# EECS 388 Course Notes

Janani Chinnam

April 24, 2017

## 1 Introduction

### 1.1 The security mindset

- Thinking like an attacker: understanding techniques for circumventing security.
- Thinking like a defender: knowing what you're defending, and against who - rational paranoia.

### 1.2 The insecurity hierarchy

- Level 2 problem: Weakness: Factors that predispose a system to vulnerability.
- Level 1 problem: Vulnerability: Specific errors that could be exploited in an assault.
- Level 0 problem: Assault: Actual malicious attempt to cause harm.

### 1.3 Properties to enforce

- Confidentiality
- Integrity
- Availability
- Privacy
- Authenticity

## 2 Integrity

Message integrity: an attacker can not modify messages without being *detected*.

## 2.1 Man-in-the-middle

Alice $\rightarrow$ **Mallory** $\rightarrow$ Bob

**Threat model:** Mallory can see, modify, and forge messages.

**Approach:** Alice computes $v = f(m)$, Bob verifies $v' = f(m')$

- $f$: easily computable by Alice/Bob, not by Mallory
- Random function: secure, but impractical
- Solution: pseudo-random function (**PRF**)

## 2.2 Building a PRF

**Kerckhoff's Principle**: A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.

1. We flip a coin secretly to get bit $b$
2. If $b = 0$, let $g$ be a random function. If $b = 1$, let $g = f_k$, where $k$ is a randomly chosen secret
3. Mallory chooses $x$; we announce $g(x)$
4. Mallory guesses $b$ quickly

**Security definition**: We say $f$ is a secure PRF if Mallory can't select $b$ any better than random guessing.

Message Authentication Code (**MAC**): Effectively the same thing as a PRF
Currently popular "PRF": **HMAC-SHA256** (Cryptographic hash function, see 2.4)

## 2.3 Hash functions

**Input**: Arbitrary Length Data
**Output**: Fixed-size digest (n bits)
**Properties**: No key, fixed function
**Examples**: SHA-256, SHA-512, SHA-3

**Qualities of Good Hash Functions:**

- Collision Resistance: Few pair of input x, x' such that H(x) = H(x')
- Second Pre-image Resistance: Given x, few x' such that H(x) = H(x')
- Pre-image Resistance: Given y, few x' such that H(x') = y

**Hash function failures**: MD5 (Broken in 2004, now easy to find collisions), SHA-1 (unsupported by HTTPS in 2016)

### 2.4 SHA-256

**Input**: Arbitrary Length Data
**Output**: 256-bit digest
**Properties**: Built with "compression function" $h$ using Merkle–Damgård

**The algorithm...**

1. Pad input $m$ to multiple of 512 bits (using a fixed algorithm)
2. Split into 512-bit blocks $b_0, b_1, ...b_{n-1}$
3. $y_0$ = constant initialization vector., $y_1 = h(y_0, b_0), ..., y_i = h(y_{i-1}, b_{i-1})$
4. Return $y_n$

**Problem**: Stand alone, a hash function is (usually) vulnerable to length extension attacks (Given $z = H(m)$ for some unknown m, calculate $H(m||$ padding $||v)$ for attacker selected $v$).

## 3 Randomness and Pseudo-Randomness

**True Randomness**: Output of a physical process that is inherently random (scarce and hard to get).

### 3.1 Pseudo-random Generator (PRG)

Takes small **seed** that is actually **random** and generates long sequence of numbers that is 'as good as random'. Considered **secure** if indistinguishable from random.

**Definition (similar to PRF):**

1. We flip a coin secretly to get a bit b
2. If $b = 0$, let $s$ be a truly random stream If $b = 1$, let $s$ be $g_k$ for random secret $k$.
3. Mallory can see as much of the output of $s$ as they want.
4. Mallory guesses $b$, wins if guesses correctly.

**Security Definition**: $g$ is a secure PRG if there is no winning strategy for Mallory.

**Theorem**: If $f$ is a secure PRF, and $g$ is built from $f$ by this construction, then $g$ is a secure PRG.

## 4 Confidentiality

Message confidentiality: Keep contents of message $p$ secret from an *eavesdropper*.

Alice $\rightarrow$ **Eve** $\rightarrow$ Bob

**Some terminology:** p (plain-text), c (cipher-text), k (secret key), E (encryption function), D (decryption function)

## 4.1 Caesar Cipher

Replaces each plain-text letter with one a fixed number of places down the alphabet.

**Cryptoanalysis:**

- Could try all 26 $k$ by brute force
- **Frequency analysis** (faster): English text has distinctive letter frequency distribution

## 4.2 Vigenere Cipher

Encrypts successive letters using a **sequence of Caesar ciphers** determined by the letters of a keyword.

**Cryptoanalysis:**

- First, find keyword length $n$: **Kasiski Method**
  - Repeated strings in long plain-text will sometimes, by coincidence, be encrypted with same key letters
  - Distance between repeated strings in the cipher-text is likely a multiple of key length
- If we know $n$:
  - Break ciphertext into $n$ slices
  - Solve each slice as a Caesar cipher individually

## 4.3 Stream Cipher

**One Time Pad (OTP)**: Alice and Bob jointly generate a secret, very long, string of random bits (the one-time pad, **k**)

- To encrypt: $c_i = p_i$ xor $k_i$
- To decrypt: $p_i = c_i$ xor $k_i$

Never reusing the pad, this is provably **secure**, while impractical.

