CSCI 3753
Operating Systems

Security

Chapters 15

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 11/30/17

Program Threats

- Writing a program that creates a breach of security.
- Trojan Horse
  - A code segment that misuses its environment, e.g. supervisor mode
- Trap Door
  - Programmer may leave a hole in the software that only he/she is capable of using
- Logic Bomb
  - A program code that initiates a security incident only under certain circumstances

Stack and Buffer Overflow

- A programmer neglected to code bounds checking on an input field for a program that runs in supervisor mode
  - Cause an overflow until bytes are written on the stack
  - Overwrite the return address with the address of the exploit code
  - Write a simple set of code in the next space on stack – simple set of commands, e.g. spawn a shell
  - See example in the next slide

Buggy code

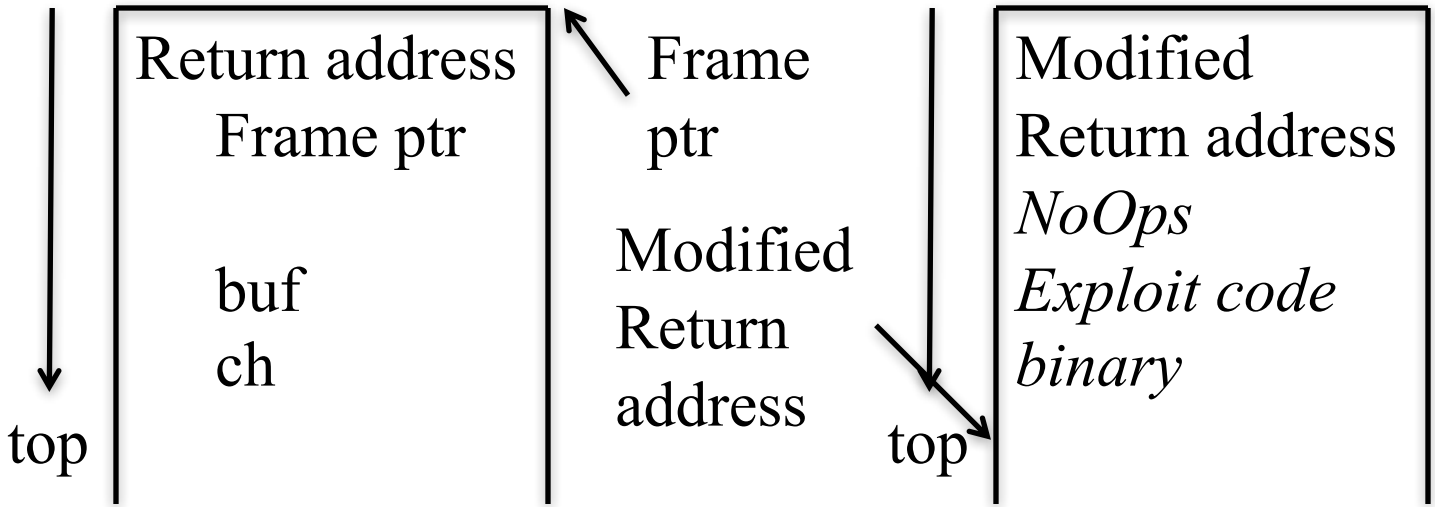
```
int foo_lib (char *ch)
{
    char buf [256];
    strcpy (buf, ch);
    return 0;
}
```

No bound checking

Exploit code

```
int main (argc, char *argv[])
{
    execvp (``\bin\sh'', ``\bin\sh'', NULL);
    return 0;
}
```

Attacker compiles the exploit code
Edits the binary to fit in the stack frame
Calls foo_lib with pointer to edited binary



The newly created shell runs in supervisor mode

Viruses

- A fragment of code embedded in a legitimate program
- Self replicating, designed to infect other programs
- Spread via spams, macros, etc.
- Virus dropper: a trojan horse, inserts the virus into the system

Worms

- A worm is a process that uses the spawn mechanism to duplicate itself
- Use up system resources and lock out other processes

