being employed.

### 9.3.2 Pseudo-random Function

TLS utilizes a pseudo-random function (PRF) to expand secrets into blocks of data for the purposes of key generation or validation. The PRF takes relatively small values such as a secret, a seed, and an identifying label as input and generates an output of arbitrary longer blocks of data.

The data expansion function, P_hash(secret, data), uses a single hash function to expand a secret and seed into an arbitrary quantity of output. The data expansion function is defined as follows:

```
P_hash(secret, seed) = HMAC_hash (secret, A(1)||seed) ||
                       HMAC_hash (secret, A(2)||seed) ||
                       HMAC_hash (secret, A(3)||seed) ||...
```

where A() is defined as

A(0) = seed
A(i) = HMAC_hash(secret, A(i-1)) and || indicates concatenation.

Applying A(i), $i = 0, 1, 2, \ldots$, to P_hash, the resulting sketch can be depicted as shown in Figure 9.10. As you can see, P_hash is iterated as many times as necessary to produce the required quantity of data. Thus, the data expansion function makes use of the HMAC algorithm with either MD5 or SHA-1 as the underlying hash function. As an example, consider SHA-1 whose value is 20 bytes (160 bits). If P_SHA-1 is used to create 64 bytes (512 bits) of data, it will have to be iterated four times up to A(4), creating $20 \times 4 = 80$ bytes (640 bits) of output data. Hence, the last 16 bytes (128 bits) of the final iteration A(4) must be discarded, leaving $(80 - 16) = 64$ bytes of output data. On the other hand, MD5 produces 16 bytes (128 bits). In order to generate an 80-byte output, P_MD5 should exactly be iterated through A(5), while P_SHA-1 will only iterate through A(4) as described above. In fact, alignment to a shared 64-byte output will be required to discard the last 16 bytes from both P_SHA-1 and P_MD5.

TLS's PRF is created by splitting the secret into two halves (S1 and S2) and using one half to generate data with P_MD5 and the other half to generate data with P_SHA-1. These two results are then XORed to produce the output. S1 is taken from the first half of the secret and S2 from the second half. Their length is respectively created by rounding up the length of the overall secret divided by 2. Thus, if the original secret is an *odd* number of bytes long, the last bytes of S1 will be the same as the first byte of S2.

L_S = length in bytes of secret

L_S1 = L_S2 = ceil(L_S/2)

The PRF is then defined as the result of mixing the two pseudo-random streams by XORing them together. The PRF is defined as

$$\text{PRF(secret, label, seed)} = \text{P_MD5(S1, label}\|\text{seed)} \oplus \text{P_SHA} - 1\text{(S2, label}\|\text{seed)}$$

The label is an ASCII string. Figure 9.11 illustrates the PRF generation scheme to expand secrets into blocks of data.

