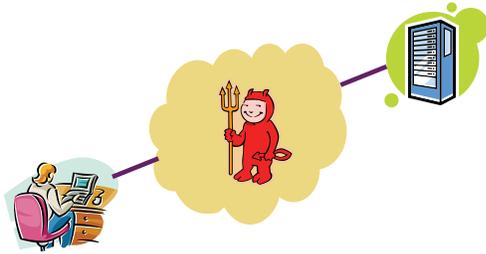
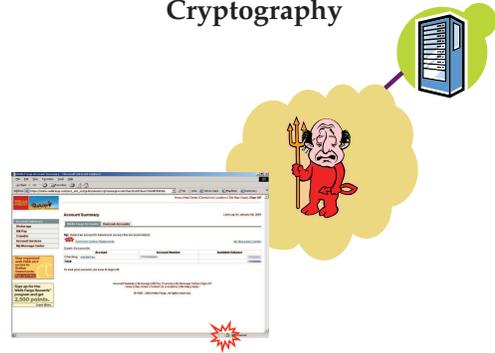


Recall from last lecture



- To a first approximation, attackers control network
- Next two lectures: How to defend against this
 1. Communicate securely despite insecure networks – *cryptography*
 2. Secure small parts of network despite wider Internet

Cryptography



- Crypto important tool for securing communication
 - But often misused
 - Have to understand what it guarantees and what it doesn't

How Cryptography Helps

- **Secrecy**
 - Encryption
- **Integrity**
 - Cryptographic hashes
 - Digital signatures
 - Message authentication codes (MACs)
- **Authentication**
 - Certificates, signatures, MACs
- **Availability**
 - Can't usually be guaranteed by cryptography alone

[Symmetric] Encryption

- Both parties share a secret key K
- Given a message M , and a key K :
 - M is known as the *plaintext*
 - $E(K, M) \rightarrow C$ (C known as the *ciphertext*)
 - $D(K, C) \rightarrow M$
 - Attacker cannot efficiently derive M from C without K
- Note E and D take same argument K
 - Thus, also sometimes called *symmetric* encryption
 - Raises issue of how to get K : more on that later
- Example algorithms: AES, Blowfish, DES, RC4, ...

One-time pad

- Share a completely random key K
- Encrypt M by XORing with K :
$$E(K, M) = M \oplus K$$
- Decrypt by XORing again:
$$D(K, C) = C \oplus K$$
- **Advantage: Information-theoretically secure**
 - Given C but not K , any M of same length equally likely
 - Also: fast!
- **Disadvantage: K must be as long as M**
 - Makes distributing K for each message difficult

Idea: Computational security

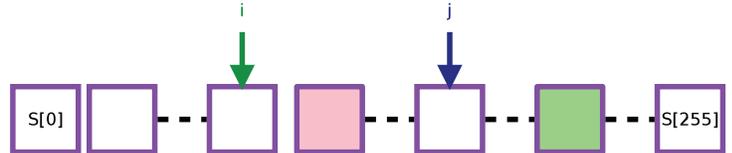
- Distribute small K securely (e.g., 128 bits)
- Use K to encrypt far larger M (e.g., 1 MByte file)
- Given $C = E(K, M)$, may be only one possible M
 - If M has redundancy
- But believed computationally intractable to find
 - E.g., could try every possible K , but 2^{128} keys a lot of work!

Types of encryption algorithms

- **Stream ciphers – pseudo-random pad**
 - Generate pseudo-random stream of bits from short key
 - Encrypt/decrypt by XORing with stream as if one-time pad
 - But **NOT** one-time PAD! (People who claim so are frauds!)
 - In practice, many stream ciphers uses have run into problems
- **More common algorithm type: Block cipher**
 - Operates on fixed-size blocks (e.g., 64 or 128 bits)
 - Maps plaintext blocks to same size ciphertext blocks
 - Today should use AES; other algorithms: DES, Blowfish, ...

