Network Security

Why Network Security?

Malicious people share your network
People who want to snoop
People who want to destroy
People who want to corrupt
People who want to pretend
People who want to steal
Problem made more severe as Internet becomes more commercialized
Active and passive attacks

Dealing with Network Security

Network security can be broken into several related areas:

Authentication: Proving that someone is who she says she is
Identity verification (use something you know, you have, you are)

Confidentiality: hiding data from prying eyes
Keep data secret from unauthorized users

Integrity: Proving that a message has not been intentionally corrupted by a third party
No tampering by someone in the middle

Non repudiation: Once someone sends a message, or conducts a transaction, she can’t later deny the contents of that message
Undeniability

Available Security Tools

Cryptography/Encryption:
Encode a message in a way that only the communicating parties can interpret it
Used for secrecy and authentication

Signatures
Allow for the authentication of a message’s sender and the message’s integrity
Used for authentication, nonrepudiation and integrity control
Cryptography

Encoding a message in a way that only the communicating parties can interpret it

Notation:

Encryption:  \( C = E_k(P) \)

Decryption:  \( P = D_k(C) \)

Rules for Encryption

Encryption requires both an encryption algorithm and an encryption “key”

Algorithm:

- Should be public and known to all
- Inspires trust that the algorithm works

Keys:

- Should be long enough to prevent easy breaking of the encryption
- Should be short enough to keep algorithm efficient

Typical key lengths: 56-bit, 128-bit, 256-bit, 512-bit

Types of Encryption Algorithms

Substitution Ciphers

Every letter (or group of letters) is replaced by another letter (or group of letters)

Example:

Caesar cipher:

- a\( \rightarrow \) D, b\( \rightarrow \) E, c\( \rightarrow \) F, d\( \rightarrow \) G, ... z\( \rightarrow \) C
- Each letter is replaced by a shift Succ(1), Succ(2), pred(1), pred(2)

Monoalphabetic cipher:

- a\( \rightarrow \) Q, b\( \rightarrow \) W, c\( \rightarrow \) E, ...
- Need a table to provide the mapping

Easy to break by analyzing statistical properties of written language

Polyalphabetic encryption

Use a set of monoalphabetic substitutions in a cycle \( S_1, S_2, S_3 \)

- \( N = 3 \), LADYGAGA
- L is replaced by I according to \( S_1 \)
- A is replaced by D according to \( S_2 \)
- Y is replaced by O according to \( S_1 \)
- G is replaced by T according to \( S_2 \) ... And so on

- Key is the 3 different monoalphabetic ciphers
Types of Encryption

Transposition Ciphers

Instead of substituting letters in the plaintext, we change their order.

**Key** = ANDREW

**Plaintext** = thisisamessageiwouldliketocryptnow

**Ciphertext** = tiihssaesmsgioewulldliktedefctynowl

- Also easy to break by analyzing structure of language

Encryption in practice

Most actual encryption algorithms use a complex combination of substitution and transposition.

Examples:

- Data Encryption Standard (DES)
  - Multiple iterations of substitution and transposition using a 56-bit key designed by IBM with input from the NSA
- DES chaining
  - Multiple stages of DES coding, in which the input of each stage is the output of previous stages
- International Data Encryption Algorithm (IDEA)
  - Uses a 128-bit key

Symmetric key cryptography

- **symmetric key** crypto: Bob and Alice share same (symmetric) key: K
- e.g., key is knowing substitution pattern in monoalphabetic substitution cipher
- How do Bob and Alice agree on key value?
- **Answer**: use out-of-band secure channels
Two types of symmetric ciphers

Stream ciphers
- encrypt one bit at a time

Block ciphers
- Break plaintext message in equal-size blocks
- Encrypt each block as a unit

Stream Ciphers

Stream ciphers combine each bit of keystream with bit of plaintext to get bit of ciphertext.

\[
\begin{align*}
\text{c(i)} & = \text{ks(i)} \oplus \text{m(i)} \\
\text{m(i)} & = \text{ks(i)} \oplus \text{c(i)}
\end{align*}
\]

RC4 Stream Cipher

- RC4 is a popular stream cipher
- Extensively analyzed and considered good
- Key can be from 1 to 256 bytes
- Used in WEP for 802.11
- Can be used in SSL

Block ciphers

Message to be encrypted is processed in blocks of k bits (e.g., 64-bit blocks).

1-to-1 mapping is used to map k-bit block of plaintext to k-bit block of ciphertext.

