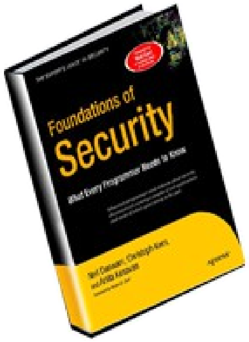
CHAPTER 12

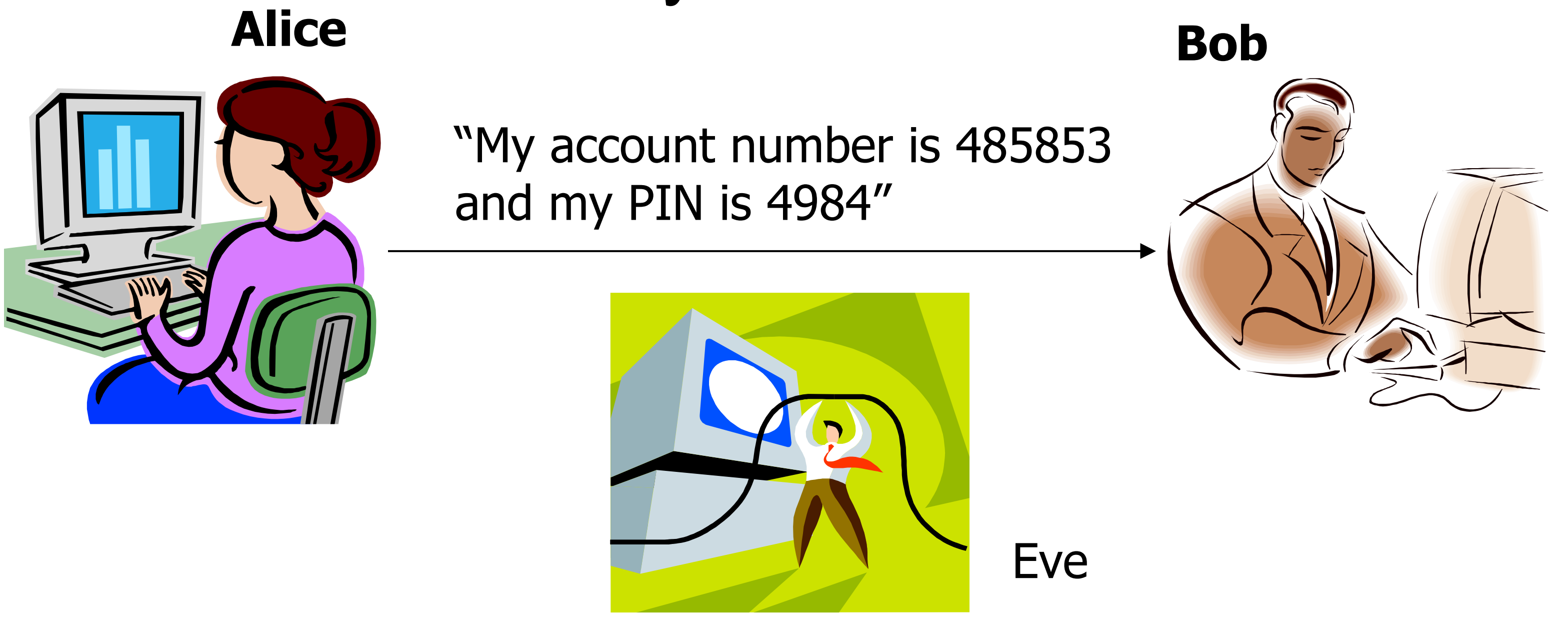
Symmetric Key Cryptography

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; http://www.foundationsofsecurity.com).



12.1. Introduction to Cryptography

- Goal: Confidentiality



Alice

Bob

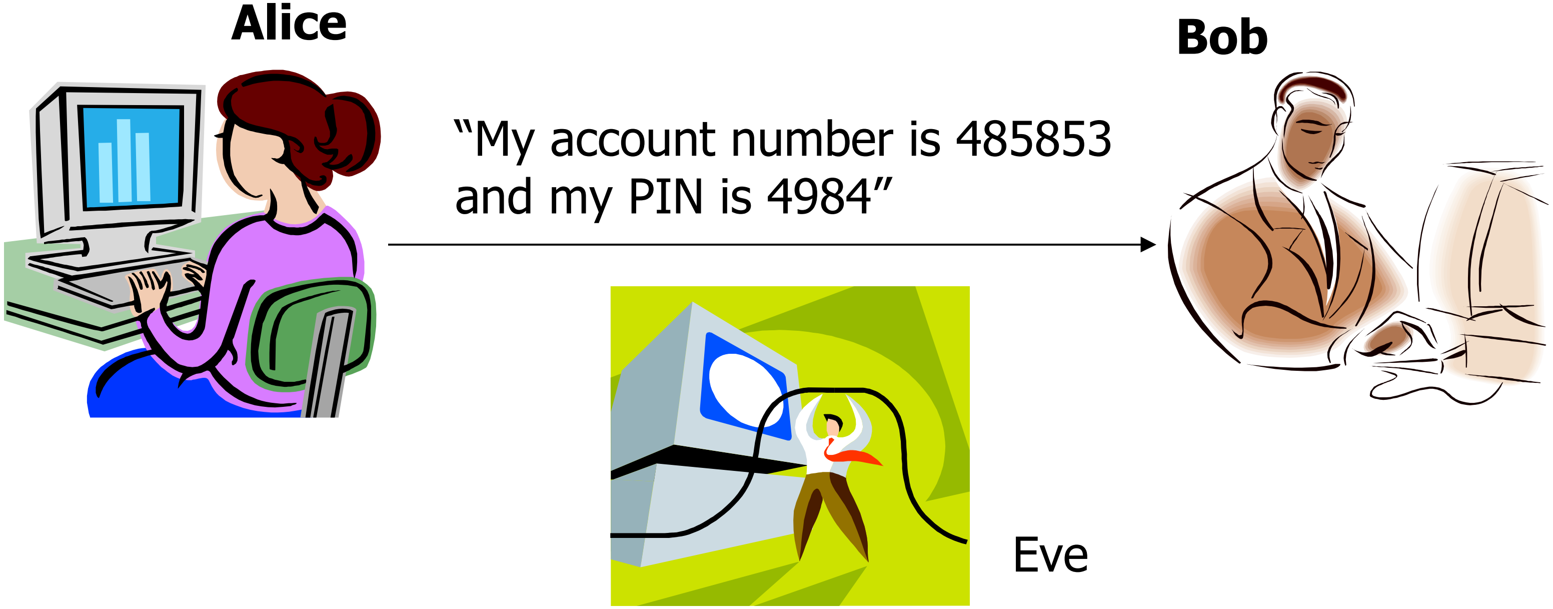
"My account number is 485853 and my PIN is 4984"

Eve

- Message "sent in clear": Eve can overhear
- Encryption unintelligible to Eve; only Bob can decipher with his secret key (shared w/ Alice)

12.1. Introduction to Cryptography

- Goal: Confidentiality



Alice

“My account number is 485853 and my PIN is 4984”

Bob

Eve

- Message “sent in clear”: Eve can overhear
- Encryption unintelligible to Eve; only Bob can decipher with his secret key (shared w/ Alice)

12.1.1. Substitution Ciphers

- Plaintext: **`meet me at central park`**
- Ciphertext: **`phhw ph dw fhqwudo sdun`**

- Plain: `abcdefghijklmnopqrstuvwxyz`
- Cipher: `defghijklmnopqrstuvwxyzabc`

- Key is 3, i.e. shift letter right by 3
- Easy to break due to frequency of letters
- Good encryption algorithm produces output that looks random: equal probability any bit is 0 or 1

12.1.2. Notation & Terminology

- m = message (plaintext), c = ciphertext
- F = encryption function
- F^{-1} = decryption function

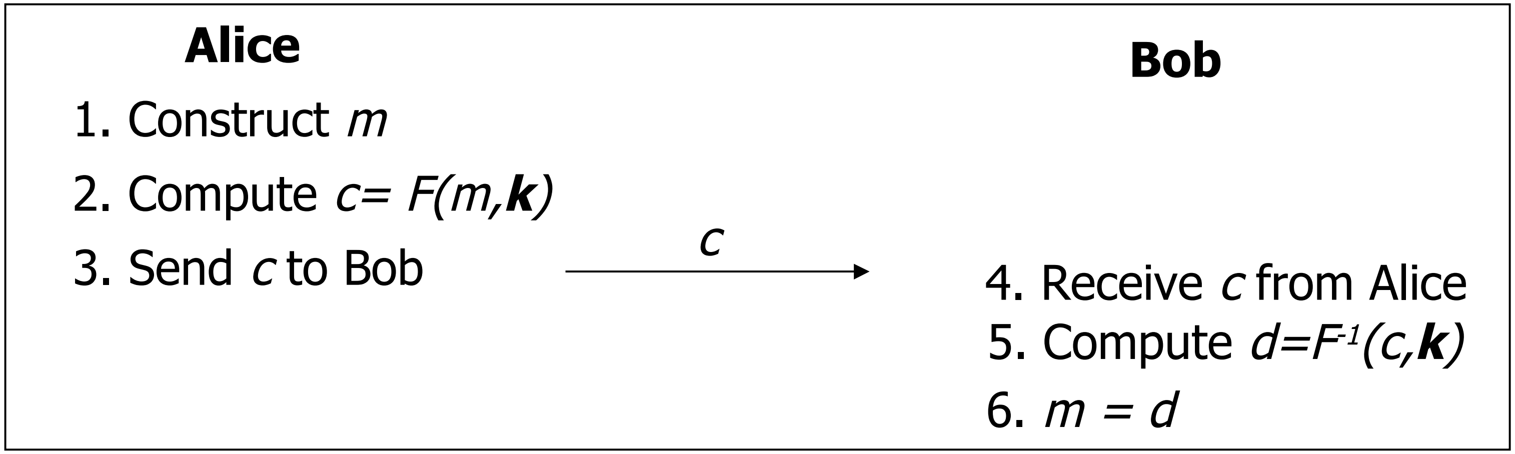
Cipher

- k = key (secret number)

- $c = F(m,k) = F_k(m)$ = encrypted message
- $m = F^{-1}(c,k) = F^{-1}{}_k(c)$ = decrypted message
- Symmetric cipher: $F^{-1}(F(m,k), k) = m$, same key

Symmetric Encryption

- Alice encrypts a message with the ***same*** key that Bob uses to decrypt.