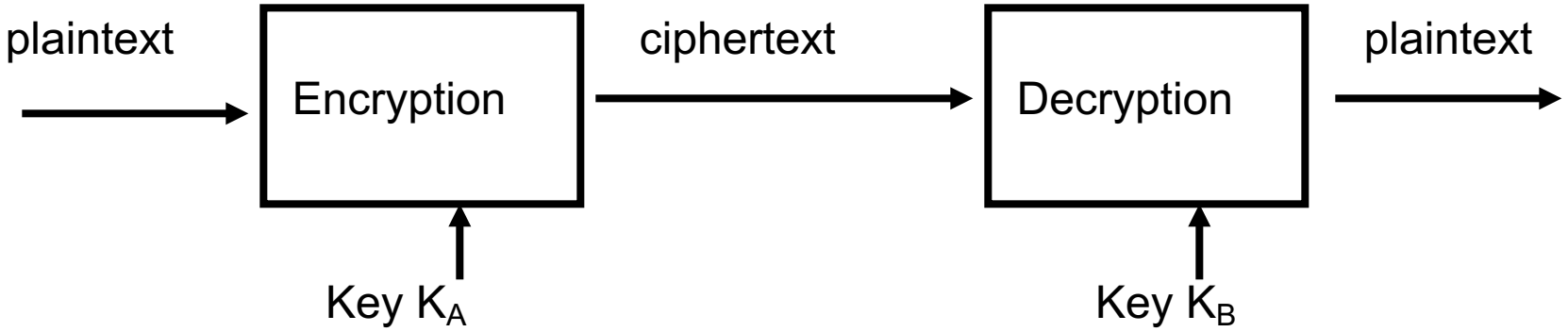
Security

- The six classic properties of security are
  1. Confidentiality
  2. Authentication
  3. Authorization
  4. Integrity
  5. Non-repudiation (verification that an event actually took place)
  6. Availability

  – We will focus primarily on confidentiality, authentication and authorization

Confidentiality

- Encrypt files and/or data communication (e.g. passwords and messages) so only the encryptor and designated decryptors can view the information
  - In the case of files, the encryptor and decryptor are usually the same person.
  - In the case of communication, the encryptor and decryptor are different people
- Modern cryptography uses keys to encrypt and decrypt
  - Only have to protect the keys, not the algorithm for encryption or decryption (caveat: input must be statistically random, otherwise subject to frequency cryptanalysis)



Symmetric key cryptography

- Encryption key and decryption key are same
  - Most of the 2000+ year history of cryptography has involved symmetric key cryptography (until 1970's)
  - Examples of symmetric key standards: AES (Advanced Encryption Standard), Triple-DES (Data Encryption Standard)
  - Encrypted file systems like EFS (Encrypting File System for Windows) and EncFS (for Linux, uses FUSE) employ symmetric key cryptography
    - The symmetric key used to encrypt/decrypt files is itself stored in encrypted form. You enter a password (more accurately, a passphrase) to decrypt this key at run time, which is then used to encrypt/decrypt files.

Confidentiality

- Question: when the decryptor is physically separate from the encryptor, as in a secure remote login, how does the decryptor securely obtain the key K?
  - Common to any remote login problem
  - This is the classic *symmetric key distribution problem*
  - one way is to securely transport the key to the destination
    - but there's no guarantee that a spy won't intercept the key
    - even worse, the spy could copy the key without letting the decryptor or encryptor know, and then eavesdrop on all future encrypted communications!