Example with k=3:

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>110</td>
</tr>
<tr>
<td>001</td>
<td>111</td>
</tr>
<tr>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>011</td>
<td>100</td>
</tr>
</tbody>
</table>

What is the ciphertext for 010110001111?
Block ciphers

How many possible mappings are there for \( k=3 \)?
How many 3-bit inputs?
How many mappings or permutations of the 3-bit inputs? \( 8! \)
Answer: 40,320; not very many!
In general, \( 2^k \) mappings; huge for \( k=64 \)

Problem:
Table approach requires table with \( 2^{64} \) entries, each entry with 64 bits
Table too big; instead use function that simulates a randomly permuted table

Symmetric key crypto: DES

DES: Data Encryption Standard
US encryption standard [NIST 1993]
56-bit symmetric key, 64-bit plaintext input
Block cipher with cipher block chaining
How secure is DES?
DES Challenge: 56-bit-key-encrypted phrase decrypted (brute force) in less than a day
No known good analytic attack
making DES more secure:
3DES: encrypt 3 times with 3 different keys
(actually encrypt, decrypt, encrypt)

Triple DES

Use three keys and three executions of the DES algorithm (encrypt-decrypt-encrypt)

\[ C = E_{K3}(D_{K2}(E_{K1}(P))) \]

Example

\[ P = D_{K1}(E_{K3}(D_{K2}(E_{K1}(P)))) \]
Symmetric key crypto: DES

**DES operation**
- Initial permutation
- 16 identical "rounds" of function application, each using different 48 bits of key
- Final permutation

**Encryption using symmetric keys**
- Same key for encryption and decryption
- Achieves confidentiality
- Vulnerable to tampering
- Bad guy can alter the encrypted message
- What about authentication?
- Vulnerable to replay attacks
- Bad guy can steal the encrypted message and later present it on behalf of legit user

---

**Replay attack**

```
Alice

```

```
Bob
```

Alice's password is encrypted
- From both Bob and attackers
- If Bob is trusted, he can decrypt using the same key
- But still subject to replay attack

```
Replay Attack

```

```
Alice

```

```
Bob
```

```
"Login: Alice"
```

```
Password please
```

```
E_K(Alice's password)
```

```
Alice's password is encrypted
```

```
Trudy
```

```
Bob
```

```
"Login: Alice"
```

```
Password please
```

```
E_K(Alice's password)
```

```
Store: E_K(Alice's password)
```
Replay Attack

This is a replay attack.
How can we prevent a replay?
By adding a NONCE value; Number used once only a temporary random number.

Login: Alice
Password please
E_K(Alice’s password)

Trudy
Bob

Challenge-Response

Login: Alice
Password please: Nonce
E_K(Alice’s password, Nonce)

Alice
Bob

- Nonce is the challenge
- The encrypted msg is the response
- Nonce changed every time
- Prevents replay, insures freshness
- Password is known only to Alice and Bob
- Even if Trudy steals the encrypted Nonce value, at next login it will not work

Types of Encryption

Problem with all the cryptography algorithms used so far:
- if the key is stolen, any message can be decrypted
Is there a way to do cryptography without worrying about this concern?

Yes, public key cryptography

Public Key Cryptography

Uses two different keys: an encryption key and a decryption key
Each user holds:
- a private decryption key to decrypt messages sent to her
- a public encryption key that everyone else should use to encrypt messages that are sent to her
Important property of public key cryptography algorithms:
- Private decryption key cannot be easily determined by knowing the public encryption key
Public key cryptography

Given \((n, e)\) and \((n, d)\)

1. To encrypt message \(m < n\), compute
   \[ c = m^e \mod n \]

2. To decrypt received bit pattern, \(c\), compute
   \[ m = c^d \mod n \]
Public Key Cryptography

An Example

If Jeff wants to send an encrypted plaintext message $P$ to Sally, he uses Sally’s public key to encrypt the message into $C$.

$$C = E_{K_{pub, sally}}(P)$$

Sally uses her private key to decrypt the message $C$ from Jeff. Only Sally can decrypt messages that are encrypted using her public key. A message to Sally cannot be decrypted using Sally’s public key.

$$P = D_{K_{priv, sally}}(C)$$

Authentication

Authentication: the problem of proving that a user is who he says he is

Two approaches:
- based on secret key cryptography
- based on public key cryptography

Secret Key Authentication

Sally and Jeff share a secret key $K$

Sally and Jeff issue authentication challenges

Identity=Sally

Challenge $R_{Jeff}$

$E_K(R_{Jeff})$

Challenge $R_{Sally}$

$E_K(R_{Sally})$
Public Key Authentication

Sally and Jeff issue authentication challenges using public keys

Efficiency issue with Public Key Cryptography

Public key encryption methods tend to be slower than secret key encryption, because they rely on more expensive computations. So, it is common in practice to use public key cryptography only to encode a private secret key ($K_s$), which is then used to encrypt all other data shared between the two communicating parties.