Alice

1. Construct *m*
2. Compute $c = F(m,\mathbf{k})$
3. Send *c* to Bob

$\xrightarrow{c}$

Bob

4. Receive *c* from Alice
5. Compute $d=F^{-1}(c,\mathbf{k})$
6. $m = d$

- Eve can see *c*, but cannot compute *m* because *k* is only known to Alice and Bob

12.1.3. Block Ciphers

- Blocks of bits (e.g. 256) encrypted at a time

- Examples of several algorithms:
  - Data Encryption Standard (DES)
  - Triple DES
  - Advanced Encryption Standard (AES) or Rijndael

- Internal Data Encryption Algorithm (IDEA), Blowfish, Skipjack, many more… (c.f. Schneier)

12.1.3. DES

- Adopted in 1977 by NIST

- Input: 64-bit plaintext, 56-bit key (64 w/ parity)
- Parity Bits: redundancy to detect corrupted keys
- Output: 64-bit ciphertext
- Susceptible to Brute-Force (try all 2^{56} keys)
  - 1998: machine Deep Crack breaks it in 56 hours
  - Subsequently been able to break even faster
  - Key size should be at least 128 bits to be safe

12.1.3. Triple DES

- Do DES thrice w/ 3 different keys (slower)
- $c = F(F^{-1}(F(m\ ,k_1),k_2),k_3)$ where $F = DES$
  - Why decrypt with k_2?
  - Backwards compatible w/ DES, easy upgrade
- Keying Options: Key Size (w/ Parity)
  - $k_1 \neq k_2 \neq k_3$: 168-bit (192-bit)
  - $k_1 = k_3 \neq k_2$: 112-bit (128-bit)
  - $k_1 = k_2 = k_3$: 56-bit (64-bit) (DES)

12.1.3. AES (Rijndael)

- Invented by 2 Belgian cryptographers
- Selected by NIST from 15 competitors after three years of conferences vetting proposals
- Selection Criteria:
  - Security, Cost (Speed/Memory)
  - Implementation Considerations (Hardware/Software)
- Key size & Block size: 128, 192, or 256 bits (much larger than DES)
- Rely on algorithmic properties for security, not obscurity

12.1.4. Security by Obscurity: Recap

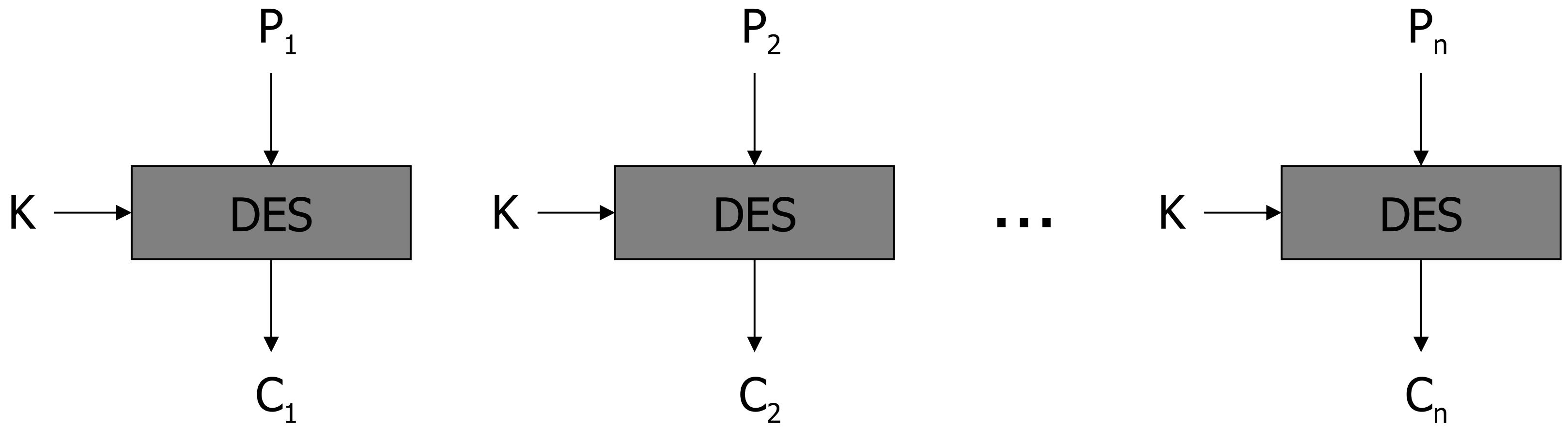
- Design of DES, Triple DES algorithms public
  - Security not dependent on secrecy of implementation
  - But rather on secrecy of key

- Benefits of Keys:
  - Easy to replace if compromised
  - Increasing size by one bit, doubles attacker's work

- If invent own algorithm, make it public! Rely on algorithmic properties (math), not obscurity.

12.1.5. Electronic Code Book

- Encrypting more data: ECB encrypt blocks of data in a large document



- Leaks info about structure of document (e.g. repeated plaintext blocks)

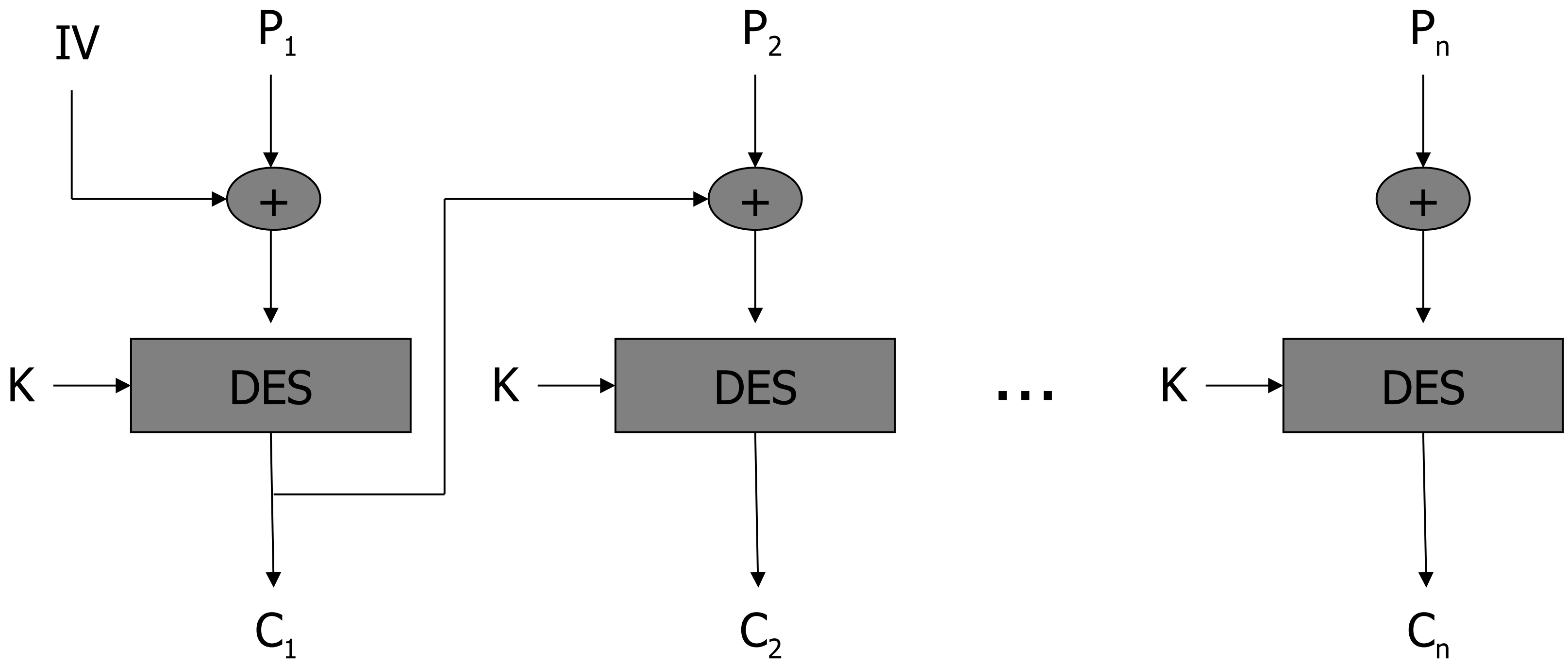
12.1.5. Review of XOR

- Exclusive OR (either x or y but not both)
- Special Properties:
  - □ x XOR y = z
  - □ z XOR y = x
  - □ x XOR z = y

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

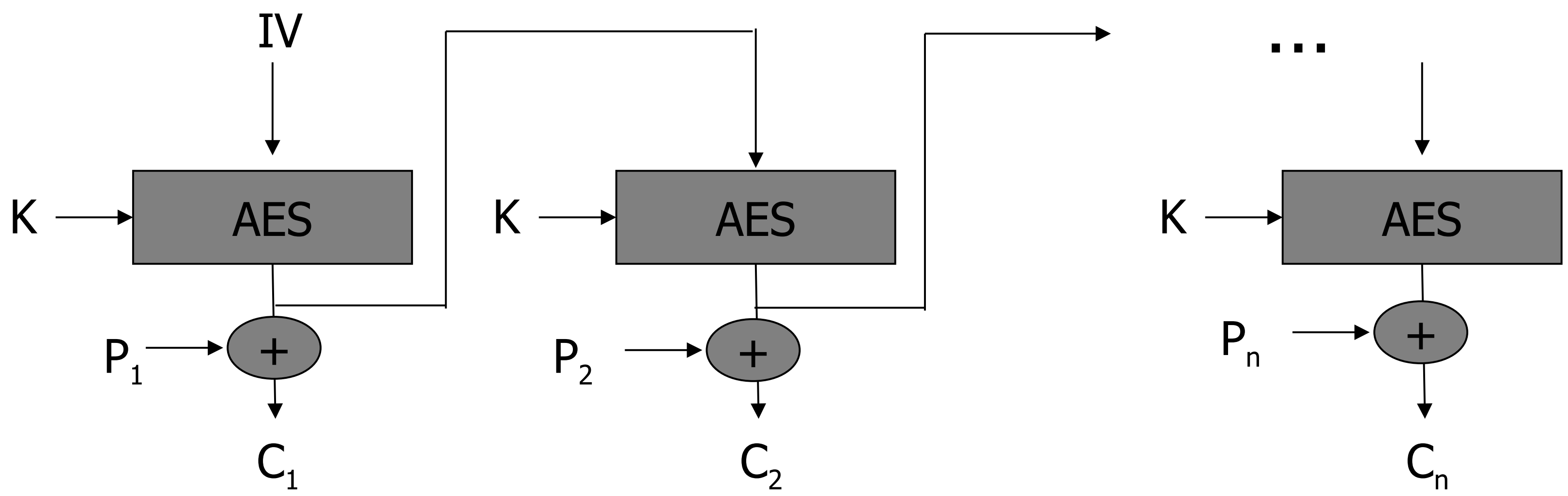
12.1.5. Cipher Block Chaining

- CBC: uses XOR, no patterns leaked!
- Each ciphertext block depends on prev block



12.1.5. Output Feedback (OFB)

- Makes block cipher into stream cipher
- Like CBC, but do XOR with plaintext after encryption



12.2. Stream Ciphers

- Much faster than block ciphers
- Encrypts one byte of plaintext at a time
- *Keystream*: infinite sequence (never reused) of random bits used as key
- Approximates theoretical scheme: one-time pad, trying to make it practical with finite keys

12.2.1 One-Time Pad

- Key as long as plaintext, random stream of bits
  - Ciphertext = Key XOR Plaintext
  - Only use key once!