Diffie-Hellman Key Exchange

- Emerged in the 1970s, invented by Diffie and Hellman (and Merkle)
  - Endpoints exchange public quantities with each other
  - Each endpoint then calculates its symmetric key from these publicly exchanged quantities
    - The symmetric keys calculated are the same
  - Diffie-Hellman key exchange has the property that even though an attacker could eavesdrop on all the public communications, it cannot calculate the symmetric key
  - This solves the classic *symmetric key distribution problem* (with a caveat explained later), and was the foundation for public key cryptography

Diffie-Hellman Key Exchange

Host X

Host Y

Choose a, g, and p
Calculate $A = g^a \bmod p$
Send A, g, and p in the clear

A, g, p

Choose b
Calculate $B=g^b \bmod p$
Send B in the clear

B

$K = B^a \bmod p$

$K = A^b \bmod p$

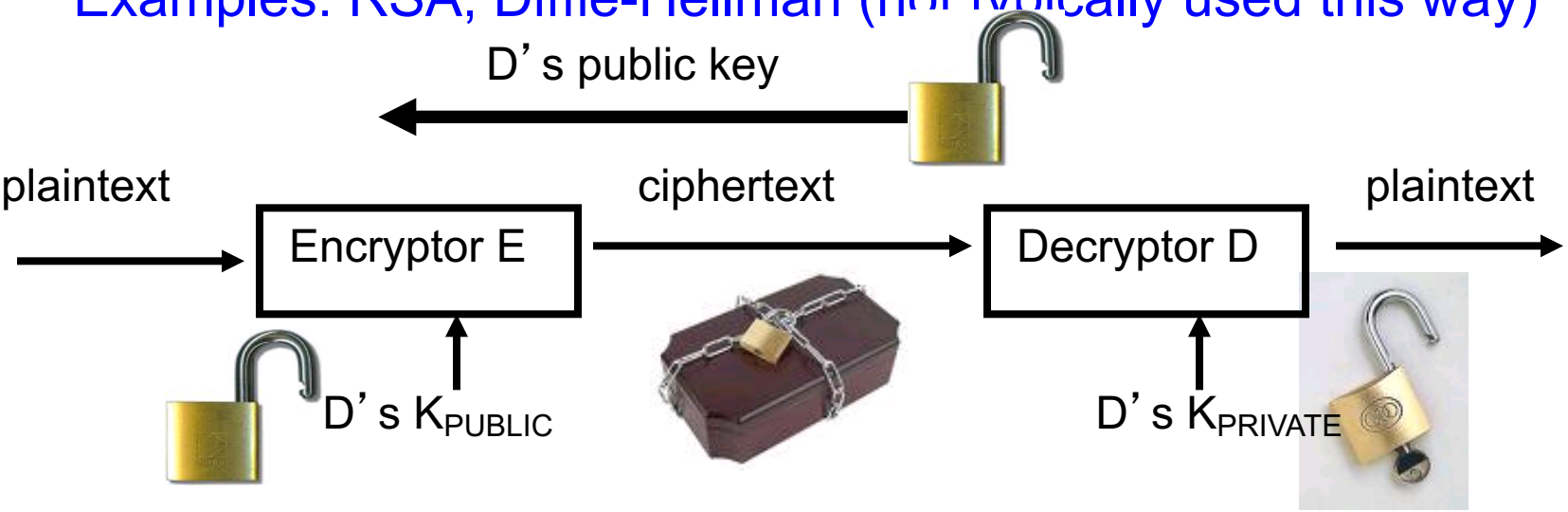
Two mathematical properties: $B^a \bmod p = A^b \bmod p$
And $A=g^a \bmod p$ is not easily invertible, i.e. can't find a from A