Public Key Cryptography (cont’d)

Why public key cryptography is so powerful:

- No secret keys need to be distributed
- Only the receiver of encrypted information holds the secret key

Examples of public key algorithms:

- Merkle-Hellman knapsack
- Rivest-Shamir-Adleman (RSA)
- Pretty Good Privacy (PGP)

Digital Signatures

A digital signature...

- allows the receiver to authenticate the identity of the sender
- prevents the sender from later claiming she sent a different (or no) message
- prevents the receiver from constructing a message that appears as if it came from the sender
Public Key Signatures

Let us use a public key cryptography system with the following properties:

\[ D_{pr}(E_{pu}(M)) = M \]
\[ D_{pu}(E_{pr}(M)) = M \]

**Message Integrity**

allows communicating parties to verify that received messages are authentic:
- Content of message has not been altered
- Source of message is who/what you think it is
- Message has not been replayed
- Sequence of messages is maintained

let's first talk about message digests

**Message Digests**

function \( H() \) that takes as input an arbitrary length message and outputs a fixed-length string:
- "message signature"

note that \( H() \) is a many-to-1 function
- \( H() \) is often called a "hash function"

desirable properties:
- easy to calculate
- irreversibility: Can't determine \( m \) from \( H(m) \)
- collision resistance: Computationally difficult to produce \( m \) and \( m' \) such that \( H(m) = H(m') \)
- seemingly random output
Internet checksum: poor message digest

Internet checksum has some properties of hash function:
✓ produces fixed length digest (16-bit sum) of input
✓ is many-to-one
♦ but given message with given hash value, it is easy to find another message with same hash value.
  * e.g., simplified checksum: add 4-byte chunks at a time:

<table>
<thead>
<tr>
<th>message</th>
<th>ASCII format</th>
</tr>
</thead>
<tbody>
<tr>
<td>I O U I</td>
<td>49 4F 55 31</td>
</tr>
<tr>
<td>0 0 . 9</td>
<td>30 30 2E 39</td>
</tr>
<tr>
<td>9 B O B</td>
<td>39 42 D2 42</td>
</tr>
</tbody>
</table>

Message Authentication Code (MAC)

```
message
<table>
<thead>
<tr>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>H( )</td>
</tr>
</tbody>
</table>
```

```
message
<table>
<thead>
<tr>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
</tr>
</tbody>
</table>
```

```
message
<table>
<thead>
<tr>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>H()</td>
</tr>
</tbody>
</table>
```

```
message
<table>
<thead>
<tr>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>compare</td>
</tr>
</tbody>
</table>
```

Basic idea

Use message as key and transform a constant string of length N repeatedly into another string of length N which is the digest.

Simple example: XOR the constant string with the message bytes

In practice, use a set of boolean operations

- String: 00000000
- D: 01000100
- J: 01001010
- T: 01010100
- Digest: 01011010

Hash Function Algorithms

MD5 hash function widely used (RFC 1321)
computes 128-bit message digest in 4-step process.
SHA-1 is also used.
US standard [NIST, FIPS PUB 180-1]
160-bit message digest
**Digital Signatures**

A cryptographic technique analogous to handwritten signatures. Sender (Bob) digitally signs document, establishing he is document owner/creator. Goal is similar to that of MAC, except now use public-key cryptography.

**Verifiable, Nonforgeable:** Recipient (Alice) can prove to someone that Bob, and no one else (including Alice), must have signed document.

---

**HMAC**

A popular MAC standard addresses some subtle security flaws. Operation:

- Concatenates secret to front of message.
- Hashes concatenated message.
- Concatenates secret to front of digest.
- Hashes combination again.

---

**Digital Signatures**

Simple digital signature for message m:

Bob signs m by encrypting with his private key $K_B$, creating "signed" message, $K_B(m)$.