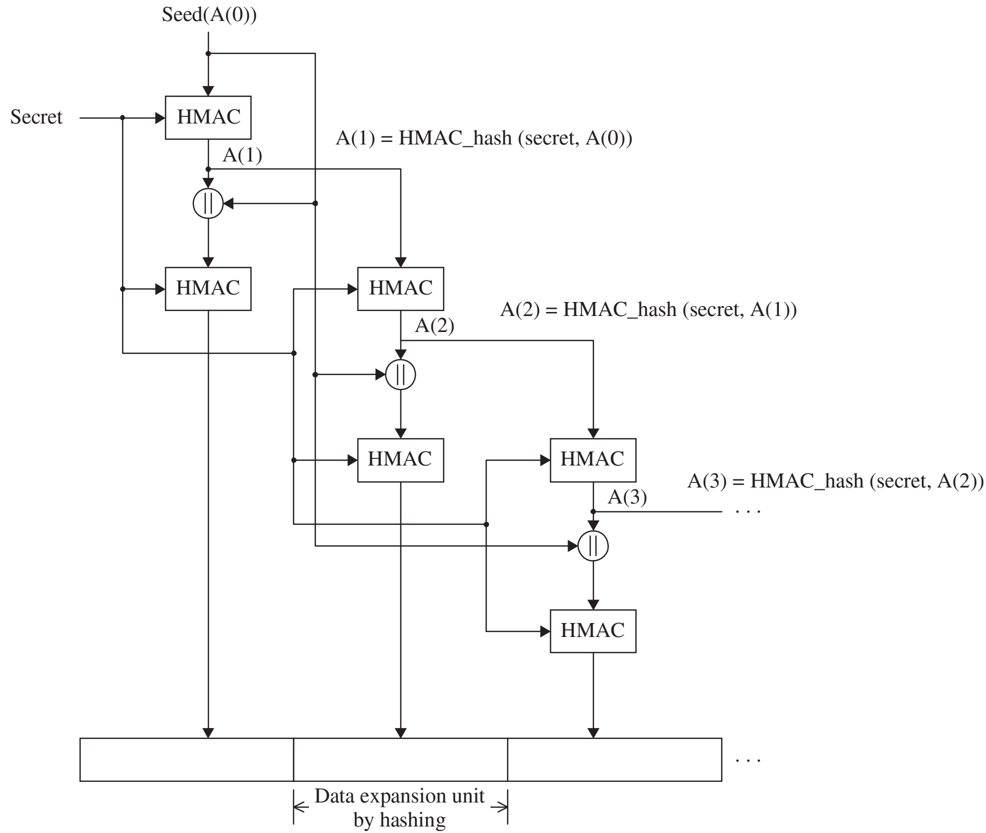



**Figure 9.10** TLS data expansion mechanism using P_hash(secret,seed).

***Example 9.3*** Refer to Figure 9.11. Suppose the following parameters are given:

seed = 0x 80 af 12 5c 7e 36 f3 21

label = rocky mountains = 0x 72 6f 63 6b 79 20 6d 6f 75 6e 74 61 69 6e 73

secret = 0x 35 79 af 12 c4

Then

label‖seed = 0x 72 6f 63 6b 79 20 6d 6f 75 6e 74 61 69 6e 73 80 af 12 5c 7e 36 f3 21

= A(0)

S1 = 0x 35 79 af for P_MD5, S2 = 0x af 12 c4 for P_SHA − 1

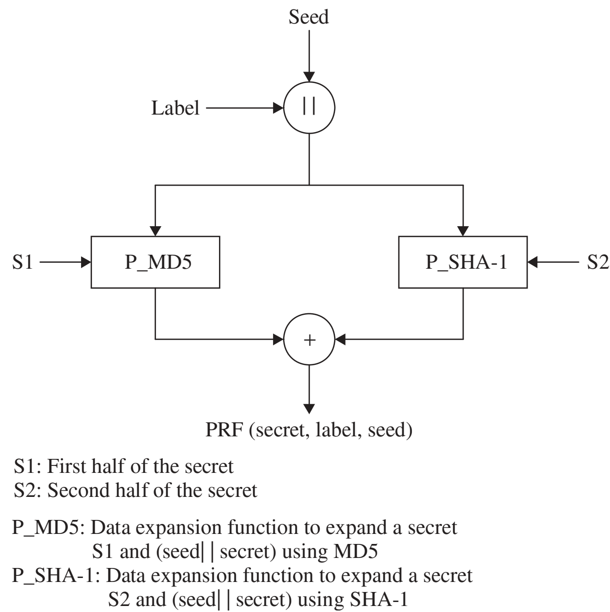



**Figure 9.11** A pseudorandom function (PRF) generation scheme.

*Data expansion by P_MD5*

A(1) = HMAC_MD5(S1, A(0))

= d0 de 36 53 79 78 04 a0 21 b8 6f f8 29 60 d5 f7

HMAC_MD5(S1, A(1)‖A(0))

= 32 fd b3 70 eb 36 11 70 a4 3b 50 a9 fb ea 2a ec

A(2) = HMAC_MD5(S1, A(1))