**Stream cipher**: use a PRG instead of a truly random pad (never sharing key or generator output bits).

1. Start with shared secret key k
2. Alice and Bob each use k to seed the PRG

3. To encrypt, Alice XORs next bit of her generator's output with next bit of plaintext
4. To decrypt, Bob XORs next bit of his generator's output with next bit of cipher-text

## 4.4 Block Cipher

Functions that encrypt fixed-size blocks with a reusable key; inverse function decrypts when used with the same key.

**Pseudo-Random Permutation (PRP)**: A secure PRP takes as input seed k, outputs a permutation that is practically indistinguishable from truly random permutation unless you know k.

**The Challenge:** Design a 'hairy' function, only invertible if you know k

- Highly nonlinear ("confusion")
- Mixes input bits together ("diffusion")
- Depends on the key

## 4.5 AES: Advanced Encryption Standard

Designed and standardized by NIST competition, long public comment/discussion period. Widely believed to be secure, no proof. Variable key and block size (128 or 256, key length matches block size).

**Construction:** "Round-based" with 10 rounds

- Split **k** into 10 subkeys
- Perform set of operations 10 times
- Each time use a different subkey

## 4.6 Counter Mode (CTR)

Cipher modes: how do we handle multi block messages? Counter Mode, uses block cipher as a PRG.

- message id = random data
- XOR $i$th block of message with $E_k$(message id || ctr)
- Effectively a stream cipher

## 4.7 Building A Secure Channel, so far

Encrypt, then MAC.

- Need 2+ shared keys but only have one? ***PRGs***
- Reverse channel (Bob to Alice)? ***seperate keys***

Performing *any* cryptographic operations before verifying the MAC on a message you've received inevitably leads to **doom**.

### 4.8 AEAD: Authenticated Encryption with Associated Data

**Input**: $p$, $k$, and optional header (plaintext, covered by MAC)
**Output**: $c$, MAC

- ciphertext, auth tag := Seal(key, plaintext, associated data)
- plaintext:= Unseal(key, ciphertext, associated data, tag)

Commonly used: AES-GCM

### 4.9 Padding Oracles

Distinguish between invalid MAC and invalid padding.

**Cipher-block Chaining (CBC):**

For each block $P_i$ do:

- $C_0$ := initialization vector
- $C_i := E_k(P_i \oplus C_{i-1})$

To decrypt $C_i$ do:

- $C_0$ := initialization vector
- $P_i := D_k(C_i) \oplus C_{i-1}$

### 4.10 Padding Oracle Defenses

1. Don't use separate errors for MAC vs padding
2. Always check *integrity*
3. *Constant-time code*, all code paths must be equal
4. Limit branching
5. Don't use CBC mode

**Key size**: 128 bits (currently safe), 256 (quantum computers). MACs/PRFs need to be 2x cipher key size.

# 5 Key Exchange and Public Key Crypto

Alice and Bob can have a **public** conversation to derive a **secret** key.

**Diffie Hellman (DH) Protocol**: relies on *discrete log* problem (proven hard)

## 5.1 DH Protocol

Standard $g$ (generator), and $p$ (prime, or modulus).

1. Alice picks secret $a$ (sends $g^a \bmod p$)
2. Bob picks secret $b$ (sends $g^b \bmod p$)
3. $g^{ab} \bmod p = g^{ba} \bmod p$

**Man in the middle (MITM)**: Intercepts each and sends $g^m \bmod p$.
**Defense**: Use digital signatures and cross your fingers! (Rely on out-of-band communication, or physical contact).

## 5.2 Key Management

This is the hard part, goal: *forward secrecy.* Each key should only have **one** purpose (vulnerability increases the more you use it, the more places you store it, the longer you have it). Keep keys far from attacker, protect against compromise of old keys.

**Forward Secrecy:**

- Learning old key shouldn't help an adversary learn new key.
- Compromising an individual session ex post facto should not compromise future sessions
- Compromising a long-term key should not enable decryption of recorded ciphertexts

## 5.3 Public Key Cryptography

The solution to scaling issues (publishing or receiving data from/to lots of people with integrity and confidentiality).

(So far, our encryption key == decryption key). Now, keys are **different** and you *can't find one from the other.*

Get a key pair and split it:

- Public key: make this *public*
- Private key: keep this *private*

Bob generates a private key $B$ and public key $A$, sharing $A$ with Alice. She can encrypt messages to Bob with $A$, that he decrypts with $B$. He signs messages with a signature generated with $B$, and she verifies it with $A$.