Confidentiality

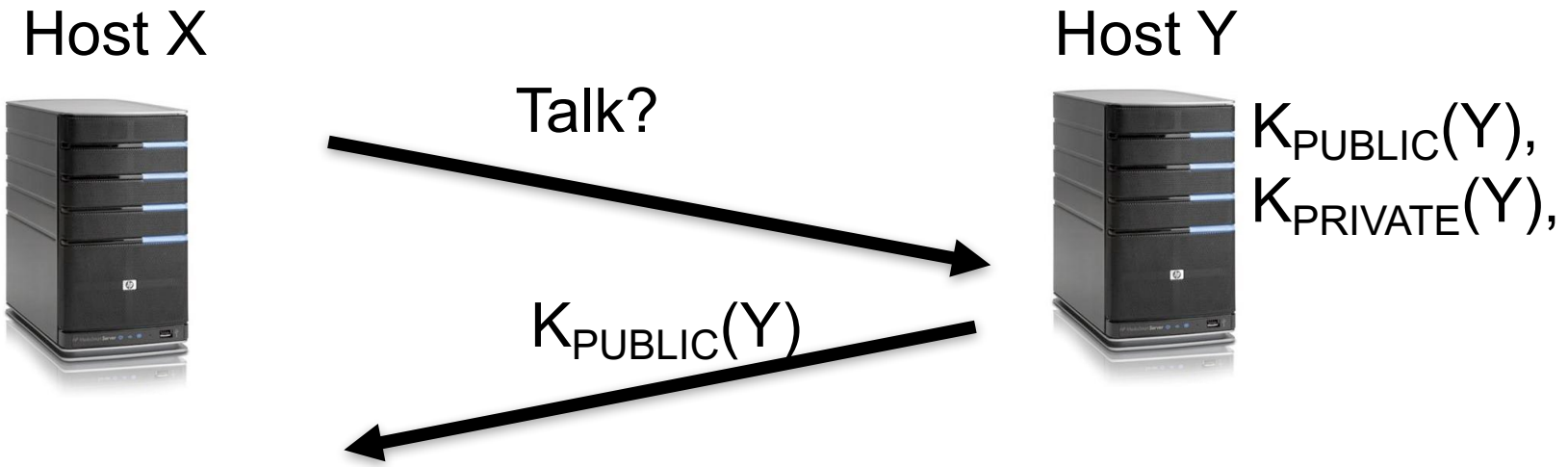
- In Diffie-Hellman:
  - Even if an attacker knows A, g, p and B, as well as the algorithm $f(x) = g^x \bmod p$, they cannot compute K, because they would have to invert f, which is not computationally feasible:
    - They would have to first invert f to find a or b. Then they could calculate K.
  - Diffie-Hellman can also be used for encryption, i.e. public key encryption, not just key establishment
    - This led to the field of public key cryptography – next slide
    - Other algorithms like RSA are typically used for public key cryptography, and Diffie-Hellman is used more for key establishment

Confidentiality: Public Key Cryptography

- The keys at the two endpoints E and D are no longer the same – hence asymmetric
  - D has a *public* key that can be advertised to everyone
  - D has a *private* key only known to itself
- If E wants to send D an encrypted message, E requests D's public key e, and then encrypts its message with D's public key e
- even if an adversary captured the encrypted message and the public key, and even knew the algorithm for encryption and decryption, the adversary cannot decrypt the packet *without D's private key*.
- Examples: RSA, Diffie-Hellman (not typically used this way)



RSA Public Key Cryptography



Encrypt message m with
Y's
public key $K_{PUB}(Y)$
Calculate ciphertext
$c = m^{KPUB(Y)} \bmod p$

send c

Decrypt ciphertext c
$m = c^{KPRIV(Y)} \bmod p$

See text for more details of RSA algorithm

Confidentiality

- SSL/TLS and SSH use a hybrid public key + symmetric key approach to encrypt remote communication between your computer and another computer:
  - Hybrid because asymmetric key is public but it's computationally expensive, while symmetric key cryptography is much faster but suffers from the key distribution problem (keys must be secret)
  - SSH is used to securely encrypt your password login and terminal traffic during a shell session (can be used to tunnel other data too)
  - SSL/TLS is used to securely encrypt http Web traffic – such securely protected Web pages are labeled https

Authentication

- Logging into your own computer
  - Type in your password. OS checks if it matches your account's password, and if so then you're authenticated and allowed to proceed

Authentication

- Key Problem: How does OS store passwords?
- Plaintext
  - Anyone with root privilege can steal passwords
  - What if the system is compromised?
- Encrypted
  - What if the encryption key is compromised?
- Key observation
  - There is no need to be able to decrypt a stored password value

Authentication

- Hashed passwords
  - Use one-way functions (cryptographic hash functions) that map an input string of arbitrary size to a fixed size output string
  - Given a hashed value, there is no way to compute an input string that maps to that hashed value, i.e. infeasible to invert
  - Hashed values is often referred to as *message digest* or *authenticator*
  - Examples: MD4, MD5, SHA-1, SHA-2, etc.