= 8c ce 5b 50 02 af 75 91 e7 20 cd 86 d9 3e 67 9d

HMAC_MD5(S1, A(2)‖A(0))

= 1f a8 4c af 5d e1 20 01 ea b0 38 6a a5 76 f9 8e

A(3) = HMAC_MD5(S1, A(2))

= 45 48 5d 00 4e 64 07 45 eb 2c 18 60 7c e6 fa 1f

HMAC_MD5(S1, A(3)‖A(0))

= f0 23 29 d9 5e 89 4b 70 cc 45 f8 aa 1f 58 8e 55

A(4) = HMAC_MD5(S1, A(3))

= 87 39 c6 d3 7a b f8 e3 29 79 3a ae 63 24 6a ff

HMAC_MD5(S1, A(4)‖A(0))

= 2e 0c 27 26 d0 b4 78 85 09 a2 69 1c 1b 1b d7 8d

A(5) = HMAC_MD5(S1, A(4))

= 3a 2c aa d8 b3 ec 2e 5d 40 1c 39 bd 3e 48 1a d9

HMAC_MD5(S1, A(5)‖A(0))

= 92 f2 63 5d 88 3a dd bf 8d ec e1 cf 0c 5c 8f 4c

where S1 = 0x 35 79 af = first half of the secret, and

A(0) = label‖seed

Thus, P_MD5 equals:

```
32  fd  b3  70  eb  36  11  70  a4  3b  50  a9  fb  ea  2a  ec
1f  a8  4c  af  5d  e1  20  01  ea  b0  38  6a  a5  76  f9  8e
f0  23  29  d9  5e  89  4b  70  cc  45  f8  aa  1f  58  8e  55
2e  0c  27  26  d0  b4  78  85  09  a2  69  1c  1b  1b  d7  8d
92  f2  63  5d  88  3a  dd  bf  8d  ec  e1  cf  0c  5c  8f  4c  (80 bytes)
```

*Data expansion by P_SHA-1*

A(1) = HMAC_SHA1(S2, A(0))

= aa ea 46 1b a6 ad 43 34 51 f8 c6 ef 70 dd f4 60 ca b9 40 2f

HMAC_SHA1(S2, A(1)‖A(0))

= d0 8a d5 07 e0 b8 30 78 70 d9 c8 bb dd ba f5 a3 d0 77 49 e8

A(2) = HMAC_SHA1(S2, A(1))

= 33 fd 23 41 01 ce 06 f8 c0 2b b3 e6 54 21 1c f4 6c 88 ab da

HMAC_SHA1(S2, A(2)‖A(0))

= 64 b5 cc 3f 79 31 5b 5d e6 e4 4f eb 98 a8 bf 3f 97 13 38 e1

A(3) = HMAC_SHA1(S2, A(2))

= 86 1f a3 a5 37 58 41 71 f1 9f a5 f3 48 2e 5d 84 7c a8 b6 52

HMAC_SHA1(S2, A(3)‖A(0))

= 03 26 11 02 ce 69 74 4a 21 f4 76 55 13 af 77 80 2d fb 2f 36

A(4) = HMAC_SHA1(S2, A(3))

= 9c 4d 01 3a 8c 48 54 42 68 07 4d f1 f0 a9 78 c3 6f ab d8 b4

HMAC_SHA1(S2, A(4)‖A(0))

= 48 56 04 b5 b4 5f 9b d8 c7 2f 28 f6 9e 1d 8a c4 72 9a b9 32

where S2 = 0x af 12 c4 = second half of the secret, and

A(0) = label‖seed

Thus, P_SHA1 equals:

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| d0 | 8a | d5 | 07 | e0 | b8 | 30 | 78 | 70 | d9 | c8 | bb | dd | ba | f5 | a3 | |
| d0 | 77 | 49 | e8 | 64 | b5 | cc | 3f | 79 | 31 | 5b | 5d | e6 | e4 | 4f | eb | |
| 98 | a8 | bf | 3f | 97 | 13 | 38 | e1 | 03 | 26 | 11 | 02 | ce | 69 | 74 | 4a | |
| 21 | f4 | 76 | 55 | 13 | af | 77 | 80 | 2d | fb | 2f | 36 | 48 | 56 | 04 | b5 | |
| b4 | 5f | 9b | d8 | c7 | 2f | 28 | f6 | 9e | 1d | 8a | c4 | 72 | 9a | b9 | 32 | (80 bytes) |

Finally, P_MD5 ⊕ P_SHA − 1 equals:

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| e2 | 77 | 66 | 77 | 0b | 8e | 21 | 08 | d4 | e2 | 98 | 12 | 26 | 50 | df | 4f | |
| cf | df | 05 | 47 | 39 | 54 | ec | 3e | 93 | 81 | 63 | 37 | 43 | 92 | b6 | 65 | |
| 68 | 8b | 96 | e6 | c9 | 9a | 73 | 91 | cf | 63 | e9 | a8 | d1 | 31 | fa | 1f | |
| 0f | f8 | 51 | 73 | c3 | 1b | 0f | 05 | 24 | 59 | 46 | 2a | 53 | 4d | d3 | 38 | |
| 26 | ad | f8 | 85 | 4f | 15 | f5 | 49 | 13 | f1 | 6b | 0b | 7e | c6 | 36 | 7e | (80 bytes) |

### 9.3.3 Error Alerts

The Alert Protocol is classified into the closure alert and the error alert. One of the content types supported by the TLS Record Layer is the alert type. Alert messages convey the severity of the message and a description of the alert. Alert messages with a fatal level result in the immediate termination of the connection.

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack. Either party may initiate a close by sending a close_notify alert. This message notifies the recipient that the sender will not send any more messages on this connection.

Error handling in the TLS Handshake Protocol is very simple. When an error is detected, the detecting party sends a message to the other party. Upon transmission or receipt of a fatal alert message, both parties immediately close the connection.

TLS supports all of the error alerts defined in SSLv3 with the exception of additional alert codes defined in TLS. The additional error alerts are described in the following.

- *decryption_failed.* A TLS ciphertext is decrypted in an invalid way: either it was not an even multiple of the block length or its padding values, when checked, were incorrect. This message is always fatal.

- *record_overflow*. A TLS record was received with a ciphertext whose length exceeds $2^{14} + 2048$ bytes, or the ciphertext decrypted to a TLS compressed record with more than $2^{14} + 1024$ bytes. This message is always fatal.
- *unknown_ca*. A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known, trusted CA. This message is always fatal.
- *access_denied*. A valid certificate was received, but when access control was applied, the sender decided not to proceed with the negotiation. This message is always fatal.
- *decode_error*. A message could not be decoded because a field was out of its specified range or the length of the message was incorrect. This message was incorrect. It is always fatal.
- *decrypt_error*. A handshake cryptographic operation failed, including being unable to verify a signature, decrypt a key exchange, or validate a finished message.
- *export_restriction*. A negotiation not in compliance with export restrictions was detected; for example, attempting to transfer a 1024-bit ephemeral RSA key for the RSA_EXPORT handshake method. This message is always fatal.
- *protocol_version*. The protocol version the client has attempted to negotiate is recognized but not supported due to the fact that old protocol versions might be avoided for security reasons. This message is always fatal.
- *insufficient_security*. Returned instead of hanshake_failure when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client. This message is always fatal.
- *internal_error*. An internal error unrelated to the peer or the correctness of the protocol, such as a memory allocation failure, makes it impossible to continue. This message is always fatal.
- *user_canceled*. This handshake is being cancelled for some reason unrelated to a protocol failure. If the user cancels an operation after the handshake is complete, just closing the connection by sending a close_notify is more appropriate. This alert should be followed by a close_notify. This message is generally a warning.
- *no_renegotiation*. This is sent by the client in response to a hello request or by the server in response to a client hello after initial handshaking. Either of these messages would normally lead to renegotiation, but this alert indicates that the sender is not able to renegotiate. This message is always a warning.