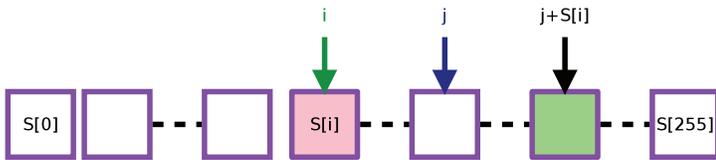
Example stream cipher (RC4)

- **Initialization:**
 - $S[0 \dots 255] \leftarrow$ permutation $(0, \dots, 255)$ (based on key); $i \leftarrow 0$; $j \leftarrow 0$;
- **Generating pseudo-random bytes:**
 - $i \leftarrow (i + 1) \bmod 256$;
 - $j \leftarrow (j + S[i]) \bmod 256$;
 - swap $S[i] \leftrightarrow S[j]$;
 - return $S[(S[i] + S[j]) \bmod 256]$;



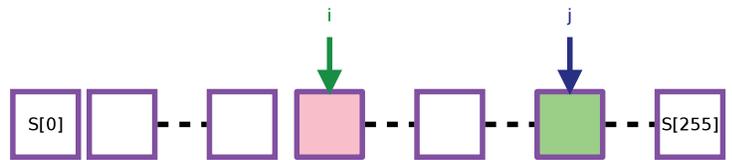
Example stream cipher (RC4)

- **Initialization:**
 - $S[0 \dots 255] \leftarrow$ permutation $(0, \dots, 255)$ (based on key); $i \leftarrow 0$; $j \leftarrow 0$;
- **Generating pseudo-random bytes:**
 - $i \leftarrow (i + 1) \bmod 256$;
 - $j \leftarrow (j + S[i]) \bmod 256$;
 - swap $S[i] \leftrightarrow S[j]$;
 - return $S[(S[i] + S[j]) \bmod 256]$;



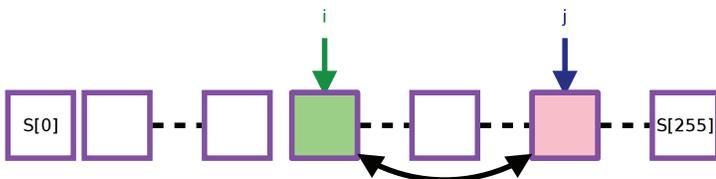
Example stream cipher (RC4)

- **Initialization:**
 - $S[0 \dots 255] \leftarrow$ permutation $(0, \dots, 255)$ (based on key); $i \leftarrow 0$; $j \leftarrow 0$;
- **Generating pseudo-random bytes:**
 - $i \leftarrow (i + 1) \bmod 256$;
 - $j \leftarrow (j + S[i]) \bmod 256$;
 - swap $S[i] \leftrightarrow S[j]$;
 - return $S[(S[i] + S[j]) \bmod 256]$;



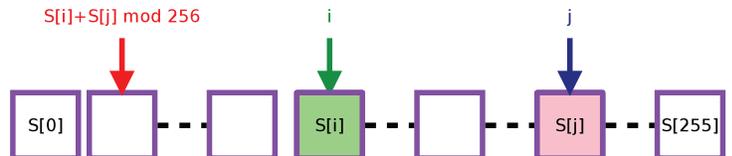
Example stream cipher (RC4)

- **Initialization:**
 - $S[0 \dots 255] \leftarrow$ permutation $(0, \dots, 255)$ (based on key); $i \leftarrow 0$; $j \leftarrow 0$;
- **Generating pseudo-random bytes:**
 - $i \leftarrow (i + 1) \bmod 256$;
 - $j \leftarrow (j + S[i]) \bmod 256$;
 - swap $S[i] \leftrightarrow S[j]$;
 - return $S[(S[i] + S[j]) \bmod 256]$;



Example stream cipher (RC4)