Authentication

- OS stores only the hashed values
- When a user enters a password, OS hashes that password and then compares it with the stored hashed value
  - If match, access allowed
  - If no match, access denied

Authentication

- Attack against hashed passwords
  - Attacker can try to guess your password, using common words, etc.
- OS solutions
  - OS can block or slow down access after too many failed login attempts
  - OS can force users to change their passwords frequently
  - OS can force users to choose hard-to-guess passwords

Linux password mechanism

- Traditional Unix systems keep user account information, including hashed passwords, in a text file called /etc/passwd
- This file is used by many tools, e.g. *ls* to display file ownerships, etc. by matching user id #'s with the user's names
  - the file needs to be world-readable → security risk
- In earlier versions, hashed values were world readable
  - makes it easier to launch a brute-force attack

Shadow Password

- Store account information in the /etc/passwd file in a compatible format. However, the password is stored as a single "x" character
  - /etc/passwd file is still world readable
- A second file, called /etc/shadow, contains hashed password as well as other information such as account or password expiration values, etc.
  - The /etc/shadow file is readable only by the root account

Shadow Passwords

- /etc/passwd:
  - smithj:x:561:561:Joe Smith:/home/smithj:/bin/bash
- /etc/shadow:
  - smithj:Ep6mckrOLChF.:10063:0:99999:7:::
- Cryptographic hash algorithm
  - Depends on distribution: MD5, SHA 1, …
  - Password is used as a key to hash a text consisting of all zeros

Authentication

- What if you want to remotely login into a computer?
  - Can't send your password in the clear, or else an eavesdropper could overhear it and then replay it later in a *replay attack*
  - Send an encrypted password. But even an encrypted password can be replayed.
  - So send an encrypted password that changes each time – let's look at how SSH achieves this

SSH

- Secure Shell (SSH) employs a hybrid public key/symmetric key approach
  1. Requestor (user trying to remotely login) asks for requestee's (server's) public key
     - Note user is asked whether to accept this public key if this is the first time this server was encountered. This is the primary verification of the public key. This public key is then saved locally and compared to the key returned during later login attempts to the same server.
  2. SSH requestor chooses a symmetric session key K and encrypts it with the server's public key, sending it as the 1st message
     - Thus, both sides of the connection securely receive a symmetric key K – solves the symmetric key distribution problem
     - SSH-2 has both sides negotiate a symmetric key K, rather than the requestor just choosing the key directly

SSH

User/Client

Server

Server's Keys: K_{PUBLIC}, $K_{PRIVATE}$

Login?

K_{PUBLIC}

Ask user if they will accept K_{PUBLIC}. If so, choose symmetric session key K[1]

$E(K_{PUBLIC},K)$[2]

Decrypt session key K with $K_{PRIVATE}$:

$K = D(K_{PRIVATE},E(K_{PUBLIC},K))$

Encrypt password and messages

E(K,message)

Decrypt message with session key K

[1]In SSH-2, user and server negotiate a session key, e.g. using Diffie-Hellman, rather than the user just choosing a session key

[2]Note E(k,d) means some data d is encrypted by key k

SSH

- After step 2, all subsequent messages (including the password) are encrypted with symmetric key K
  - Symmetric key encryption is much faster than asymmetric key encryption, so asymmetric keys are only used at the initial phase to safely distribute the symmetric key
  - Since the key K changes for every login session, then even typing the same password will still result in a different encrypted message, so there is no replay attack possible