**Most famous:** RSA, will stay secure about 5 years: gap between multiplication (expensive of increasing n) and factoring (getting efficient as n increases) is closing

## 5.4 RSA

How it works:

1. Pick 2 large random primes $p, q$
2. $N := pq$ (RSA does multiplication mod $N$)
3. Pick $e$ to be relatively prime to $(p-1)(q-1)$
4. Find $d$ so that $ed \bmod (p-1)(q-1) = 1$
5. Public key := $(e, N)$, Private key := $(d, N)$
6. $E_e(x) = x^e \bmod N$, $D_d(x) = x^d \bmod N$

**Why does RSA work?** For all $0 < x < N$ we can show that

$$E(D(x)) = D(E(x)) = x$$

**Drawbacks**: 1000x or more slower than AES, dominated by exponentiation ($\sim$cube of key size), message must be shorter than $N$.

## 5.5 RSA Security

Best way to compute $d$ from $e$ is factoring $N$ into $p$ and $q$.

RSA can be used for *either* confidentiality or authentication.

- Confidentiality: $E_e(m), D_d(m)$
- Integrity: $E_d(m), D_e(m)$ (digital signature)

## 5.6 In Practice

Use a crypto library!

**Encryption:**

1. $x :=$ random integer
2. Encrypt $x$ using RSA
3. $k :=$PRF$(x)$
4. Encrypt $m$ using $k$ and a symmetric cypher

**Signing:**

1. $v :=$ PRF $(m)$
2. Use RSA to sign a carefully padded version of $v$

### 5.7 Summary: A Secure Channel

1. Establish a shared secret $K$ using DH protocol
2. Make sure they're really talking to each other by exchanging and verifying RSA signatures on $k$
3. Use a PRG to split $k$ into 4 distinct keys (integrity and confidentiality in each direction)
4. Encrypt with symmetric cipher, then add MACs for integrity

### 5.8 Elliptic Curve Crypto

$y^2 = x^3 + ax + b$, horizontal symmetry (any non-vertical line will intersect in $\leq$ 3 places)

## 6 Web Architecture

Involves Internet, client/server, protocols.

### 6.1 Early Protocols

- telnet (later ssh)
- ftp (later scp, sftp)
- smtp (later pop2, pop3, pop3s, imap)
- nfs (later AFS, NFSv4)

**World Wide Web:** combines HTTP and HTML

### 6.2 Hypertext Transfer Protocol (HTTP)

Core Web request-response protocol, with 4 phases:

1. Open connection
2. Client request
3. Server response (stateless)
4. Close connection

Internet robustness principle: conservative in sending, liberal in accepting.

### 6.3 Cookies

While HTTP is a 'stateless' protocol, **cookies** were invented as a way to **store state** on client that could be used by the server (e.g., identity)

- Set-Cookie: <name>=<value>, sent by server in HTTP response (stored by browser)
- Cookie: <name>=<value>, sent by client in all subsequent HTTP requests

**Tracking cookies**: site 1 embeds a request to site 2 (e.g. img src), site 2 server responds with Set-Cookie along with the image; site 3 also embeds an image from site 2, the browser sends a 'GET' with the same cookie.

## 7 Web Attacks

Attacks and defenses regarding cookies, CSRF, SQL injections, XSS.

### 7.1 Cross Site Request Forgery (CSRF)

Forces an end user to execute unwanted actions on a web application in which they're currently authenticated.

**Defense:** Authenticate that each user action originates from our site (each action gets an associated token)

1. On a new action (page), verify token is present and correct
2. Attacker can't find token for another user, and thus can't make actions on the user's behalf

**Same-origin policy**: Two pages have the same origin if the protocol port and host are the same for both pages (cross-origin embedding is allowed but cross-origin scripting is not).

### 7.2 Code Injection

An attacker 'injects' code into a vulnerable computer program to change the course of execution.

- Confusing **data** and **code** (server expects data but instead gets and (unintentionally) executes code
- Common and dangerous class of vulnerabilities
  - Shell injection
  - SQL injection
  - Cross-site scripting (XSS)

### 7.3 Structured Query Language (SQL) Injections

SQL is a language to ask 'query' databases questions. An attacker's SQL injection statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker).

**Defense:** Make sure *data* is interpreted as *data*

- Sanitize: escape control/comment/quote characters
- Use prepared statements (declare what is data)

### 7.4 Cross Site Scripting (XSS)

An attacker uses a web application to inject malicious scripts into otherwise benign and trusted web sites.

**Defense:** Make sure *data* is treated as *data* and **not** executed as *code*

- Escape or reject special characters
- Use frameworks (declare what's user-controlled data and the framework can automatically escape it)

## 8 HTTPS

HTTPS, TLS and the CA Ecosystem.

Normal HTTP Transaction:

$$\text{HTTP} \leftrightarrow \text{TCP}$$

HTTPS Transaction:

$$\text{HTTP} \leftrightarrow \text{TLS} \leftrightarrow \text{TCP}$$

### 8.1 Transport Layer Security (TLS)

(Formerly SSL: Secure Socket Layer) Most important cryptographic protocol on the Internet. Almost all popular encrypted protocols (except SSH) use SSL/TLS for **transport encryption**.

When you need an encrypted socket for your application, use TLS.

Threat Model (Adversarial Network):

$$\text{Browser} \leftrightarrow \textbf{Malicious Network} \leftrightarrow \text{Web Server}$$

- Attacker controls infrastructure
  - Routers
  - DNS
  - Wireless access points

- Passive attacker: eavesdrops
- Active attacker: eavesdrops, injects, blocks, and modifies packets
- E.g. Internet cafés, hotels

## 8.2 Certificate Authority (CA)

Similar to a notary, it verifies identity.