- **Initialization:**
 - $S[0 \dots 255] \leftarrow$ permutation $(0, \dots, 255)$ (based on key); $i \leftarrow 0$; $j \leftarrow 0$;
- **Generating pseudo-random bytes:**
 - $i \leftarrow (i + 1) \bmod 256$;
 - $j \leftarrow (j + S[i]) \bmod 256$;
 - swap $S[i] \leftrightarrow S[j]$;
 - return $S[(S[i] + S[j]) \bmod 256]$;



RC4 security

- **Warning: Lecture goal just to give a feel**
 - May omit critical details necessary to use RC4 and other algorithms securely
- **RC4 Goal: Indistinguishable from random sequence**
 - Given part of the output stream, it should be intractable to distinguish it from a truly random string
- **Problems**
 - Second byte of RC4 is 0 with twice expected probability [MS01]
 - Bad to use many related keys (see WEP 802.11b) [FMS01]
 - Recommendation: Discard the first 256 bytes of RC4 output [RSA, MS]

Wrong!

- **Let's say an attacker has the following:**
 - $c_1 = \text{Encrypt}(s, \text{"Visa card \#3273..."})$
 - $c_2 = \text{Encrypt}(s, \text{"Order confirmed, have a nice day"})$
- **Now compute:**
 - $m \leftarrow c_1 \oplus c_2 \oplus \text{"Order confirmed, have a nice day"}$
- **Lesson: Never re-use keys with a stream cipher**
 - Similar lesson applies to one-time pads (That's why they're called **one-time** pads.)

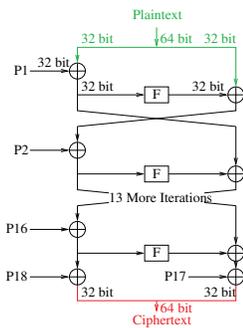
Example use of stream cipher

- **Pre-arrange to share secret s with web vendor**
- **Exchange payment information as follows**
 - Send: $E(s, \text{"Visa card \#3273..."})$
 - Receive: $E(s, \text{"Order confirmed, have a nice day"})$
- **Now an eavesdropper can't figure out your Visa #**

Wired Equivalent Privacy (WEP)

- **Initial security standard for 802.11**
 - Serious weaknesses discovered: able to crack a connection in minutes
 - Replaced by WPA in 2003
- **Stream cipher, basic mode uses 64-bit key: 40 bits are fixed and 24 bits are an initialization vector (IV), specified in the packet**
 - One basic flaw: if IV ever repeated (only 4 million packets), then key is reused
 - Many implementations would reset IV on reboot
- **Other flaws include IV collisions, altered packets, etc.**

Example block cipher (blowfish)



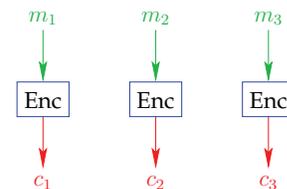
"Feistel network"

(Note: This is just to give an idea; it's not a complete description)

- **Derive F and 18 subkeys ($P_1 \dots P_{18}$) from key**
- **Divide plaintext block into two halves, L_0 and R_0**
- $R_i = L_{i-1} \oplus P_i$
 $L_i = R_{i-1} \oplus F(R_i)$
- $R_{17} = L_{16} \oplus P_{17}$
 $L_{17} = R_{16} \oplus P_{18}$
- **Output $L_{17}R_{17}$.**

Using a block cipher

- **In practice, message may be more than one block**
- **Encrypt with ECB (electronic code book) mode:**
 - Split plaintext into blocks, and encrypt separately