SSH

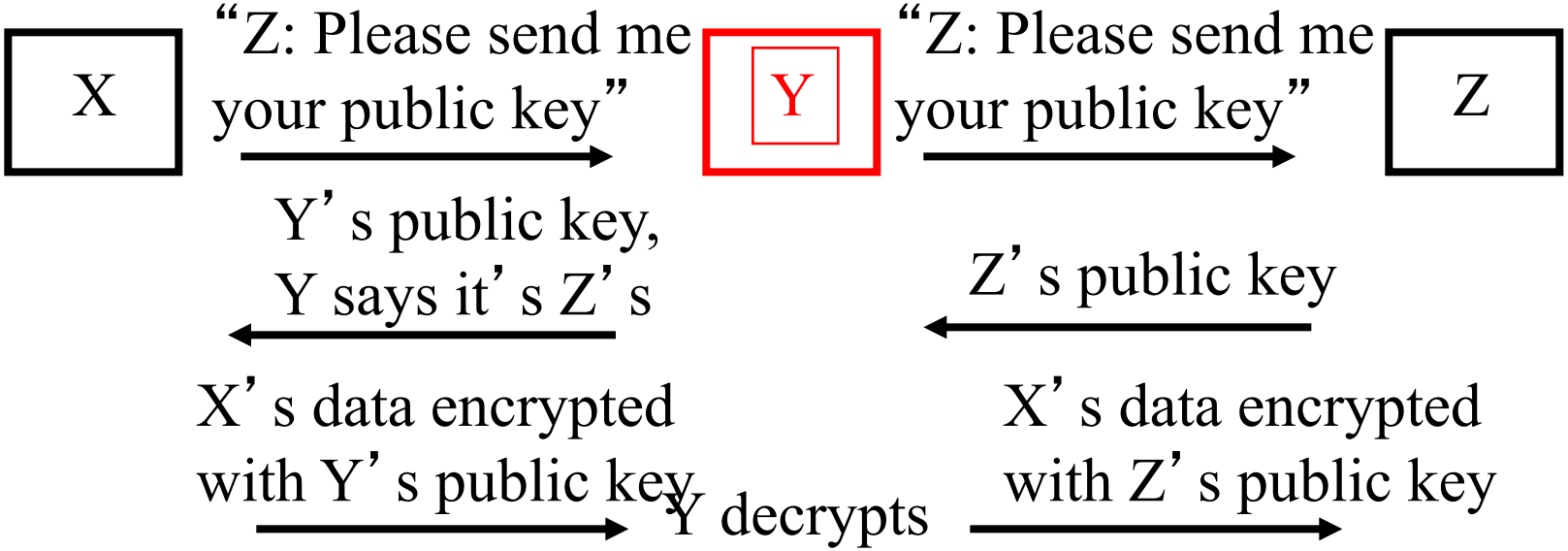
- This approach solves
  - confidentiality: password is always encrypted with a symmetric key
  - key distribution: public keys are used to distribute symmetric keys
  - speed: most data is encrypted with fast symmetric keys. Only the initial handshaking phase is encrypted using slow asymmetric keys.
  - replay attack: the encrypted password changes each time (though the entered password doesn't change) because the symmetric key changes every session

SSH

- Problem
  - Susceptible to *man-in-the-middle* attacks, because the initial public key is not certified – see next slide
    - SSH's typical model is trust on first access, hoping there is not a bad guy the first time. If there was a man-in-the-middle during first access, then it could steal your password and view the contents of the session, all without being noticed! It could also view future sessions.
  - There is a provision for certificates in SSH-2 (OpenSSH has a patch), but it is not widely used yet
    - Servers probably don't provide such certificates because it's somewhat difficult to obtain certification from one of the trusted 3rd parties

Man-in-the-middle Attack

- Public key distribution with uncertified public keys is subject to a *man-in-the-middle attack*
  - suppose X wants Z's public key, and Y is the man in the middle



X and Z never know that Y has seen their data

Solution: the public key must be signed or certified by a trusted 3rd party, e.g. Verisign or GoDaddy

Certified Public Keys

- When you receive a message M advertising a public key from X, you also receive a certificate

  M = [public key of X, certificate of public key of X]

- This certificate contains a digital signature signed by a trusted 3rd party (also called the certificate authority (CA))