- Impractical having key the same size as plaintext (too long, incurs too much overhead)

- Theoretical Significance: “perfect secrecy” (Shannon) if key is random.
  - Under brute-force, every decryption equally likely
  - Ciphertext yields no info about plaintext (attacker’s a priori belief state about plaintext is unchanged)

12.2.2. RC4

- Most popular stream cipher: 10x faster than DES

- Fixed-size key “seed” to generate infinite stream

- *State Table S* that changes to create stream
- 40/256-bit key used to seed table (fill it)

RC4 implementation

Table initialization

```
for i = 0 to 255
    S[i] = i
j = 0
for i = 0 to 255
    j = (j + S[i] + key[i mod kl]) mod 256
    swap (S[i], S[j])
```

- kl=keylength

Encrypt/decrypt

```
i = 0
j = 0
for l = 0 to len(input)
    i = (i + 1) mod 256
    j = (j + S[i]) mod 256
    swap (S[i], S[j])
    output[l] = S[(S[i] + S[j]) mod 256] XOR input[l]
```

12.2.2. RC4 Pitfalls

- Never use the same key more than once!

- Clients & servers should use different RC4 keys!
  - C → S: P XOR k [Eve captures P XOR k]
  - S → C: Q XOR k [Eve captures Q XOR k]
  - Eve: (P XOR k) XOR (Q XOR k) = P XOR Q!!!
  - If Eve knows either P or Q, can figure out the other

- Ex: Simple Mail Transfer Protocol (SMTP)
  - First string client sends server is `HELO`
  - Then Eve could decipher first few bytes of response

12.2.2. More RC4 Pitfalls

- Initial bytes of key stream are “weak”
  - Ex: WEP protocol in 802.11 wireless standard is broken because of this
  - Discard first 256-512 bytes of stream
- Active Eavesdropper
  - Could flip bit without detection
  - Can solve by including MAC to protect integrity of ciphertext

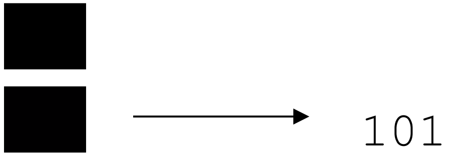
12.3. Steganography

- All ciphers transform plaintext to random bits
- Eve can tell Alice is sending sensitive info to Bob
- Conceal existence of secret message
- Use of a “covert channel” to send a message.

12.3.1. What is Steganography?

- Study of techniques to send sensitive info and hide the fact that sensitive info is being sent
- Ex: “All the tools are carefully kept” -> Attack
- Other Examples: Invisible ink, Hidden in Images
  - □ Least significant bit of image pixels
  - □ Modifications to image not noticeable by an observer
  - □ Recipient can check for modifications to get message

Red	Green	Blue
00000000	00000000	00000000
00000001	00000000	00000001



12.3.2. Steganography vs. Cryptography

- Key Advantage: when Alice & Bob don’t want Eve to know that they’re communicating secrets

- Traffic Analysis can return useful information

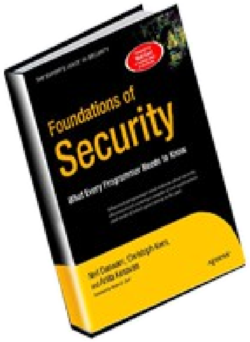
- Disadvantages compared to encryption
  - □ Essentially relying on security by obscurity
  - □ Useless once covert channel is discovered
  - □ High overhead (ratio of plain bits/secret bits high)

- Can be used together with encryption, but even more overhead (additional computation for both)

CHAPTER 13

Asymmetric Key Cryptography

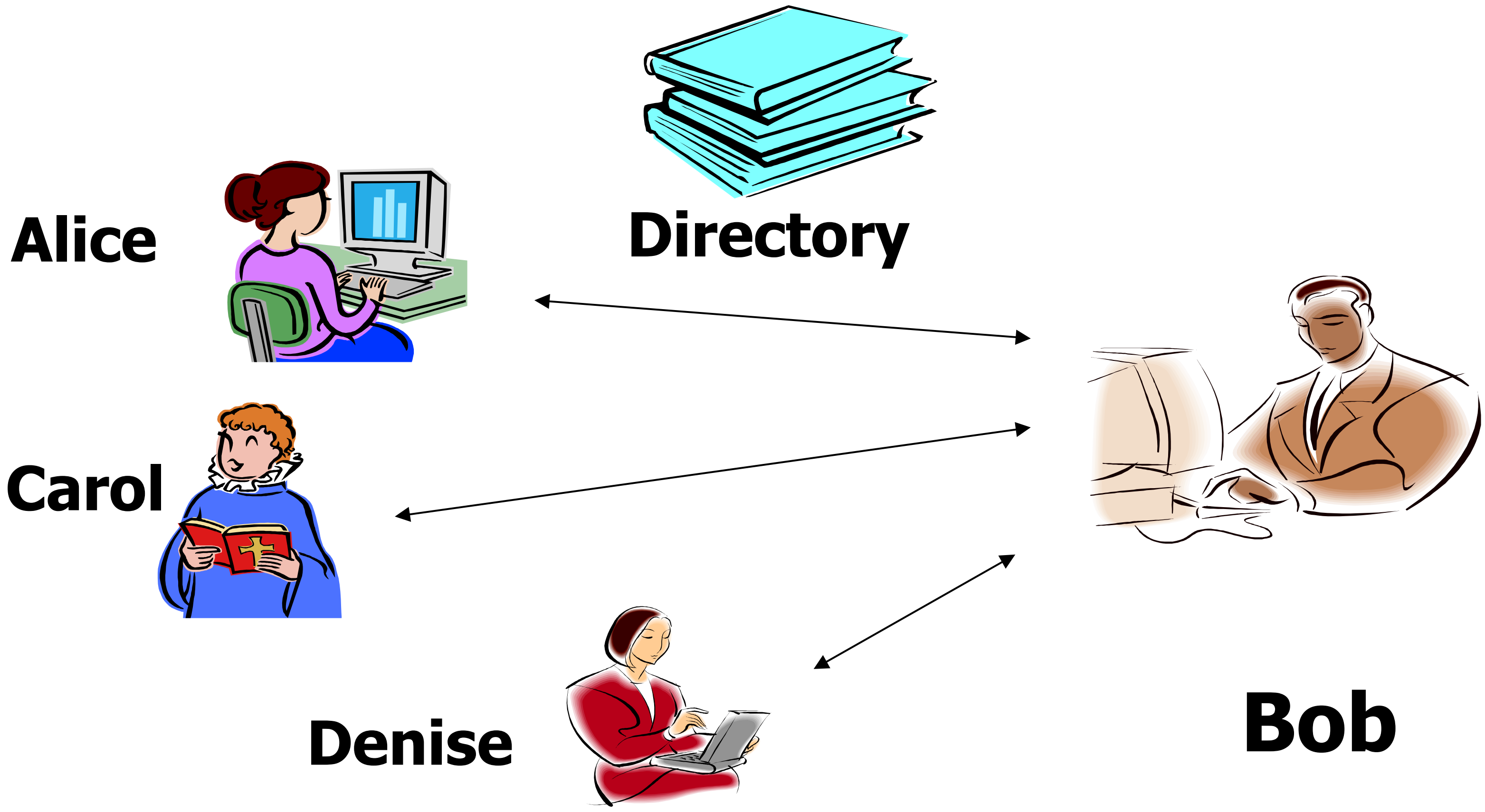
Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; http://www.foundationsofsecurity.com). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.



13.1. Why Asymmetric Key Cryptography?

- So two strangers can talk privately on Internet
- Ex: Bob wants to talk to Alice & Carol secretly
  - Instead of sharing different pairs of secret keys with each (as in symmetric key crypto)
  - Bob has 2 keys: *public* key and *private* (or secret) key
- Alice and Carol can send secrets to Bob encrypted with his public key
- Only Bob (with his secret key) can read them

13.1. Public Key System



Directory

Alice

Carol

Denise

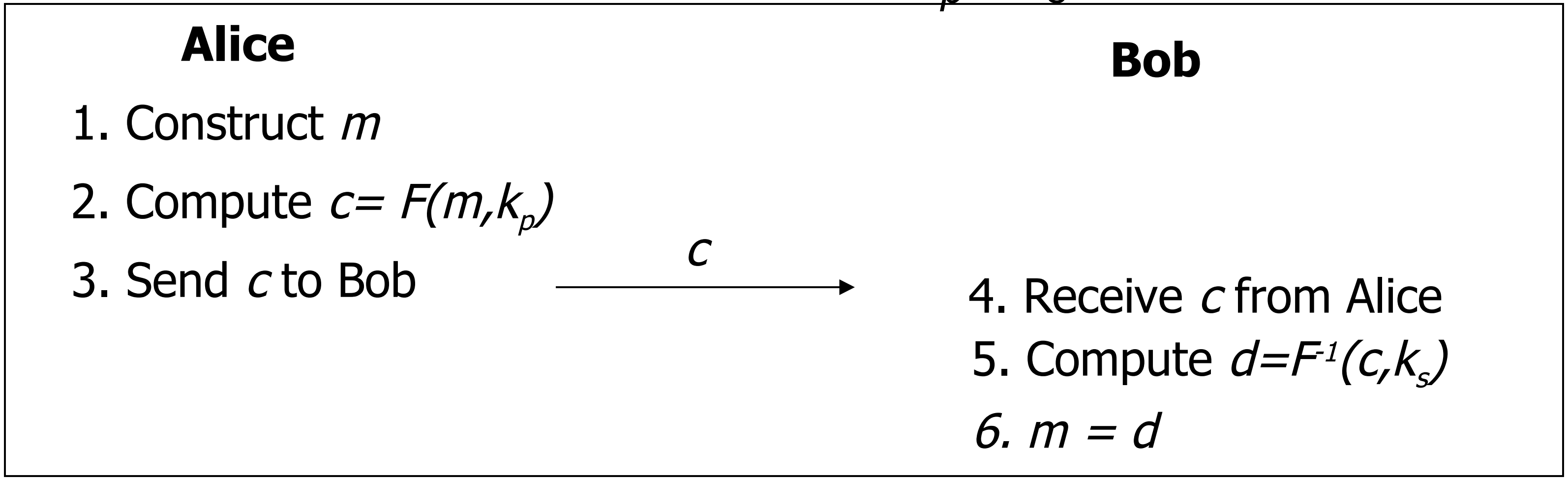
Bob

13.1. The Public Key Treasure Chest

- Public key = Chest with open lock
- Private key = Key to chest
- Treasure = Message
- Encrypting with public key
  - Find chest with open lock
  - Put a message in it
  - Lock the chest
- Decrypting with private key
  - Unlock lock with key
  - Take contents out of the chest