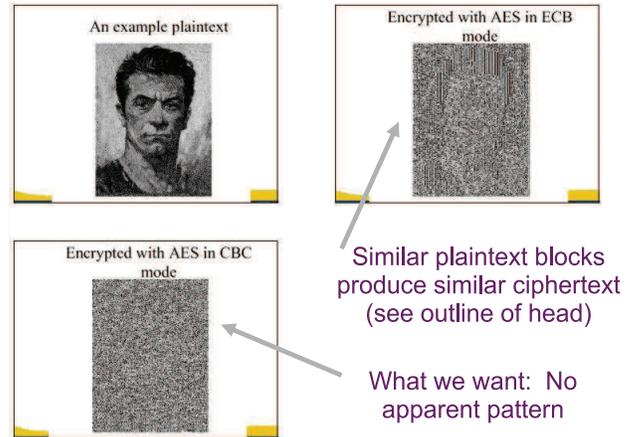


- Attacker can't decrypt any of the blocks; message secure
- **Note: can re-use keys, unlike stream cipher**
 - Every block encrypted with cipher will be secure

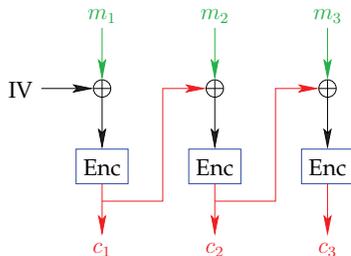
Wrong!

- **Attacker will learn of repeated plaintext blocks**
 - If transmitting sparse file, will know where non-zero regions lie
- **Example: Intercepting military instructions**
 - Most days, send encryption of "nothing to report."
 - On eve of battle, send "attack at dawn."
 - Attacker will know when battle plans are being made

Another example [Preneel]



Cipher-block chaining (CBC)



- **Choose initialization vector (IV) for each message**
 - Can be 0 if key only ever used to encrypt one message
 - Choose randomly for each message if key re-used
 - Can be publicly known (e.g., transmit openly with ciphertext)
- $c_1 = E(K, m_1 \oplus IV)$, $c_i = E(K, m_i \oplus c_{i-1})$
 - Ensures repeated blocks are not encrypted the same

Encryption modes

- CBC, ECB are encryption modes, but there are others
- **Cipher Feedback (CFB) mode:** $c_i = m_i \oplus E(K, c_{i-1})$
 - Useful for messages that are not multiple of block size
- **Output Feedback (OFB) mode:**
 - Repeatedly encrypt IV & use result like stream cipher
- **Counter (CTR) mode:** $c_i = m_i \oplus E(K, i)$
 - Useful if you want to encrypt in parallel
- **Q: Given a shared key, can you transmit files securely over net by just encrypting them in CBC mode?**

2-minute break



Problem: Integrity

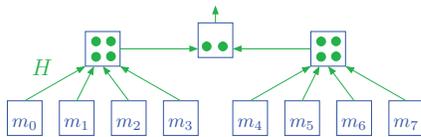
- **Attacker can tamper with messages**
 - E.g., corrupt a block to flip a bit in next
- **What if you delete original file after transfer?**
 - Might have nothing but garbage at recipient
- **Encryption does not guarantee integrity**
 - A system that uses encryption alone (no integrity check) is often incorrectly designed.
 - Exception: Cryptographic storage (to protect disk if stolen)

Message authentication codes

- **Message authentication codes (MACs)**
 - Sender & receiver share secret key K
 - For message m , compute $v \leftarrow \text{MAC}(K, m)$
 - Recipient runs $\text{CHECK}(K, v, m) \rightarrow \{\text{yes, no}\}$
 - Intractable to produce valid $\langle m, v \rangle$ without K
- **To send message securely, append MAC**
 - Send $\{m, \text{MAC}(K, m)\}$ (m could be ciphertext, $E(K', M)$)
 - Receiver of $\{m, v\}$ discards unless $\text{CHECK}(K, v, m) = \text{yes}$
- **Careful of Replay – don't believe previous $\{m, v\}$**

Applications of cryptographic hashes

- **Small hash uniquely specifies large data**
 - Hash a file, remember the hash value
 - Recompute hash later, if same value no tampering
 - Hashes often published for software distribution
- **Hash tree [Merkle] lets you check small piece of large file or database with log number of nodes**