  Certificate = E(Private_Key$_{\text{trusted 3rd party}}$, public key of X)

  – Certificate is the encryption of the public key of X (plus other info) with the private key of the trusted 3rd party
  – Your browser is predistributed with the public keys of trusted 3rd parties like Verisign embedded within it – this avoids the man-in-the-middle attack
  – To verify the public key of X, decrypt the attached certificate with the public key of the 3rd party (which you have), and if it matches the advertised key, then you know the key is certifiably authentic and couldn't have been faked

Certified Public Keys

- Note that this approach depends on the following
  - $D[K_{PUBLIC}(X),E(K_{PRIVATE}(X),m)] = m$
  - That is, it is OK in RSA to *switch* the order of exponentiation, so that we use the private key to encrypt/“sign”/exponentiate m first, then use the public key to decrypt/exponentiate m later.

SSL/TLS

- Secure Sockets Layer (SSL), renamed to Transport Layer Security (TLS), is the basis of the secure Web protocol *https*
- In SSL (or TLS)
  1. Requestee's *certified* public key is initially passed to requestor, who verifies certificate
  2. SSL requestor negotiates a symmetric session key K with requestee
     - Negotiation is protected by encrypting it with certified public key
  3. All subsequent messages (e.g. secure Web page contents) are encrypted with symmetric key K

SSL/TLS

User/Client

Server

Server's Keys: K_{PUBLIC}, $K_{PRIVATE}$

Login?

$[K_{PUBLIC}, certificate(K_{PUBLIC})]$

Verify K_{PUBLIC} with certificate. If OK, generate random premaster secret.
K = f(premaster secret)

$E(K_{PUBLIC}, premaster\ secret)$

Decrypt premaster secret with $K_{PRIVATE}$: session key K = f(premaster secret)

Send messages

E(K,message)

Decrypt message with session key K

SSL/TLS

- Key difference with SSH is that SSL/TLS uses certified public key
- SSL/TLS's approach solves
  - confidentiality: Web page is always encrypted with a symmetric key
  - key distribution: public keys are used to distribute symmetric keys
  - speed: most data is encrypted with fast sym. keys
  - replay attack: the encrypted Web page changes every time because the symmetric key changes every session
  - man-in-the-middle attack: because the initial public key is certified (improvement over SSH)

Authentication

- Other non-SSH solutions to replay attacks during password login
  - Use a new one-time password for each login
    - A list of one-time passwords is generated a priori and then consulted during login at both ends
    - Alternatively a list could be generated using a one-way function →one way hash chain
  - Use a challenge-response type protocol
    - The authenticator sends a random number N used only once, or *nonce*, to the authenticatee. This N could be in plaintext or in the clear.
    - The authenticatee encrypts the password and nonce with the shared symmetric key K, i.e. send message M = E(K, password+nonce)
    - Since the nonce changes every time, then the message M changes every time and cannot be replayed
    - added benefit is that the human user can type in the same password for each login

Defense In Depth

- We've seen
  - Secure authentication via ssh for secure login
  - encrypted files
  - authorization to protect files
- Other defenses
  - Virus scanners,
  - Personal firewall in your OS
    - Disables most ports, filters traffic on open ports, etc.
  - Intrusion detection systems (IDS)
    - Analyze behavior of network traffic and code for attack signatures, generate a warning if found

Defense In Depth

- Standard security philosophy is *defense-in-depth*: employ multiple layers of security
- To design multiple layers of defenses, you have to define what attacks you're defending against, i.e. what is the threat model?
  - Define the resources that the attacker has available to them
    - Is it a single attacker, or multiple distributed colluding attackers?
    - Does the attacker have a laptop or a supercomputer?
  - Define your system model – what resources do you have available to you for defense?
  - Define the specific type or types of attacks that you're defending against, e.g.
    - Eavesdropping attacks
    - Replay attacks
    - Man-in-the-middle attacks
    - Cryptanalysis attacks, e.g. frequency analysis, etc.