Bob sends digitally signed message:

Alice verifies signature and integrity of digitally signed message:

Digital signature = signed message digest

Bob's message, m

Bob's private key $K_B$

Public key encryption algorithm

Bob's message, m, signed (encrypted) with his private key $K_B$

H: Hash function

H(m)

Bob's public key

encrypted msg digest $K_B(H(m))$

equal?

Alice verifies signature and integrity of digitally signed message.

Network Security
Digital Signatures (more)

suppose Alice receives msg m, digital signature $K_B(m)$

Alice verifies m signed by Bob by applying Bob’s public key $K_B$ to $K_B(m)$ then checks $K_B(K_B(m)) = m$.

if $K_B(K_B(m)) = m$, whoever signed m must have used Bob’s private key.

Alice thus verifies that:

- Bob signed m.
- no one else signed m.
- Bob signed m and not m'.

Non-repudiation:

- Alice can take m, and signature $K_B(m)$ to court and prove that Bob signed m.

Public-key certification

motivation: Trudy plays pizza prank on Bob

Trudy creates e-mail order:

Dear Pizza Store, Please deliver to me four pepperoni pizzas. Thank you, Bob

Trudy signs order with her private key

Trudy sends order to Pizza Store

Trudy sends to Pizza Store her public key, but says it’s Bob’s public key.

Pizza Store verifies signature; then delivers four pizzas to Bob.

Bob doesn’t even like Pepperoni

Certification Authorities

Certification authority (CA): binds public key to particular entity, E.

E (person, router) registers its public key with CA.

E provides “proof of identity” to CA.

CA creates certificate binding E to its public key, certificate containing E’s public key digitally signed by CA — CA says “this is E’s public key”

when Alice wants Bob’s public key:

gets Bob’s certificate (Bob or elsewhere).

apply CA’s public key to Bob’s certificate, get Bob’s public key
Certificates: summary

primary standard X.509 (RFC 2459)
certificate contains:
  issuer name
  entity name, address, domain name, etc.
  entity’s public key
digital signature (signed with issuer’s private key)

Public Key Infrastructure (PKI)
certificates, certification authorities
often considered “heavy”

Message Integrity

Original Message content Has not been changed/tampered with
Use Message Digest
One-way hash functions
F(M) → D
Given D, it is impossible to find M
Also, it is not possible to find another M’ such that F(M’) = D
Even a single bit change to M will result in a different digest
Two well-known schemes
MD5 and SHA-1

Message Integrity

MD5: Message Digest uses a 128-bit message digest
  Break the message into 512 bit blocks
    From an initial 128 bit (four 32-bit words) key compute the hash for the first block, then use this hash as the seed for the next block and so on until the final value which is the digest
SHA-1: Secure Hash uses a 160-bit digest
  Same idea as MD5 but uses 160-bit (five 32-bit words)
  Still a function iteratively applied to successive 512 bit blocks of the message
  Output hash value is the final buffer value

Secure Socket Layer SSL

SSL: used in authentication of client and server
A server sends a certificate
Client verifies the certificate and uses public key to send encrypted data to the server
Https is the protocol that enables secure communication between client and server
SSL Architecture

<table>
<thead>
<tr>
<th>SSL Handshake Protocol</th>
<th>SSL Change Cipher Spec Protocol</th>
<th>SSL Alert Protocol</th>
<th>HTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SSL Record Protocol</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TCP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Toy SSL: a simple secure channel

**handshake:** Alice and Bob use their certificates, private keys to authenticate each other and exchange shared secret.

**key derivation:** Alice and Bob use shared secret to derive set of keys.

**data transfer:** data to be transferred is broken up into series of records.

**connection closure:** special messages to securely close connection.

MS = master secret
EMS = encrypted master secret

SSL Protocol

Client

- Client initiates a connection
- Client verifies the server’s ID by using servers public key. If requested by the server, the client sends its ID.
- When the authentication is complete, the client sends a session key encrypted using the server’s public key.
- Once a session key is established, secure communications commence between client and server.