Like 5.3, Bob.com (server) has a private key $B_{priv}$ and public key $B_{pub}$. Alice (browser) knows $C_{pub}$, while the CA keeps a secret $C_{priv}$. How does the browser obtain the server's public key?

1. Bob.com generates $B_{pub}$ and $B_{priv}$, sending $B_{pub}$ and proof of his identity to the CA.
2. CA checks identity proof
3. CA sends Bob.com a certificate, stating $B_{pub}$ and signed with $C_{priv}$
4. Bob.com keeps this certificate on file
5. When Alice visits Bob.com, the server sends the certificate to the browser
6. Alice can verify the signature on the certificate using $C_{pub}$

**The Ecosystem**:

- Each browser trusts a set of CAs
  - CAs can sign certificates for new CAs
  - CAs can sign certificates for *any* website
- If a *single* CA is compromised the *whole system* is compromised (complete trust of the Internet lies in the weakest CA)

## 8.3 The TLS Handshake

The 4-way handshake between the client and server for a **encrypted communication channel** (symmetric).

1. (*client* $\rightarrow$ *server*)
   - Client Hello: Here's what I support // nonce
2. (*client* $\leftarrow$ *server*)
   - Server Hello: Chosen Cipher
   - Server certificate: Here is my x.509 Certificate
   - Server: Here's $\big\{$your nonce // server DH parms $\big\}$ signed
3. (*client* $\rightarrow$ *server*)
   - Client Key Exchange: DH parms
   - Client: Change Cipher Spec
4. (*client* $\leftarrow$ *server*)
   - Server: Change Cipher Spec

### 8.4 Cipher Suites

ECDHE - RSA - AESGCM - SHA256

ECDHE: Ephemeral Key Exchange
RSA: Authentication
AESGCM: Data Transfer Cipher
SHA256: Message Digest

### 8.5 HTTPS User Interface

Help users authenticate sites.

- Lock icon (displayed when page was fetched with HTTPS)
  - HTTPS cert must be issues by a CA trusted by browser
  - HTTPS cert is valid (not expired/revoked)
  - CommonName in cert matches domain in URL
- Extended Validation (EV) certificates
  - Green bar in Firefox with org name (banks, e-commerce)
  - CA does extra verification of identity (expensive, more secure)
- Invalid certificate warnings: hard to override, but users do anyways

**HTTPS Beyond Crypto:**

1. Mixed Content: can't load *active* HTTP content on a HTTPS page (some passive is allowed)
2. Strict Transport Security (HSTS): HTTP header indicates always HTTPS to prevent downgrade attacks and protect future sessions (not first session)
3. Preload lists: lists of HTTPS only sites shipped with the browser

## 9 Attacks Against HTTPS

### 9.1 Attacking Browser UI

- **Picture-in-picture Attack:** Spoof the user interface (attacker page draws fake browser window with lock icon)
- **Semantic Attacks:** using international character sets or hiding domain in a long URL (e.g. micros0ft.com)
- **Invalid Certificates:** Expired certificates, CommonName doesn't match URL, unknown CA (self-signed)

### 9.2 Attacking Site Design

- **ssl_strip Attack:** (Browsing is HTTP, switches to HTTPS for checkout *or* connects via HTTP, switches to HTTPS for login) a transparent proxy strips out redirects, relays HTTP to HTTPS on server
- **Mixed Content Attack:** Page loads over HTTPS but contains HTTP content (e.g. JS, Flash) that an attacker can tamper with (browser warnings as a defense)

### 9.3 Attacking CA Ecosystem

The system is a distributed architecture so *nobody knows* the complete set of trusted intermediate CAs (there is a history of CAs being hacked, e.g. **DigiNotar**).

### 9.4 Attacking Implementations

1. OpenSSL Heartbleed (2014) trusted user provided length fields and echoed back memory contents following request data
2. Apple Goto Fail (2014) skipped CA checks due to a stray goto statement
3. Mozilla Bersek (2014) allowed spoofing of certificates
4. Null Prefix Attack (2009): browsers use c-strings (e.g. CA validates CommonNames containing the null character)

### 9.5 Attacking the TLS Protocol

- RC4, CBC, Compression-related, Export-related attacks
- DROWN Attack (2015) was a Bleichenbacher padding-oracle attack that exploited servers that supported obsolete SSL 2.0 to attack connections that use modern TLS.

## 10 Networking

The Internet is a 'network of networks', loosely hierarchical with **protocols** that control sending/receiving of data (e.g. TCP, IP, HTTP, FTP, PPP).

### 10.1 Organizational Layers

Like an airport, utilizes **layers of functionality**. Each layer implements a service via its own internal-layer actions (relies on services provided by layer below).