13.1. Asymmetric Encryption

- Alice encrypts a message with ***different*** key than Bob uses to decrypt
- Bob has a public key, k_p, and a secret key, k_s. Bob's public key is known to Alice.
- Asymmetric Cipher: $F^{-1}(F(m,k_p),k_s) = m$



Alice

1. Construct m

2. Compute $c= F(m,k_p)$

3. Send c to Bob $\xrightarrow{c}$

Bob

4. Receive c from Alice

5. Compute $d=F^{-1}(c,k_s)$

6. m = d

13.2. RSA (1)

- Invented by Rivest/Shamir/Adelman (1978)
  - First asymmetric encryption algorithm
  - Most widely known public key cryptosystem
- Used in many protocols (e.g., SSL, PGP, …)

- Number theoretic algorithm: security based on difficulty of factoring ***large*** prime numbers
- 1024, 2048, 4096-bit keys common

13.2. RSA (2)

- Public Key Parameters:
  - Large composite number n with two prime factors
  - Encryption exponent e coprime (no common factor) to $\phi(n) = (p-1)(q-1)$
- Private Key:
  - Factors of n: p, q ($n = pq$)
  - Decryption exponent d such that ed *1 (mod $\phi(n)$)

- Encryption: Alice sends $c = m^e \mod n$
- Decryption: Bob computes $m = c^d \mod n$

Key generation and proof :-)

- Choose two distinct prime p and q and compute n = pq.
- n is used as the modulus for both the public and private keys
- Compute φ(n) = (p – 1)(q – 1), where φ is Euler's totient function.
- Choose an integer e where 1 < e < φ(n) and gcd of (e, φ(n)) = 1
- e is released as the public key exponent.
- Determine d = e–1 mod φ(n); i.e., d is the multiplicative inverse of e mod φ(n) = solve for d given (de) mod φ(n) = 1.
- d is kept as the private key exponent.

$$m^{ed} = m^{(ed-1)}m = m^{h(p-1)(q-1)}m = \left(m^{p-1}\right)^{h(q-1)}m \equiv 1^{h(q-1)}m \equiv m \mathrm{mod} p,$$

$$a^{(p-1)} \equiv 1 \pmod{p}.$$

Fermat little theorem

13.3. Elliptic CurveCryptography

- Invented by N. Koblitz & V. Miller (1985)
- Based on hardness of elliptic curve discrete log problem
- Standardized by NIST, ANSI, IEEE for government, financial use
- Certicom, Inc. currently holds patent
- Small keys: 163 bits (<< 1024-bit RSA keys)

13.3: RSA vs. ECC

- RSA Advantages:
  - Has been around longer; math well-understood
  - Patent expired; royalty free
  - Faster encryption
- ECC Advantages:
  - Shorter key size
  - Fast key generation (no primality testing)
  - Faster decryption

13.4. Symmetric vs. Asymmetric Key Cryptography

- Symmetric-Crypto (DES, 3DES, AES)
    - Efficient (smaller keys / faster encryption) because of simpler operations (e.g. discrete log)
    - Key agreement problem
    - Online
- Asymmetric-Crypto (RSA, ECC)
    - RSA 1000x slower than DES, more complicated operations (e.g. modular exponentiation)
    - How to publish public keys? Requires PKI / CAs
    - Offline or Online

13.5. Certificate Authorities

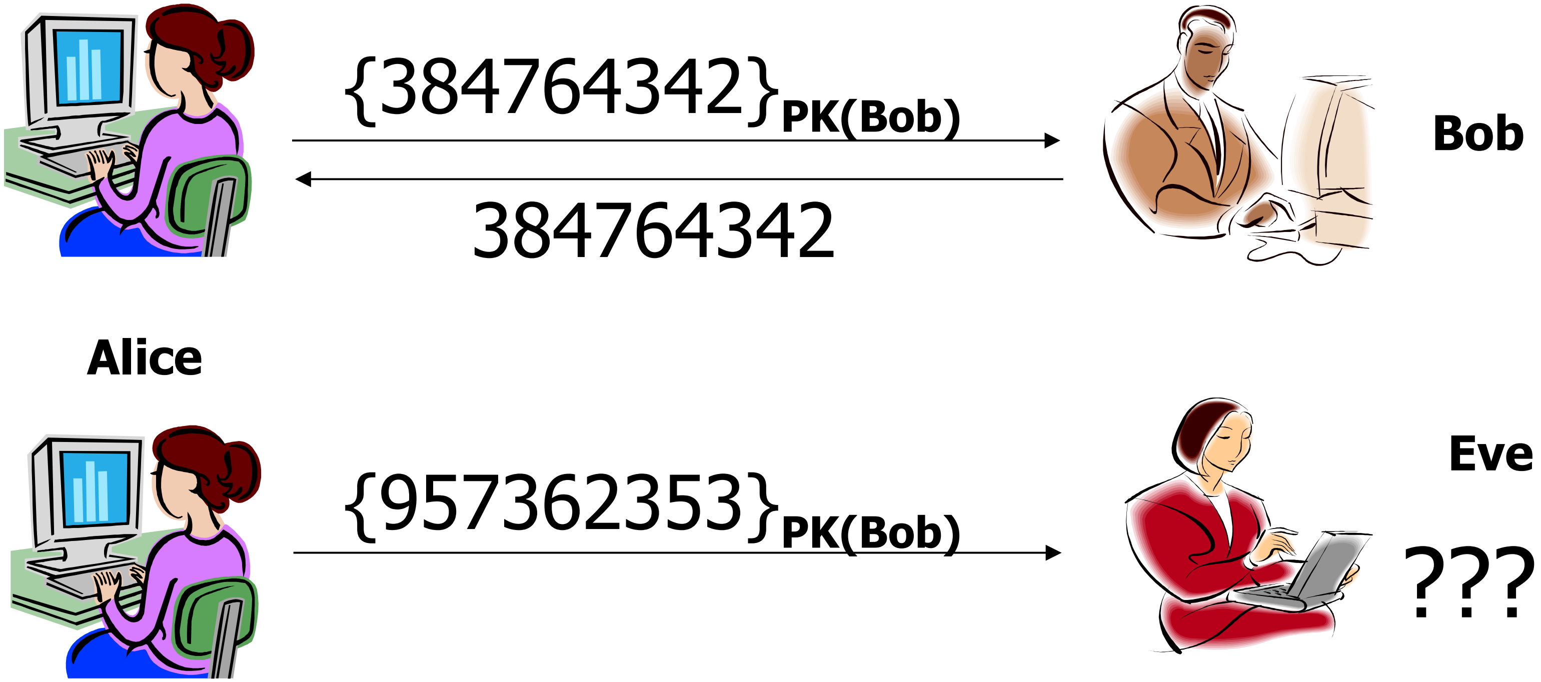
- Trusted third party: CA verifies people's identities
- Authenticates Bob & creates *public key certificate* (binds Bob's identity to his public key)

- CA also revokes keys and certificates
- Certificate Revocation List: compromised keys

- Public Key Infrastructure (PKI): CA + everything required for public key encryption

13.7. Challenge – Response with Encryption

- Alice issues “challenge” message to person
  - Random # (nonce) encrypted with Bob’s public key
  - If person is actually Bob, he will be able to decrypt it



$\{384764342\}_{PK(Bob)}$

Bob

384764342

Alice

$\{957362353\}_{PK(Bob)}$

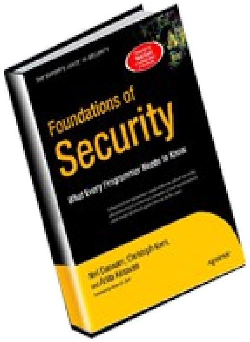
Eve

???

CHAPTER 14

Key Management & Exchange

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; http://www.foundationsofsecurity.com). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.



14.1. Types of Keys

- *Encryption* keys can be used to accomplish different security goals
- Identity Keys
- Conversation or Session Keys
- Integrity Keys
- One Key, One Purpose: Don’t reuse keys!

14.1.1. Identity Keys

- Used to help carry out authentication
- Authentication once per connection between two parties
- Generated by principal, long-lifetime (more bits)
- Bound to identity with certificate (e.g. public keys in asymmetric system)

14.1.2. Conversation or Session Keys

- Helps achieve confidentiality
- Used after 2 parties have authenticated themselves to each other
- Generated by key exchange protocol (e.g. Diffie-Hellman algorithm)
- Short-lifetime (fewer bits)

14.1.3. Integrity Keys

- Key used to compute Message Authentication Codes (MACs)

- Alice and Bob share integrity key
  - Can use to compute MACs on message
  - Detect if Eve tampered with message

- Integrity keys used in digital signatures

14.2. Key Generation

- Key generated through algorithms (e.g. RSA)
  - Usually involves random # generation as a step
  - But for Identity Based Encryption, master key
- Avoid weak keys (e.g. in DES keys of all 1s or 0s, encrypting twice decrypts)
- Don’t want keys stolen: After generation
  - Don’t store on disk connected to network
  - Also eliminate from memory (avoid *core dump* attack)
- Generating keys from passwords: Use password-based encryption systems to guard against dictionary attacks

14.2.1. Random Number Generation

- Ex: Alice & Bob use RSA to exchange a secret key for symmetric crypto (faster)
  - Alice generates random # *k*
  - Sends to Bob, encrypted with his public key
  - Then use *k* as key for symmetric cipher

- But if attacker can guess *k*, no secrecy
- Active eavesdropper can even modify/inject data into their conversation
- Problem: Generating hard to guess random #s

14.2.2. The rand() function

- How about using `rand()` function in C?
  - Uses linear congruential generator
  - After some time, output repeats predictably

- Can infer seed based on few outputs of `rand()`
  - Allows attacker to figure out all past & future output values
  - No longer unpredictable

- Don’t use for security applications

14.2.3. Random Device Files

- Virtual devices that look like files: (e.g. on Linux)
    - Reading from file provides unpredictable random bits generated based on events from booting
    - `/dev/random` – blocks until random bits available
    - `/dev/urandom` – doesn’t block, returns what’s there

```
$ head -c 20 /dev/random > /tmp/bits # read 20 chars
$ uuencode --base64 /tmp/bits printbits # encode,
print
begin-base64 644 printbits
bj4Ig9V6AAaqH7jzvt9T60aogEo===== # random output
```

14.2.4. Random APIs

- Windows OS: `CryptGenKey()` – to securely generate keys

- Java: `SecureRandom` class in `java.security` package (c.f. `AESEncrypter` example, Ch. 12)
  - Underlying calls to OS (e.g. `CryptGenKey()` for Windows or reads from `/dev/random` for Linux)
  - No guarantees b/c cross-platform
  - But better than `java.util.Random`

14.3. Key (Secret) Storage

- Secret to store for later use
  - Cryptographic key (private)
  - Password or any info system's security depends on

- Recall Kerchoff's principle: security should depend not on secrecy of algorithm, but on secrecy of cryptographic keys

- Options for storing secrets?