Server

- Server responds by sending the client its certificate, identity signed by its private key. The server may also, optionally, request the client’s ID for client authentication.
- Server Certificate
- Client certificate
- Session key
Considered bad to use same key for more than one cryptographic operation
use different keys for message authentication code (MAC) and encryption

four keys:
\[ K_c = \text{encryption key for data sent from client to server} \]
\[ M_c = \text{MAC key for data sent from client to server} \]
\[ K_s = \text{encryption key for data sent from server to client} \]
\[ M_s = \text{MAC key for data sent from server to client} \]

depend: derived from key derivation function (KDF)
takes master secret and (possibly) some additional random data and creates the keys

**Toy: Key derivation**

why not encrypt data in constant stream as we write it to TCP?
where would we put the MAC? If at end, no message integrity until all data processed.
E.g., with instant messaging, how can we do integrity check over all bytes sent before displaying?

instead, break stream in series of records
Each record carries a MAC
Receiver can act on each record as it arrives

issue: in record, receiver needs to distinguish MAC from data
want to use variable-length records

**Toy: Data Records**

why not encrypt data in constant stream as we write it to TCP?
where would we put the MAC? If at end, no message integrity until all data processed.
E.g., with instant messaging, how can we do integrity check over all bytes sent before displaying?
instead, break stream in series of records
Each record carries a MAC
Receiver can act on each record as it arrives

issue: in record, receiver needs to distinguish MAC from data
want to use variable-length records

**Toy SSL isn’t complete**

how long are fields?
which encryption protocols?
want negotiation?
allow client and server to support different encryption algorithms
allow client and server to choose together specific algorithm before data transfer
SSL Cipher Suite

- Cipher suite
- Public-key algorithm
- Symmetric encryption algorithm
- MAC algorithm

SSL supports several cipher suites

Negotiation: client, server agree on cipher suite

- Client offers choice
- Server picks one

Common SSL symmetric ciphers

- **DES** - Data Encryption Standard: block
- **3DES** - Triple strength: block
- **RC2** - Rivest Cipher 2: block
- **RC4** - Rivest Cipher 4: stream

SSL Public key encryption

- **RSA**

Real SSL: Handshake (1)

**Purpose**

1. Server authentication
2. Negotiation: agree on crypto algorithms
3. Establish keys
4. Client authentication (optional)

Real SSL: Handshake (2)

1. Client sends list of algorithms it supports, along with client nonce
2. Server chooses algorithms from list; sends back: choice + certificate + server nonce
3. Client verifies certificate, extracts server’s public key, generates pre_master_secret, encrypts with server’s public key, sends to server
4. Client and server independently compute encryption and MAC keys from pre_master_secret and nonces
5. Client sends a MAC of all the handshake messages
6. Server sends a MAC of all the handshake messages

Real SSL: Handshaking (3)

Last 2 steps protect handshake from tampering
Client typically offers range of algorithms, some strong, some weak
Man-in-the-middle could delete stronger algorithms from list
Last 2 steps prevent this

Last two messages are encrypted
Real SSL: Handshaking (4)

why two random nonces?
suppose Trudy sniffs all messages between Alice & Bob
next day, Trudy sets up TCP connection with Bob, sends exact same sequence of records
Bob (Amazon) thinks Alice made two separate orders for the same thing
solution: Bob sends different random nonce for each connection. This causes encryption keys to be different on the two days
Trudy’s messages will fail Bob’s integrity check

SSL Record Protocol

record header: content type; version; length
MAC: includes sequence number, MAC key M_k
fragment: each SSL fragment 2^14 bytes (~16 Kbytes)

SSL Record Format

1 byte 2 bytes 3 bytes
content type SSL version length

data

MAC

data and MAC encrypted (symmetric algorithm)

Real Connection

TCP Fin follow

Everything henceforth is encrypted

handshake: ClientHello
handshake: ServerHello
handshake: Certificate
handshake: ServerHelloDone
handshake: ClientKeyExchange
ChangeCipherSpec
handshake: Finished
application_data
Alert: warning, close_notify
Key derivation

client nonce, server nonce, and pre-master secret input into pseudo random-number generator. produces master secret
master secret and new nonces input into another random-number generator: "key block"
Because of resumption: TBD
key block sliced and diced:
  client MAC key
  server MAC key
  client encryption key
  server encryption key
  client initialization vector (IV)
  server initialization vector (IV)

Public–key certificate

Certificate consists of a public key and a user ID of the owner
Signed by the private key of certificate authority
Receiver of certificate can verify owner by using the signature of the CA
SSL

What’s in a Certificate

Legal owner of the certificate
May be a chain of certificates leading to a well-known provider
The physical location of owner (city, state, country, etc)
Valid dates for the certificate
Server’s public key

HTTPS

HTTPS (HTTP over SSL)
combination of HTTP & SSL/TLS to secure communications between browser & server
documented in RFC2818
no fundamental change using either SSL or TLS
use https:// URL rather than http://
and port 443 rather than 80
encrypts
  URL, document contents, form data, cookies, HTTP headers