Layering is useful in complex systems:

- **Explicit structure** allows identification, relationship of complex system's pieces

- **Modularization** eases maintenance, updating of system (change of implementation of *one* layer service transparent/doesn't effect the rest of the system)

**The Layers:**

1. Application Layer (client app $\leftrightarrow$ server app): *the application packet*
2. Transport Layer (socket $\leftrightarrow$ socket): *TCP header and data*
3. Network Layer (host $\leftrightarrow$ host): *IP header and data*
4. Link Layer (device $\leftrightarrow$ device): *the frame header, data, and trailer*

## 10.2 Protocol Definition

Protocol *defines*:

1. Types of messages exchanged (e.g., request, response)
2. Message syntax (fields in messages and how they are delineated)
3. Message semantics (meaning of field information)
4. Rules for when/how processes send/respond to messages

Types of protocols:

- **Open**: defined in RFCs, allow for interoperability (e.g. HTTP, SMTP)
- **Proprietary**: e.g. Skype

## 10.3 Domain Name System (DNS)

A core Internet function, a distributed database implemented in hierarchy of many name servers to *map* between **IP addresses** and **hostnames**, and vice versa.

**Application-layer protocol**: hosts, name servers communicate to resolve names (address/name translation). **End-to-end principle**.

**Name Resolution:**

$$\text{Client} \rightarrow \text{ISP DNS Server} \rightarrow \text{Name servers}$$

Name Servers: *Where is www.example.com?*

1. **root**: try com nameserver
2. **com**: try example.com nameserver
3. **example.com**: 7.208.77.188.166

$$\text{Client} \leftarrow \text{ISP DNS Server} \leftarrow \text{Name servers}$$

## 10.4 Internet Transport Protocols

**TCP: Transmission Control Protocol**

- *Connection-oriented* setup required between client and server
- *Reliable transport* between sending and receiving
- *Flow control* so sender won't overwhelm receiver
- *Congestion control* to throttle sender when network overloaded

Doesn't provide: timing, minimum throughput guarantee, security.

**UDP: User Datagram Protocol**

- *Message-oriented*
- *Unreliable Data Transfer* between sending and receiving

Doesn't provide: connection setup, reliability, flow/congestion control, timing, throughput guarantee, security

## 10.5 Transmission Control Protocol (TCP)

Provides connection-oriented between app processes running on different hosts. Runs entirely in end systems:

- **send** side: breaks app data into segments, passes to network layer
- **recv** side: reassembles data into messages, passes to app layer

Connection Establishment: *TCP Handshake*

1. client sends SYN packet to listening server
2. server sends SYNACK packet to client
3. client receives SYNACK *indicating server is live*,
   sends ACK packet to server
4. server receives ACK packet, *indicating client is live*

## 10.6 Network Layer: IP

Network layer protocol (**IP**) is used in *every* host and router (the funnel of an hourglass) to transport data from sending to receiving host (routers along path examine header fields in datagrams to identify where to send them).

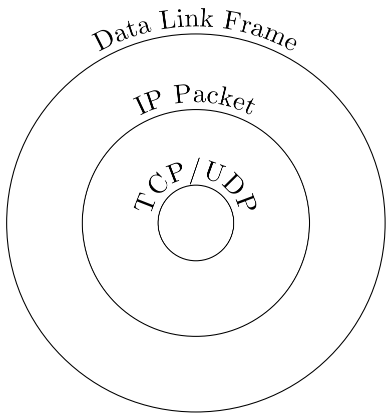
Core Functions:

1. **Forwarding**: move packets from router's input to appropriate router output
2. **Routing**: determine path taken by packets from source to dest.

Internet Protocol:

- **Connectionless**: each packet transported independently
- **Unreliable**: delivery on 'best effort', no acknowledgements (packets could be lost, reordered, corrupted, duplicated)

**IP Addresses**: used to route datagrams through the network. Divided into 2 parts: Network (used to route, like a street name) and Host (used to identify host, like a house number)



**Router**: IP network device with *multiple links*. When a packet arrives, the router forwards it along one of the links based on its **destination address** (in header), according to a **forwarding table** (directs next hop). Internet routers communicate to compute the tables using an **routing algorithm** (determines end-to-end path).

## 10.7 Autonomous Systems (ASes)

The Internet is a collection of autonomous systems (a set of routers and networks under the same administrative control).

- Intra-domain routing: under a single org
- Inter-domain routing: product of distributed computation
- **Border Gateway Protocol** (BGP): inter-domain routing protocol, allows each AS to advertise existence ("I'm here").

## 10.8 Link Layers

Transfers a datagram from one node to a *physically adjacent* node over a **link** (wired, wireless, LAN).

**Communication**: connectionless (no handshake), unreliable (packets dropped in network errors).

### 10.9 Ethernet

**Frame Structure**: sending adapter encapsulates IP datagram (or other network layer protocol packet) in Ethernet frame.

**Switch**: a common interconnection device (operates at link layer, with multiple ports connected to computers) that:

- learns the MAC address of connected computers
- forwards frames to the destination computer

**Address Resolution Protocol** (ARP): *Host A sending an IP datagram to B over a LAN, how to find B's MAC address?*

1. *A* **broadcasts** ARP query packet to all LAN nodes (Who has IP address X, tell *A*)
2. *B* receives ARP query, recognizes its IP, replies to *A* with its MAC.
3. *A* caches IP-to-MAC address pair in its ARP table until the info times out

## 11 Network Attacks

**Vulnerabilities**: unencrypted transmission, no source authentication, no integrity, no built-in bandwidth control.
**Attackers**: Eve the Eavesdropper (passive), Mallory the Man in the Middle (active).