The general principle

- Cryptography does not solve a problem but simplifies it
- We have encrypted a huge file with a small key
- How we protect the key?
- The file is protected provided that we can protect the key

14.3.1. Keys in Source Code

- Ex: Program storing a file on disk such that no other program can touch it Might use key to encrypt file: Where to store it?
- Maybe just embed in source code? Easy since you can use at runtime to decrypt.

- Can reverse-engineer binary to obtain the key (even if obfuscated) e.g. `strings` utility outputs sequence of printable chars in object code

14.3.1. Reverse-Engineering

```
/* vault program (from 6.1.2) */
int checkPassword() {
    char pass[16];
    bzero(pass, 16); // Initialize
    printf ("Enter password: ");
    gets(pass);
    if (strcmp(pass, "opensesame") == 0)
      return 1;
    else
      return 0;
}

void openVault() {
      // Opens the vault
}

main() {
    if (checkPassword()) {
        openVault();
        printf ("Vault opened!");
    }
}
```

Key Leaked!

```
# partial output of printable
# characters in object code
$ strings vault
C@@0@
$ @
Enter password:
opensesame
__main
_impure_ptr
calloc
cygwin_internal
dll_crt0__FP11per_process
free
gets
malloc
printf
realloc
strcmp
GetModuleHandleA
cygwin1.dll
KERNEL32.dll
```

14.3.2. Storing the Key in a File on Disk

- Alternative to storing in source code, could store in file on disk

- Attacker with read access could
  - Find files with high entropy (randomness)
  - These would be candidate files to contain keys

- C.f. “Playing Hide and Seek with Stored Keys” (Shamir and van Someren)

14.3.3. “Hard to Reach” Places

- Store in Windows Registry instead of file?
  - Part of OS that maintains config info
  - Not as easy for average user to open

- But `regedit` can allow attacker (or slightly above-average user) to read the registry
  - Also registry entries stored on disk
  - Attacker with full read access can read them

- Registry not the best place to store secrets

14.3.4. Storing Secrets in External Devices (1)

- Store secrets in device external to computer!
  - Key won't be compromised even if computer is
  - Few options: smart card, HSMs, PDAs, key disks

- Smart Card (contains tamper-resistant chip)
  - Limited CPU power, vulnerable to power attacks
  - Must rely on using untrusted PIN readers
  - Attacker observes power of circuits, computation times to extract bits of the key

14.3.4. Storing Secrets in External Devices (2)

- Hardware Security Module (HSM)
  - Device dedicated to storing crypto secrets
  - External device, add-on card, or separate machine
  - Higher CPU power, key never leaves HSM (generated and used there)

- PDA or Cell phone
  - No intermediate devices like PIN readers
  - More memory, faster computations
  - Can have security bugs of their own

14.3.4. Storing Secrets in External Devices (3)

- Key Disk
  - USB, non-volatile memory, 2nd authentication factor
  - No CPU, not tamper-resistant
  - No support for authentication
  - Ex: IronKey, secure encrypted flash drive
- External Devices & Keys
  - Allows key to be removed from host system
  - Problem: connected to compromised host
  - Advantage: if crypto operation done on device & key never leaves it, damage limited
  - Can attack only while connected, can’t steal key

14.4. Key Agreement and Exchange

- Keys have been generated and safely stored, now what?
  - If Alice & Bob both have it, can do symmetric crypto
  - Otherwise, have to agree on key

- How to create secure communication channel for exchange?

- Few Options
  - Use Asymmetric Keys
  - Diffie-Hellman (DH) Key Exchange

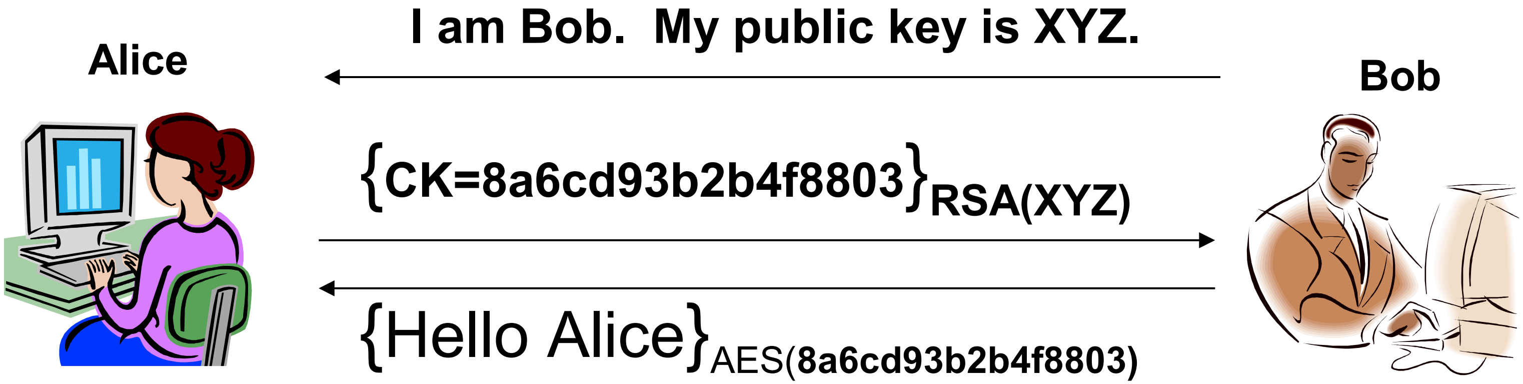
14.4.1. Using Asymmetric Keys

- Public-key crypto much more computationally expensive than symmetric key crypto

- Use RSA to send cryptographically random conversation key *k*

- Use *k* as key for faster symmetric ciphers (e.g. AES) for rest of conversation

14.4.1. Key Exchange Example



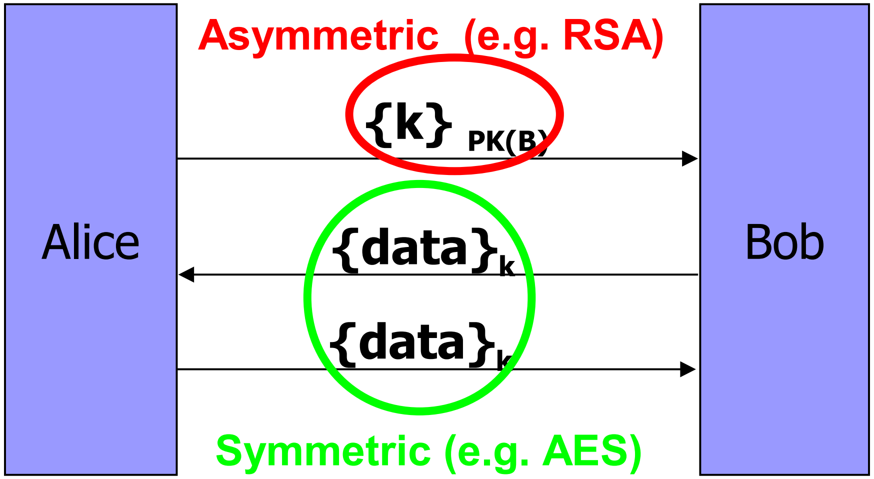
Alice

Bob

I am Bob. My public key is XYZ.

$\{CK=8a6cd93b2b4f8803\}_{RSA(XYZ)}$

$\{\text{Hello Alice}\}_{AES(8a6cd93b2b4f8803)}$



Alice

Bob

Asymmetric (e.g. RSA)

$\{k\}_{PK(B)}$

$\{data\}_k$

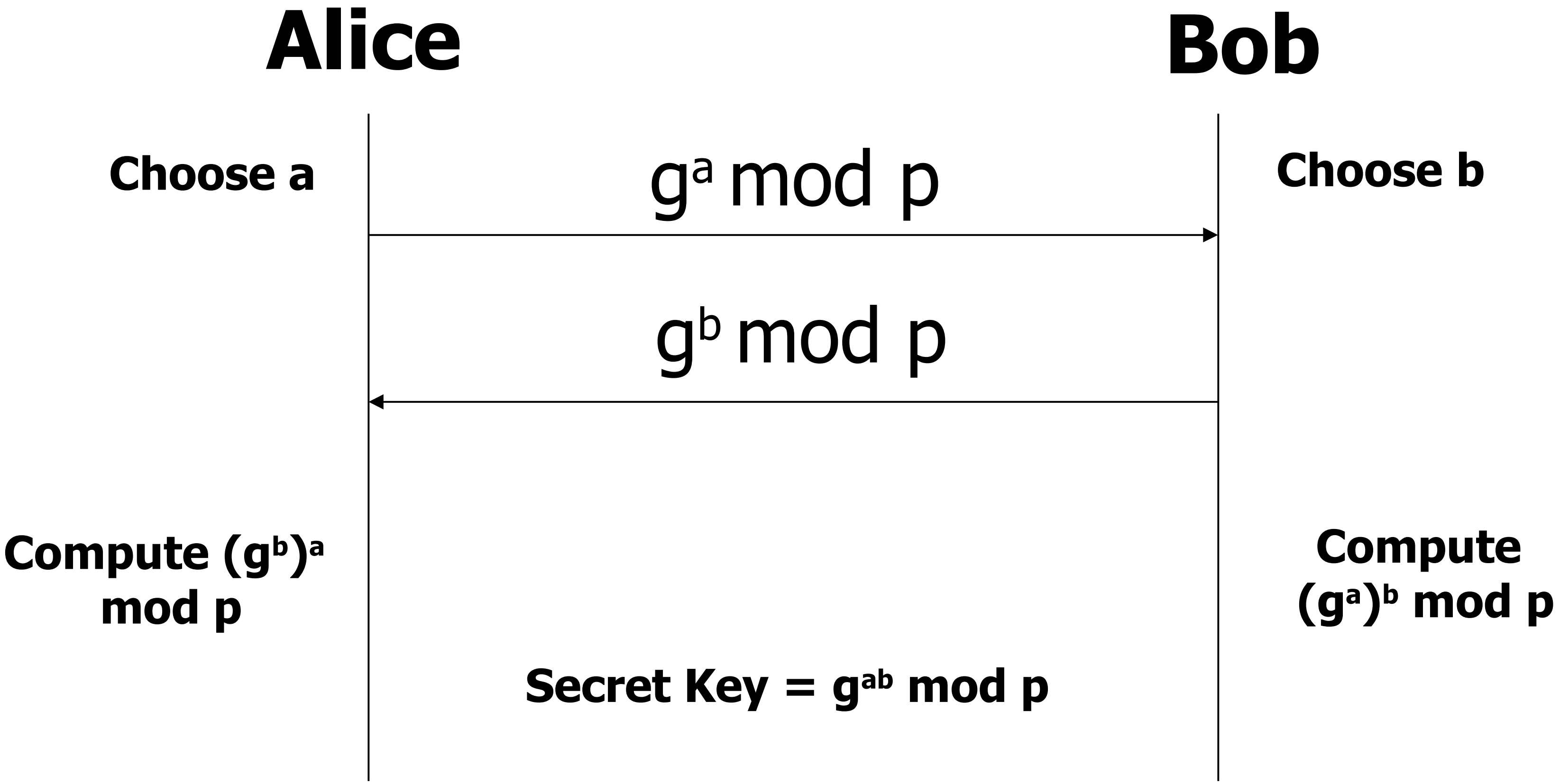
$\{data\}_k$

Symmetric (e.g. AES)