For all errors where an alert level is not explicitly specified, the sending party may determine at its discretion whether this is a fatal error or not; if an alert with a level of warning is received, the receiving party may decide at its discretion whether to treat this as a fatal error or not. However, all messages which are transmitted with a level of fatal must be treated as fatal messages.

### 9.3.4 Certificate Verify Message

Recall that the hash computations for SSLv3 are included with the master secret, the handshake message, and pads. In the TLS certificate verify message, the MD5 and SHA-1 hashes are calculated only over handshake messages as shown below.

```
CertificateVerify.signature.md5_hash
       MD5(handshake_message)
CertificateVerify.signature.sha_hash
       SHA(handshake_message)
```

Here handshake messages refer to all handshake messages sent or received starting at client hello up to, but not including, this message, including the type and length fields of the handshake messages.

### 9.3.5 Finished Message

A finished message is always sent immediately after a change cipher spec message to verify that the key exchange and authentication processes were successful. It is essential that a change cipher spec message be received between the other handshake messages and the finished message. As with the finished message in SSLv3, the finished message in TLS is a hash based on the shared master_secret, the previous handshake messages, and a label that identifies client and server. However, the TLS computation for verify_data is somewhat different from that of the SSL calculation as shown below.

```
PRF(master_secret, finished_label, MD5(handshake_message)||
      SHA-1(handshake_message))
```

where

- The finished_label indicates either the string “client finished” sent by the client or the string “server finished” sent by the server, respectively.
- The handshake_message includes all handshake messages starting at client hello up to, but not including, this finished message. This is only visible at the handshake layer and does not include record layer headers. In fact, this is the concatenation of all the handshake structures exchanged thus far. This may be different from handshake messages for SSL because it would include the certificate verify message. Also, the handshake_message for the finished message sent by the client will be different from that for the finished message sent by the server.

Note that change cipher spec messages, alters and any other record types are not handshake messages and are not included in the hash computations.

### 9.3.6 Cryptographic Computations (for TLS)

In order to begin connection protection, the TLS Record Protocol requires specification of a suite of algorithms, a master secret, and the client and server random values. The authentication, encryption, and MAC algorithms are determined by the cipher_suite selected by the server and revealed in the server hello message. The compression algorithm is negotiated in the hello messages, and the random values are exchanged in the hello messages.

All that remains is to compute the master secret and the key block. The premaster_secret for TLS is calculated in the same way as in SSLv3. The presmater_secret should be deleted from memory once the master_secret has been computed. As in SSLv3, the master_secret in TLS in calculated as a hash function of the premaster_secret and two hello random numbers. The TLS master_secret computation is different from that of SSLv3 and is defined as follows:

```
master_secret = PRF(premaster_secret, ‘‘master secret’’,
                   ClientHello.random||ServerHello.random)
```

The master_secret is always exactly 48 bytes (384 bits) in length. The length of the premaster_secret will vary depending on key exchange method.

- *RSA*. When RSA is used for server authentication and key exchange, a 48-byte premaster_secret is generated by the client, encrypted with the server’s public key, and sent to the server. The server uses its private key to decrypt the premaster_secret. Both parties then convert the premaster_secret into the master_secret, as specified above.
- *DH*. A conventional DH computation is performed. The negotiated key Z is used as the premaster_secret, and is converted into the master_secret, as specified above.

The computation of the key block parameters (MAC secret keys, session encryption keys, and IVs) is defined as follows:

```
key_block = PRF(master_secret, ‘‘key expansion’’,
            SecurityParameters.server_random||
                     SecurityParameters.client_random)
```

This is computed until sufficient output has been generated. As with SSLv3, key_block is a function of the master_secret and the client and the server random numbers, but for TLS, the actual algorithm is different.

On leaving this chapter, it is recommended that readers search for and find any other small differences between SSL and TLS.