Order of Encryption and MACs

- **Should you Encrypt then MAC, or vice versa?**
- **MACing encrypted data is always secure**
- **Encrypting $\{\text{Data}+\text{MAC}\}$ may not be secure!**
 - Consider the following secure, but stupid encryption alg
 - Transform $m \rightarrow m'$ by mapping each bit to two bits:
Map $0 \rightarrow 00$ (always), $1 \rightarrow \{10, 01\}$ (randomly pick one)
 - Now encrypt m' with a stream cipher to produce c
 - Attacker flips two bits of c —if msg rejected, was 0 bit in m

Cryptographic hashes

- **Hash arbitrary-length input to fixed-size output**
 - Typical output size 160–512 bits
 - Cheap to compute on large input (faster than network)
- **Collision-resistant: Intractable to find**
 $x \neq y$ such that $H(x) = H(y)$
 - Of course, many such collisions exist
 - But no one has been able to find one, even after analyzing the algorithm
- **Historically most popular hash SHA-1**
 - [Nearly] broken
 - Today should use SHA-256 or SHA-512
 - Competition underway for new hash standard

HMAC

- **Use cryptographic hash to produce MAC**
- $\text{HMAC}(K, m) = H(K \oplus \text{opad}, H(K \oplus \text{ipad}, m))$
 - H is a cryptographic hash such as SHA-1
 - ipad is $0x36$ repeated 64 times, opad $0x5c$ repeated 64 times
- **To verify, just recompute HMAC**
 - $\text{CHECK}(K, v, m) = (v \stackrel{?}{=} \text{HMAC}(K, m))$
 - Many MACs are deterministic and work like this (“PRFs”), but fastest MACs randomized so CHECK can't just recompute
- **Note: Don't just use $H(K, M)$ as a MAC**
 - Say you have $\{M, \text{SHA-1}(K, M)\}$, but not K
 - Can produce $\{M', \text{SHA-1}(K, M')\}$ where $M' \neq M$
 - Hashes provide collision resistance, but do not prevent spoofing new messages

Public key encryption

- **Three randomized algorithms:**
 - Generate – $G(1^k) \rightarrow K, K^{-1}$ (randomized)
 - Encrypt – $E(K, m) \rightarrow \{m\}_K$ (randomized)
 - Decrypt – $D(K^{-1}, \{m\}_K) \rightarrow m$
- **Provides secrecy, like conventional encryption**
 - Can't derive m from $\{m\}_K$ without knowing K^{-1}
- **Encryption key K can be made public**
 - Can't derive K^{-1} from K
 - Everyone can use same pub. key to encrypt for one recipient
- **Note: Encrypt *must* be randomized**
 - Same message must encrypt to different ciphertext each time
 - Otherwise, can easily guess plaintext from small message space (E.g., encrypt “yes”, encrypt “no”, see which matches message)

Digital signatures

- **Three (randomized) algorithms:**
 - *Generate* – $G(1^k) \rightarrow K, K^{-1}$ (randomized)
 - *Sign* – $S(K^{-1}, m) \rightarrow \{m\}_{K^{-1}}$ (can be randomized)
 - *Verify* – $V(K, \{m\}_{K^{-1}}, m) \rightarrow \{\text{yes, no}\}$
- **Provides integrity, like a MAC**
 - Cannot produce valid $\langle m, \{m\}_{K^{-1}} \rangle$ pair without K^{-1}
 - But only need K to verify; cannot derive K^{-1} from K
 - So K can be publicly known

Cost of cryptographic operations

- **Cost of public key algorithms significant**
 - E.g., encrypt or sign only ~ 100 msgs/sec
 - Can only encrypt small messages ($<$ size of key)
 - Signature cost relatively insensitive to message size
 - Some algorithm variants provide faster encrypt/verify (e.g., Rabin, RSA-3 can encrypt $\sim 10,000$ msgs/sec)
- **In contrast, symmetric algorithms much cheaper**
 - Symmetric can encrypt+MAC faster than 1Gbps/sec LAN

Server authentication

- **Often want to communicate securely with a server**
- **Easy once you have server's public key**
 - Use public key to bootstrap symmetric keys
- **Problem: Key management**
 - How to get server's public key?
 - How to know the key is really server's?