14.4.2. Diffie-Hellman (DH) (1)

- Key exchange (over insecure channel) without public-key certificates?
- DH: use public parameters *g*, *p*
  - □ Large prime number *p*
  - □ Generator *g* (of $Z_p = \{1, \ldots, p\text{-}1\}$), i.e. powers g, g^2, …, g^{p-1} produce all these elements
- Alice & Bob generate rand #s *a*, *b* respectively
- Using *g*, *p*, *a*, *b*, they can create a secret known only to them (relies on hardness of solving the discrete log problem)

14.4.2. Diffie-Hellman (DH) (2)



Alice

Bob

Choose a

$g^a \bmod p$

Choose b

$g^b \bmod p$

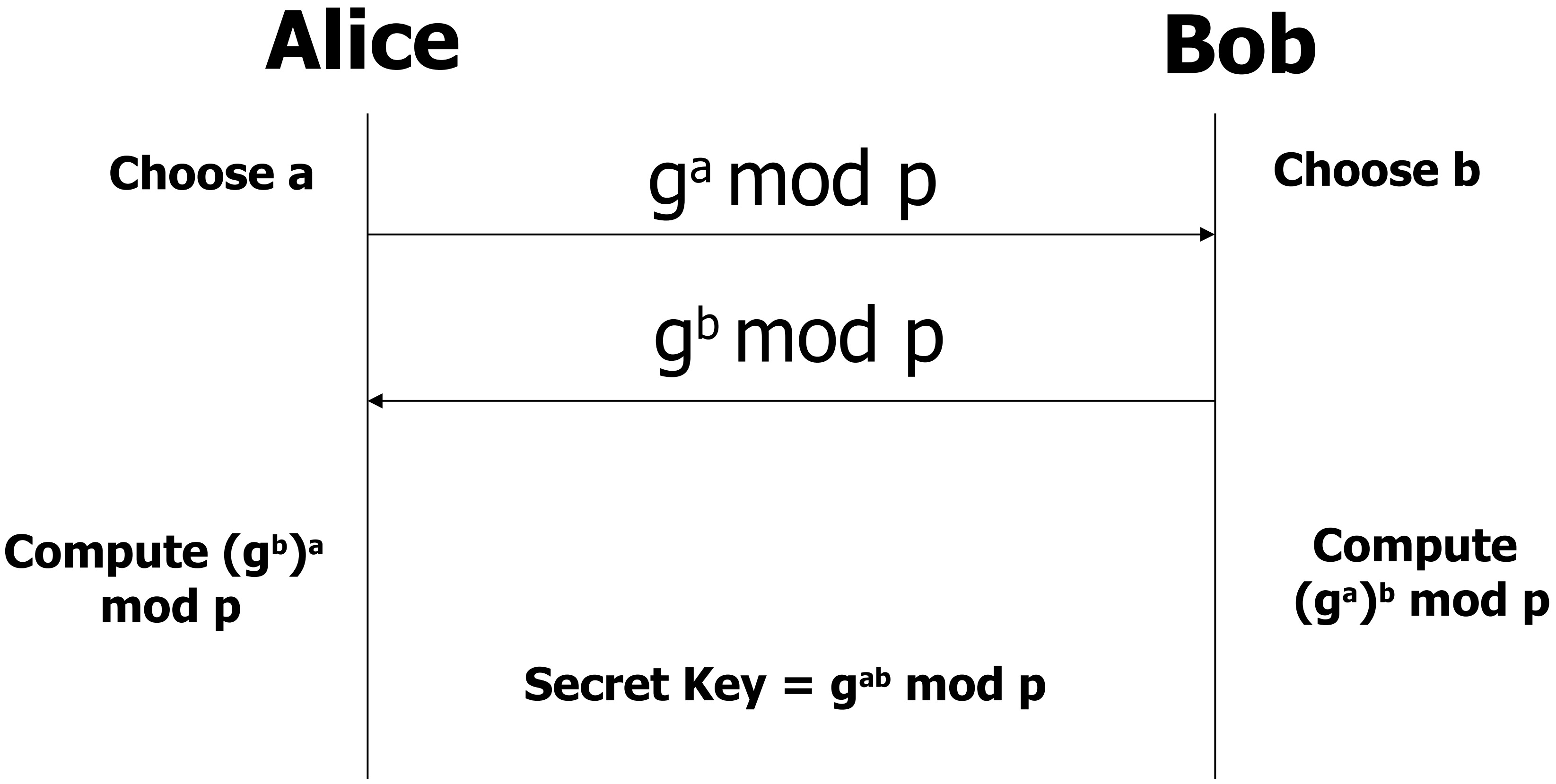
Compute $(g^b)^a$ mod p

Compute $(g^a)^b$ mod p

Secret Key = g^{ab} mod p

Eve can compute $(g^a)(g^b) = g^{a+b}$ mod p but that's not the secret key!

14.4.2. Diffie-Hellman (DH) (2)



Alice

Bob

Choose a

$g^a \bmod p$

Choose b

$g^b \bmod p$

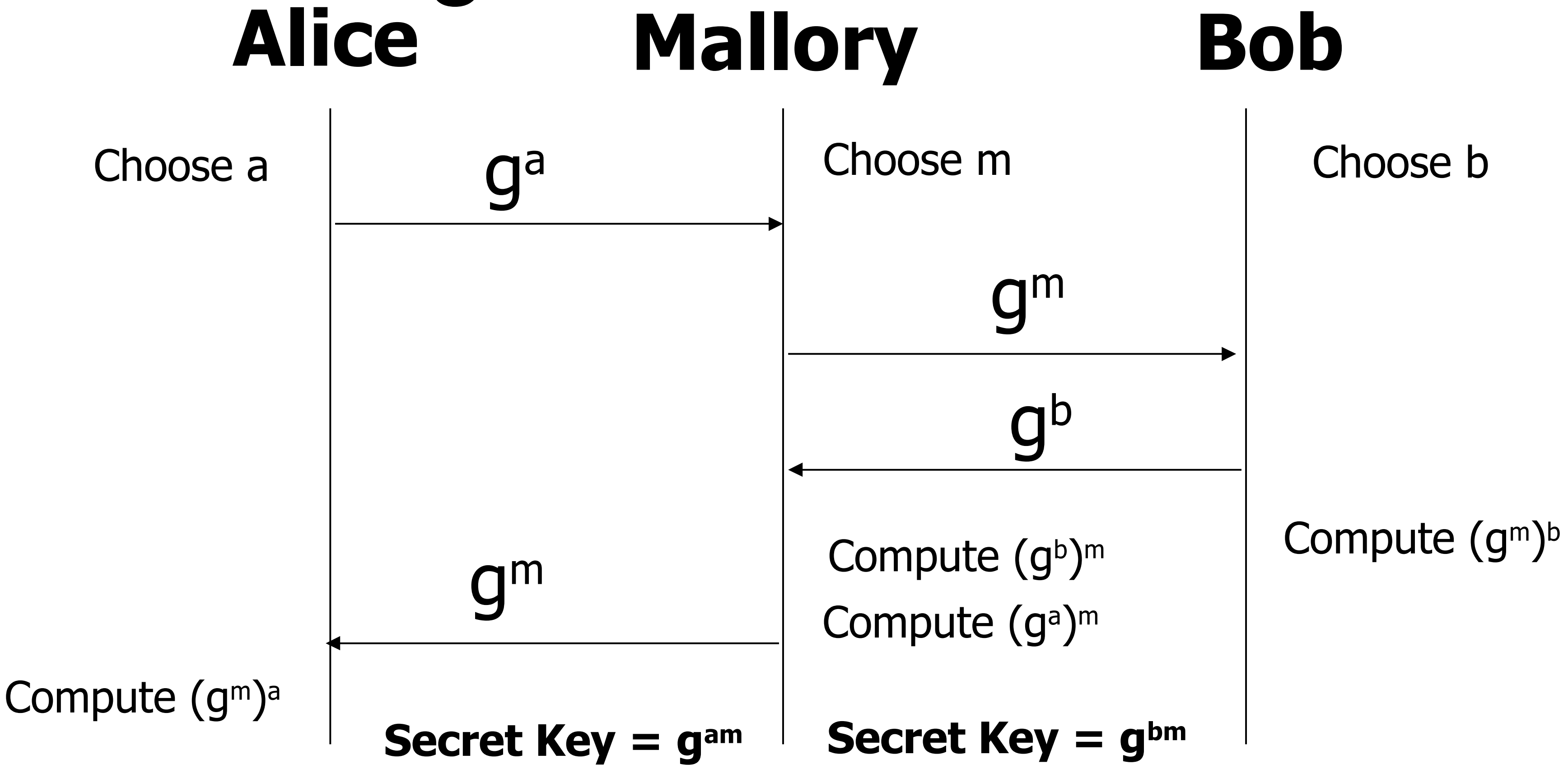
Compute $(g^b)^a$ mod p

Compute $(g^a)^b$ mod p

Secret Key = g^{ab} mod p

Eve can compute $(g^a)(g^b) = g^{a+b}$ mod p but that’s not the secret key!

14.4.2. Man-in-the-Middle Attack against DH



Alice **Mallory** **Bob**

Choose a

g^a

Choose m

Choose b

g^m

g^b

Compute $(g^m)^b$

Compute $(g^b)^m$

g^m

Compute $(g^a)^m$

Compute $(g^m)^a$

Secret Key = g^{am}

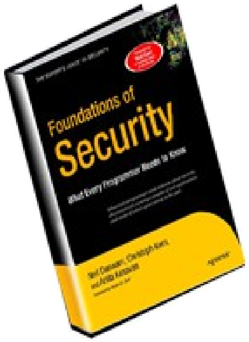
Secret Key = g^{bm}

Mallory can see all communication between Alice & Bob!

CHAPTER 15

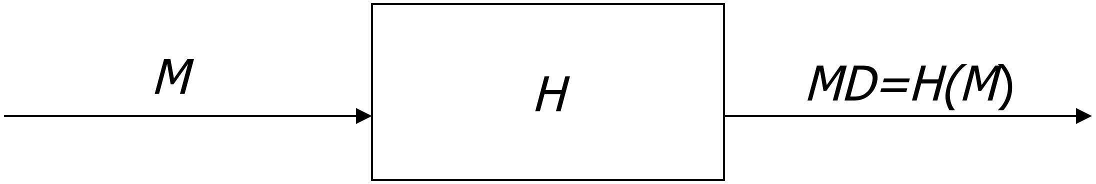
MACs and Signatures

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; http://www.foundationsofsecurity.com). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.