### 11.1 Wired and Wireless

- Wireless Networks: anyone in range can listen (defense: WPA2)
- Wired Network
  - Hub: sends copies to anyone connected
  - Switch: uses MAC address to send to relevant ports

**Dynamic Host Configuration Protocol:**

1. Client connects, spams out DHCP discover request (to whole network)
2. A DHCP server replies with an offer (willing to assign it IP address X)
3. Client says yes
4. Server passes along IP and network configuration

How does a client know its gateway to the Internet? **DHCP**.
How does a switch know where to send traffic? **ARP**.

**ARP Spoofing:**
*ARP isn't authenticated* (shout loud enough and you get the traffic)
How could we authenticate ARP? *Signatures.* Defend against the spoofing? *tcpdump, Wireshark.*

## 11.2 Overview

**What can attackers do?**

1. **Capture**: record sensitive or session (cookies) data
2. **Modify**: infect executables, inject content (ads)

**Solution?**

- Secure protocols (TLS, SSH, etc)
- VPN
- Wireless: WPA2

## 11.3 Kaminsky Attack

IP has no source authentication (raw sockets). DNS happens over UDP (TCP is harder to attack because it has the handshake).

1. **Problem:** DNS has transaction IDS (which response went where), but it's *only* a 16-bit field (easy to guess, only $2^{16}$ possible IDs)
2. **Solution:** randomization of source port

*Reflection*: A UDP has a to, from, data, and protocols (NTP, DNS).

## 11.4 Border Gateway Protocol (BGP)

Unauthenticated ASes communication: how to get to all the IPs each one owns; turns into a spanning tree, → **efficient routes between IPs** (iterative broadcasting).

**Vulnerability:** Attempting *internal* routing (drop packets to 'block' sites), e.g. Youtube instance

- Nothing can stop this type of hijacking
- No centralized trust
- Solution, **DNSSEC**: *Signs DNS records.*

## 11.5 Network Exploits

1. *Virus*: exploit that attaches onto another file/program.
2. *Trojan*: exploit disguised as something of interest
3. *Worm*: self-propagating exploit
    - **Propagate** by *scanning for vulnerable hosts* or sending to everyone in email/chat contacts
    - They blackmail/ransom-ware, ex-filtrate data, corrupt files, or are bots, leading to **botnets** (permanently compromised systems with communications), that:

  - Spam
  - DDos attacks
  - Bitcoin mining

## 11.6 Summary: Attacking Each Layer

1. Layer 2 (Link): Ethernet
   - ARP spoofing, only works on local network
2. Layer 3 (Network): IP
   - IP spoofing, can/should be filtered at the edge
   - BGP route hijacking
   - DNS spoofing
3. Layer 4 (Transport):
   - TCP injection: guess sequence numbers
   - DoS: TCP RST
   - Can also be solved with edge filtering (everyone is lazy, selfish, cheap)
   - UDP amplification (DNS, NTP)
4. Layer 7 (Application)
   - DNS maps names to IPs
   - BGP maps routers to IPs

Don't trust layers beneath you!

# 12 Authentication

How can we authenticate a person?

1. Something you *know*, could forget
   (a) Passwords
2. Something you *have*, could lose
   (a) Two-factor
   (b) Tokens
   (c) Phone
3. Something you *are*, could no longer be
   (a) Biometrics

## 12.1 Passwords

They **fail**: forgotten, too easy, stolen, brute force attacks, unsafe recovery process; guess online (slow, detectable), guess offline (fast), phishing, authentication bypass vulnerabilities.

**Online Brute Force Attack:**

1. Attack: Submit guesses directly to website, try to log in
2. Defenses: lock after $x$ attempts, captcha checks, rate-limit, anomaly detection

**Offline Password Brute Force Attack:** password database is compromised

1. Plaintext passwords:
   (a) *pros*: fast/easy
   (b) *cons*: revealed when system compromised
2. Encrypted passwords:
   (a) *pros*: none
   (b) *cons*: revealed when system compromised
3. Hashed passwords:
   (a) *pros*: leak doesn't immediately reveal passwords
   (b) *cons*: identical passwords have same hash
4. Salted hashed passwords: randomly generate *salt*, store both salt and hash. To authenticate, lookup salt and compare hash.
   (a) *pros*: passwords not revealed in leak, same passwords don't have same hash
   (b) *cons*: complicated

**Attacking Hashes/Dictionary Attacks:** brute force through possible passwords, check for hash matches (sped up with rainbow tables).

## 12.2 Rainbow Tables

Reduction function (not collision resistant) that maps a hash value back into likely plaintexts. *Only* stores **starting and ending value** (continue reducing table).