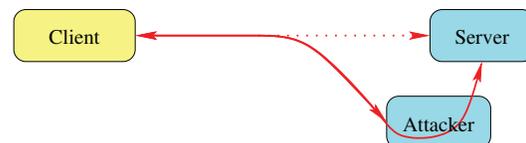
Popular public key algorithms

- **Encryption: RSA, Rabin, ElGamal**
- **Signature: RSA, Rabin, ElGamal, Schnorr, DSA, ...**
- **Warning: Message padding critically important**
 - E.g., basic idea behind RSA encryption simple
 - Just modular exponentiation of large integers
 - But simple transformations of messages to numbers not secure
- **Many keys support both signing & encryption**
 - But Encrypt/Decrypt and Sign/Verify different algorithms!
 - Common error: Sign by "encrypting" with private key

Hybrid schemes

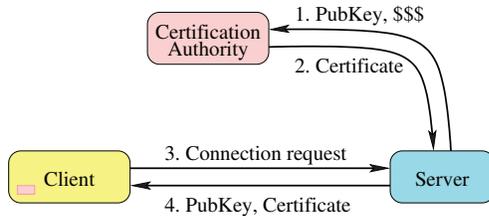
- **Use public key to encrypt symmetric key**
 - Send message symmetrically encrypted: $\{\text{msg}\}_{K_S}, \{K_S\}_{K_P}$
- **Use PK to negotiate secret session key**
 - Use Public Key crypto to establish 4 keys symmetric keys
 - Client sends server: $\{\{m_1\}_{K_1}, \text{MAC}(K_2, \{m_1\}_{K_1})\}$
 - Server sends client: $\{\{m_2\}_{K_3}, \text{MAC}(K_4, \{m_2\}_{K_3})\}$
- **Often want mutual authentication (client & server)**
 - Or more complex, user(s), client, & server
- **Common pitfall: signing underspecified messages**
 - E.g., Always specify intended recipient in signed messages
 - Should also specify expiration, or better yet fresh data
 - Otherwise like signing a blank check...

Danger: impersonating servers

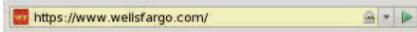


- **Attacker pretends to be server, gives its own pub key**
- **Attacker mounts man-in-the-middle attack**
 - Looks just like server to client (except for different public key)
 - Attacker sees, then re-encrypts sensitive communications
 - Attacker can also send bad data back to client

One solution: Certificate authorities (CAs)



- **Everybody trusts some certificate authority**
- **Everybody knows CA's public key**
 - E.g., built into web browser
- **This is how HTTPS (over SSL/TLS) works**
 - Active when you see padlock in your web browser



Digital certificates

- **A digital certificate binds a public key to name**
 - E.g., "www.ebay.com's public key is 0x39f32641..."
 - Digitally signed with a CA's private key
- **Certificates can be *chained***
 - E.g., start with root CAs like Verisign
 - Verisign can sign Stanford's public key
 - Stanford can sign keys for cs.stanford.edu, etc.
 - Not as widely supported as it should be (Maybe because CAs want \$300 for every Stanford server)
- **Assuming you trust the CA, solves the key management problem**

Another solution: Use passwords

- **User remembers a password to authenticate himself**
 - Server stores password or secret derived from password
 - Can then use password to authenticate server to client, as well

- **Simplest example:**



- **Big limitations of above (simple) protocol:**
 - Users choose weak passwords
 - Since pubkey known, attacker gets one message from server, then guess all common passwords offline
 - Also, users employ same passwords at multiple sites
- **Limitations addressed by fancier crypto protocols**
 - E.g., SRP, PAKE₂ protocols developed here at Stanford