15.1. Secure Hash Functions

- Given arbitrary-length input, *M*, produce fixed-length output (message digest), *H(M)*, such that:



- *Efficiency*: Easy to compute *H*
- *One-Way/Pre-Image resistance*: Given *H(M)*, hard to compute *M* (*pre-image*)
- *Collision resistance*: Hard to find $M_1 \neq M_2$ such that $H(M_1) = H(M_2)$

15.1. Secure Hash Functions Examples

- Non-Examples:
  - Add ASCII values (collisions): *H('AB') = H('BA')*
  - Checksums CRC32 not one-way or collision-resistant

- MD5: “Message Digest 5” invented by Rivest
  - Input: multiple of 512-bits (padded)
  - Output: 128-bits
- SHA1: developed by NIST & NSA
  - Input: same as MD5, 512 bits
  - Output: 160-bits

15.2. MACs

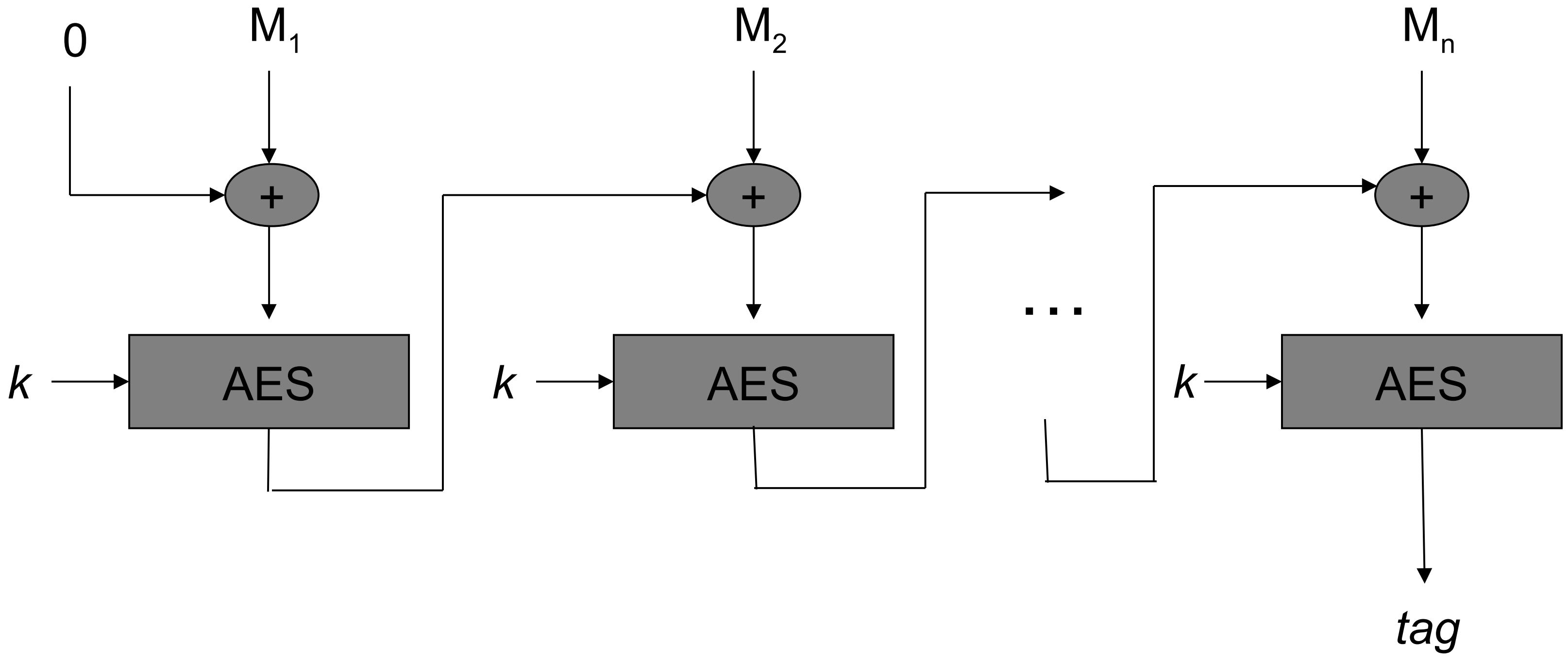
- Used to determine sender of message

- If Alice and Bob share key *k*, then Alice sends message *M* with MAC *tag t = MAC(M,k)*

- Then Bob receives *M’* and *t’* and can check if the message or signature has been tampered by verifying *t’ = MAC(M’, k)*

15.2.1. CBC MACs

- Encrypt message with block cipher in CBC mode
- IV = 0, last encrypted block can serve as tag
- Insecure for variable-length messages



15.2.2. HMAC

- Secure hash function to compute MAC
- Hash function takes message as input while MAC takes message and key
- Simply prepending key onto message is not secure enough (e.g. given MAC of *M*, attacker can compute MAC of *M*||*N* for desired *N*)

- Def: $HMAC(M,k) = H((K \oplus \text{opad}) \,\|\, H((K \oplus \text{ipad}) \,\|\, M))$
  - □ Where *K* is key *k* padded with zeros
  - □ opad, ipad are hexadecimal constants

15.2.2. HMAC

- Secure hash function to compute MAC
- Hash function takes message as input while MAC takes message and key
- Simply prepending key onto message is not secure enough (e.g. given MAC of *M*, attacker can compute MAC of *M*||*N* for desired *N*)

- Def: $HMAC(M,k) = H((K \oplus \text{opad}) \| H((K \oplus \text{ipad}) \| M))$
    - Where *K* is key *k* padded with zeros
    - opad, ipad are hexadecimal constants

15.3. Signatures (1)

- Two major operations: *P*, principal
  - *Sign(M, k)* – *M* is message
  - *Verify(M, sig, P)* – *sig* is signature to be verified
- *Signature*: sequence of bits produced by *Sign()* such that *Verify(M, sig, P) , (sig == Sign(M, k))*
  - Non-repudiable evidence that *P* signed *M*
  - Many applications: SSL, to sign binary code, authenticate source of e-mail
- Use asymmetric encryption ops *F* & F^{-1}

15.3. Signatures (2)

- *S()* & *V()* : implement sign & verify functions

- Signature is $s = S(M, k_s) = F^{-1}(h(M), k_s)$
  - □ Decrypt hash with secret key
  - □ Only signer (principal with secret key) can sign

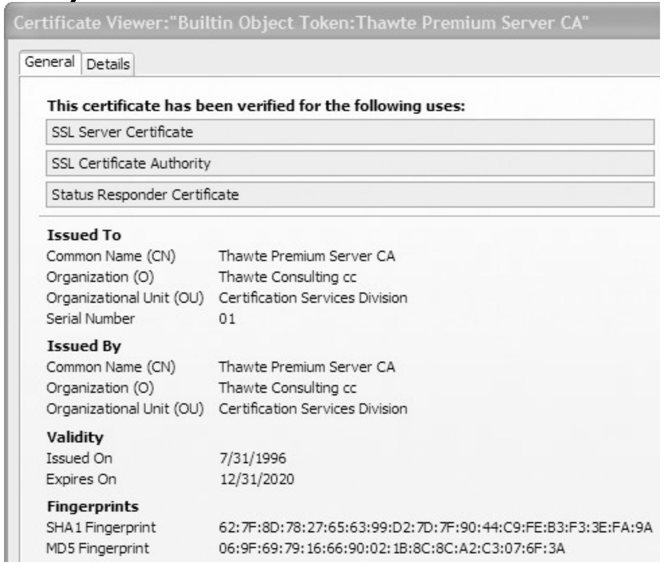
- Verify *s*: $V(M, s, k_p) = (F(s,k_p) == h(M))$
  - □ Encrypting with public key
  - □ Allows anyone to verify a signature
  - □ Need to bind principal’s identity to their public key

15.3.1. Certificates & CAs (1)

- Principal needs certificate from CA (i.e. its digital signature) to bind his identity to his public key
- CA must first sign own certificate attesting to own identity (“root”)
- Certificate, $C(P)$, stored as text: name of principal P, public key ($k_{p(P)}$), expiration date
- $C(P) = (C_{text}(P), C_{sig}(P))$
- Root Certificate, $C(CA)$, looks like
  - □ $C_{text}(CA) = (\text{"CA"}, \mathrm{k}_{\mathrm{p(CA)}}, \mathrm{exp})$
  - □ $C_{sig}(CA) = S(\mathrm{C}_{\mathrm{text}}(\mathrm{CA}), \mathrm{k}_{\mathrm{s(CA)}})$

15.3.1. Certificates & CAs (2)

- Alice constructs certificate text:
  - $C_{text}(Alice)=("Alice", k_{p(Alice)}, exp)$
  - Authenticates herself to CA (through “out-of-band” mechanism such as driver’s license)
- CA signs Alice’s certificate: $C_{sig}(Alice)=S(C_{text}(Alice),k_{s(CA)})$
- Alice has *public key* certificate
  - $C(Alice)=(C_{text}(Alice),C_{sig}(Alice))$
  - Can use to prove that $k_{p(Alice)}$ is her public key



Certificate Viewer:"Builtin Object Token:Thawte Premium Server CA"

This certificate has been verified for the following uses:

SSL Server Certificate

SSL Certificate Authority

Status Responder Certificate

Issued To

Common Name (CN)	Thawte Premium Server CA
Organization (O)	Thawte Consulting cc
Organizational Unit (OU)	Certification Services Division
Serial Number	01

Issued By

Common Name (CN)	Thawte Premium Server CA
Organization (O)	Thawte Consulting cc
Organizational Unit (OU)	Certification Services Division

Validity

Issued On	7/31/1996
Expires On	12/31/2020

Fingerprints

SHA1 Fingerprint	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
MD5 Fingerprint	06:9F:69:79:16:66:90:02:1B:8C:8C:A2:C3:07:6F:3A

15.3.2. Signing and Verifying

- Signing: $sig = Sign(M, k_{s(P)}) = (S(M, k_{s(P)}), C(P))$
  - Compute $S()$ with secret key: $sig.S$
  - Append certificate: $sig.C$

- Verifying: $Verify(M, sig, P) =$
  - $V(M, sig.S, k_{p(P)})$ &
  - $V(sig.C_{text}(P), sig.C_{sig}(P), k_{p(CA)})$ &
  - $(C_{text}(P).name == P)$ &
  - $(today < sig.C_{text}(P).date)$

signature verifies message?

signed by CA?

name matches on cert?

certificate not expired?

15.3.3. Registration Authorities

- Authenticating every principal can burden CA
- Can authorize RA to authenticate on CA's behalf
  - CA signs certificate binding RA's identity to public key
  - Signature now includes RA's certificate too
  - Possibly many intermediaries in the verification process starting from "root" CA certificate
  - More links in chain, more weak points: careful when verifying signatures
- Ex: IE would not verify intermediate certificates and trust arbitrary domains (anyone could sign)

15.3.4. Web of Trust

- Pretty Good Privacy (PGP): digital signatures can be used to sign e-mail
- “Web of trust” model: users sign own certificates and other’s certificates to establish trust
- Two unknown people can find a certificate chain to a common person trusted by both

HEY, I JUST GOT HOME FROM THE PARTY
THE ONE WITH THE IRC FOLKS?
YEAH.
HOW WAS IT?
GOT TOO DRUNK. I SCREWED UP, BAD.
WHAT HAPPENED?
THERE WAS A GIRL. NO IDEA WHO SHE WAS. DON'T EVEN KNOW HER NAME. I WAS TOO DRUNK TO CARE.
AND WHAT, YOU SLEPT WITH HER?
NO.
I SIGNED HER PUBLIC KEY.
S*** MAN.