## 12.3 Password Storage

**What Hash Function to Use?** Not SHA, not MD5, not anything we use for integrity (those are fast-focused). We want **slow** and **memory-hard** (as many resources as we can without overloading). e.g., PBKDF2, bcrypt, scrypt, argon2

**Storage**:

1. *Not* in plaintext
2. *Not* encrypted
3. Use salted and hashed passwords
4. Use bcrypt or scrypt

### 12.4 Two-Factor Authentication

Defense against leaked/compromised passwords, requiring user action on each login (prove identity)

- Something you *have*
- Distinct from password (not even $2^{nd}$ password)
- Not computable from password

**Approaches**:

1. Synchronized hashing based on a seed **K** (store seed on server and token)
2. SMS/Voicemail/Phone Call (type login code)
3. Signed login approvals (applciation layer approval over TLS, sign responses with client private key)
4. Universal $2^{nd}$ Factor, phish proof! (open standard by Google)

## 13 Network Probing

How global measurement can improve Internet security.

- **NMAP**: probes TCP ports for open servers (tries to open a TCP connection - *handshake*)
- **ZMAP**: an open-source tool that port scans the entire IPv4 address space from one machine quickly

How do we randomly scan addresses without excessive state? **Addressing Probes** (scan hosts according to random permutation, iterate over multiplicative group of integers modulo p)

How can we identify problems with crypto implementations at *Internet scale*?

1. Collect public data from network (public key, signatures...)
2. Mine data for vulnerabilities
3. Investigate causes in implementations

**Problems with RSA**

- Repeated Keys (shared $N$)
- Repeated RSA factors (moduli share a prime factor)

# 14 Control Hijacking

CPU, call-stack, buffer overflows.

## 14.1 The CPU

**Registers:**

1. General purpose registers (EAX, EBX, EDI, ESI...)
2. Special purpose registers
   (a) EIP: instruction pointer (next instruct)
   (b) ESP: stack pointer (top of call-stack)
   (c) EBP: frame/base pointer (reference to current stack frame)

**Assembly code:**

- Move: mov eax, 0x34
- Add: add eax, 10
- Jump: jmp 0x12345678 //jump and *don't return*
- Call: call 0x12345678 //jump then *return*
- push, pop

Stack frames: start at 0xffffffff, grows toward 0x0 (x86 specific)

## 14.2 Buffer Overflow

- User input buffer overflow
- Network input buffer overflow

## 14.3 Integer Overflow

Unsafe: strcopy( ), sprintf( ), gets( ).

*Instead*, use strncpy( ), snprintf( ), fgets( ).

## 14.4 Automated Testing

Finding vulnerabilities manually is hard.

1. Memory Analysis Tools: useful for finding leaks (virtual environment, dynamic run-time checks)
2. Static Analysis Tools: look for dangerous coding practices/patterns
3. Taint Analysis Tools: trace value usage
4. Fuzzers: brute force inputs to monitor behavior

*Format String Vulnerabilities*: attack on a lack of sanitation

*Heap-Spray/Fung Shui*: abuse memory allocation by injecting data

## 15 Side Channels

Any observable side effect of computation that an attacker could measure and possibly influence (e.g. timing, light, power)

1. Power analysis attacks
2. Cache timing attacks
3. SSH password timing attack

Side channels: modem light, RF side, Acoustic (sound of keys), data remanace (image degradation)

**Defenses**: Ciphers with bounded side channel leakage (fixed-time algorithms (no data-dependent delays, branches)), TEMPEST fonts.

## 16 Malware

Malicious software, a set of instructions that run on your computer and do something an attacker wants it to do.

- Steal private data
- Display ads, spam, extortion, commit online fraud (click fraud)
- Damage local machine, congest network
- Attack other systems (DoS, relays)
- Grant unauthorized access (back door)

**Cause**? client machines are *badly insecure*.

### 16.1 Trojan Horses

Software that *appears* to perform a desirable function (in order to trick the user into installing it), but is actually designed to perform **undisclosed malicious function**.

1. Spyware
2. Adware
3. Ransomware

### 16.2 Computer Viruses

**Self-replicating software** that infects other programs by modifying them to include a version of itself (usually requires user action).

**Polymorphic**: mutate to avoid detection, changing code while keeping the algorithm intact

Types:

- Overwriting: destroys original code
- Pre-pending: keeps original code, compressed
- Infection of libraries: virus becomes memory resident
- Macro: infects MS Office docs, installs in main doc template

**Metamorphic Code:** every time the virus propagates, generate semantically different version of it (using a code rewriter)

**Computer Worm:** Self-replicating software that infects other systems by automatically spreading over a network or other media (generally infects machines by altering running code). *No user intervention required.*

1. Can potentially spread quickly because they parallelize the process of replicating
2. Results in exponential growth of the infection, until vulnerable population is saturated

A **rootkit** is a malware component that uses stealth to maintain persistent and undetected presence on a machine.

## 16.3 Bots and Botnets

Most botnets are operated by pro criminals for financial gain.

- **Botnets** are wide-scale, centrally controlled malware.
- **Bots** infect many hosts (via any of the above methods).
- **Botmaster** controls bots remotely, via command and control infrastructure.
- **Botnet Command and Control:** Upon infection, new bot '*phones home*' to rendezvous with command-and-control. Messages can be...
    1. Activation reports (bot to botmaster)
    2. DDoS/spam/HTTP proxy instructions
    3. Delivery reports: sniffed passwords, email address harvests

## 16.4 Malware Defenses (Anti-virus)

Defense against Trojans, viruses, bots, slow worms.