Source:
http://xkcd.com/364/

15.4. Attacks Against Hash Functions

- Researchers have been able to obtain collisions for some hash functions
  - □ Collision against SHA-1: 2^{63} computations (NIST recommends phase out by 2010 to e.g. SHA-256)
  - □ MD5 seriously compromised: phase out now!
- Collision attacks can’t fake arbitrary digital signatures (requires finding pre-images)
- However could get 2 documents with same hash and sign one and claim other was signed

15.5. SSL

- *Handshake*: steps client & server perform before exchanging sensitive app-level data

- Goal of handshake: client & server agree on master secret used for symmetric crypto

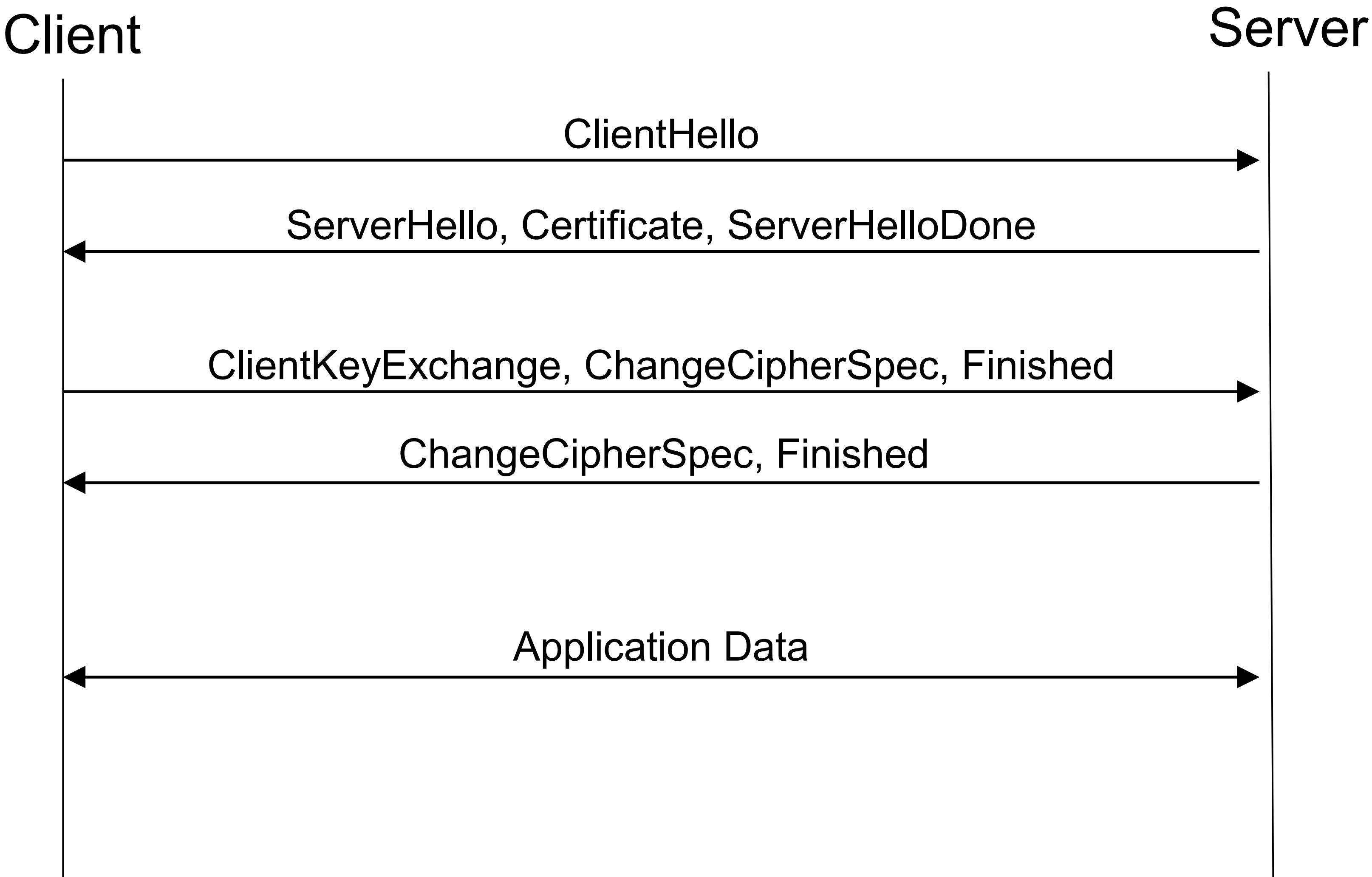
- Two round trips:
  - 1st trip is “hello” messages: what versions of SSL and which cryptographic algorithms supported
  - 2nd varies based on client or mutual authentication

15.5.1. Server-Authenticated Only (1)

- Client creates random pre-master secret, encrypts with server’s public key
- Server decrypts with own private key
- Both compute hashes including random bytes exchanged in “hello” to create master secret

- With master secret, symmetric session key and integrity key derived (as specified by SSL)
- App Data encrypted with symmetric key

15.5.1. Server-Authenticated Only (2)



Client

Server

ClientHello

ServerHello, Certificate, ServerHelloDone

ClientKeyExchange, ChangeCipherSpec, Finished

ChangeCipherSpec, Finished

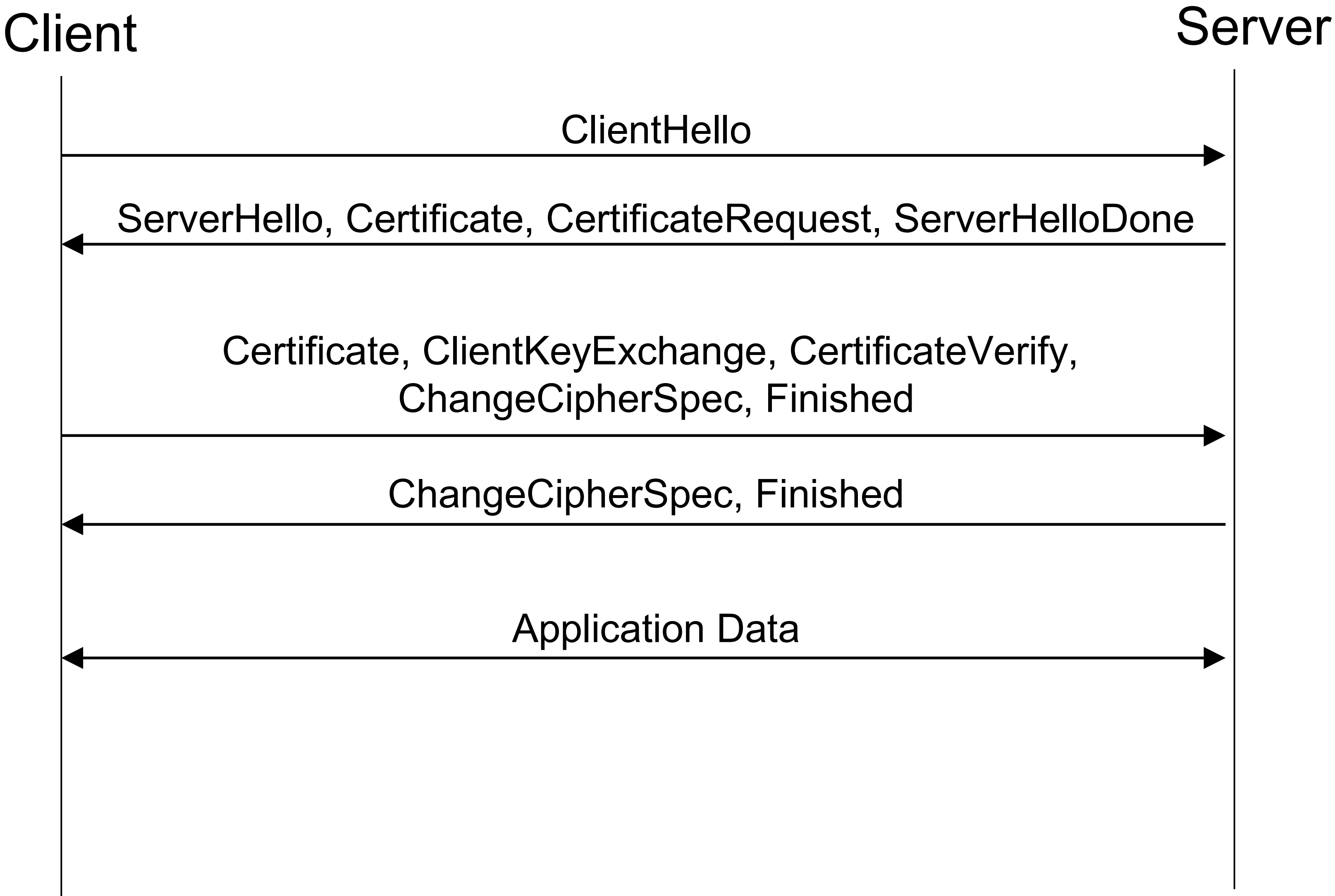
Application Data

15.5.2. Mutual Authentication (1)

- Client also sends own certificate to server
- Sends CertificateVerify message to allow server to authenticate client's public key
- Pre-master secret set, compute master secret
- Derive symmetric key & exchange data

- SSL mechanisms prevent many attacks (e.g. man-in-the-middle) and has performance optimizations (e.g. caching security params)

15.5.2. Mutual Authentication (2)



Client

Server

ClientHello

ServerHello, Certificate, CertificateRequest, ServerHelloDone

Certificate, ClientKeyExchange, CertificateVerify,
ChangeCipherSpec, Finished

ChangeCipherSpec, Finished

Application Data

Summary

- MACs - protect integrity of messages
  - Compute *tag* to detect tampering
  - Ex: CBC-MAC, HMAC (relies on secure hashes)
- Signatures – binds messages to senders
  - Allows anyone to verify sender
  - Prevents forged signatures
  - Use CAs to bind identities to public keys
  - Or use Web of Trust model
- Application: SSL (“Putting it all together”)
  - Relies on Cryptography: symmetric & public-key
  - And MACs & signatures

Some final words

- Using cryptography in a system full of vulnerabilities = fortness built on sand because the keys can be stolen
- Cryptography does not solve the problems, it just simplifies them
  - You cannot hide a 1 Terabyte file
  - You can encrypt the file with a 512 bits key and hide the key =
  - The same problem but much more simpler