- Signature-based detection
    - Analyze virus to find a string that can identify it (like a fingerprint)
    - Collection of signatures in a malware database is usually proprietary
    - Difficult against mutating viruses
- Heuristic-based detection

  - Analyze program behavior to identify unusual pattern
  - e.g. network access, file open or delete, modify boot sector

- Tripwire
  - Maintain database of cryptographic hashes for operating system files and popular applications
  - Compute hash of each file on the disk and look up in database
  - Need to protect the integrity of the database (e.g. boot from clean external disk)

Defense against *fast spreading* worms (too quick to propagate a signature)? **Detect in the network instead.**

## 17 E-Voting

What could go wrong with elections in the context of cybersecurity? Systems need to enforce security requirements:

1. **Integrity**: election outcome matches voter intent
2. **Ballot Secrecy**: nobody can figure out how you voted, even if you try to prove it to them
3. **Voter Authentication**: only authorized voters can cast votes, and each voter can only vote $x$ times
4. **Enfranchisement**: all authorized voters have the opportunity to vote
5. **Availability**: the system accepts all votes on schedule, produces results in timely manner

**Post-election audits:** Pick precincts *randomly* for paper recounts. If the electronic tallies disagree, **recount everywhere**.

### 17.1 Internet Voting

**Server-side** Threats:

1. Denial of service
2. Remote intrusion
3. Insider attacks
4. State-sponsored attacks

**Client-side** Threats:

1. Credential theft
2. Imposter sites
3. Malware

**End-to-End (E2E) Voter-Verifiability**: my vote is casted as I intended, and counted along with all other votes cast, and I can't demonstrate how I voted.

# 18 Defending Weak Applications

Access control, isolation, sandboxing, virtual machines.

## 18.1 Access Control

A list of rights (read, write, execute) attached to an object (special permissions for user, group, everyone).

1. Services (daemons: web, mail, SSH, etc.) need root access to acquire resources.
2. Dropping root: running root is scary, principle of least privilege
   - (a) Start service
   - (b) Acquire resources
   - (c) Change effective user to deprivileged user

## 18.2 Isolation

Classic example: **chroot** (change root). chroot(2) changes apparent root directory.

## 18.3 Containers

Operating-system level **virtualization**: multiple isolated user-space instances. Why? For *limits* on disk space/RAM, root privilege isolation.

Sometimes the word doesn't fit into isolation models (users make life complicated). Solution? **Sandbox**.

## 18.4 Sandboxing

A sandbox defines a set of allowable actions, intercepts privileged actions, e.g. iOS Sandbox (hardened model where each app runs in its own container)

**Caveats**:

1. Need to know resources ahead of time
2. Restrictions
3. Doesn't make it easy to separate privilege
4. Less dynamic model

## 18.5 Virtual Machines

1. Host OS: what's running on hardware
2. Guest OS: virtual OS, running on virtual hardware, has restrictions
   - predefined disk, RAM, CPU cores

    - predefined access to, e.g. USB and network

*Virtual Machines vs Containers*:

- Virtual machine can have safe access to root
- Containers are more flexible, lightweight because of shared resources

## 19 Digital Forensics

1. Image forensics: don't believe everything you see
2. Code stylometry: stylistic fingerprints in source code
3. Data sanitation
4. Data recovery
5. Steganography: covert writing (hide content)
    (a) Unlike watermarks, which must be robust and detectable
    (b) e.g. LSB encoding (hide message in least significant bits of digital representation)
    (c) e.g. JPEG embedding (embed hidden message in LSB of JPEG coefficients)
    (d) Countermeasures: **outguess** tries to defeat statistical detector

## 20 Usable Security

Users, user testing, and web security. *Usability* is invisible when you don't need it, helpful when you do.

- **User interface (UI):** the visual/physical/aural elements that the user interacts with
- **User experience (UX):** the overall experience

### 20.1 Summary: Usability

1. Users don't have to think about security by default
2. We have to show **value** and build understanding of importance
3. Security will never be something users **like**
4. To make security easy, it has to be **simple**
5. Make decisions for user when possible

# 21 IoT Security

Risks exist in many new *emerging application domains.*

- Wearable devices: usable security (sensors are a new attack surface)
- Smart hospitals: access control
- Smart cities: privacy
- Autonomous driving: safety

# 22 Privacy, Anonymity, Anticensorship

## 22.1 Privacy

Control over your own information. Beware of:

1. Surveillance
2. 'Direct' sharing
3. Third-party tracking
4. Third-party cookies

**Defenses**: k-anonymity, differential privacy (algorithm for answering queries), encrypted mail

## 22.2 Anonymity

Concealing your identity.

- Proxies: intermediary that relays our traffic (not robust)
- Tor: works at transport layer, lets you make TCP connections without revealing IP address (much better!)
  - Network made of volunteer-run nodes (onion routers) globally
  - Risk: message bounces around a lot

## 22.3 Internet Censorship

1. Government censors (block 'offensive sites')
2. Observed techniques (IP blocking, DNS blackholes, forged RST packets)
3. Popular countermeasures (proxy)