# Encryption

Slides by Andrew DeOrio

# Agenda

- Overview
- Symmetric encryption
- Asymmetric encryption
  - Public key infrastructure
- Cryptographic hash functions
- Summary

- Today: general network security
- Next time: web security

# Network security

- Two parties communicate over a network
- Assume powerful adversary
  - Can read (eavesdrop on) all data transmitted
  - Can modify or delete any data
  - Can inject new data

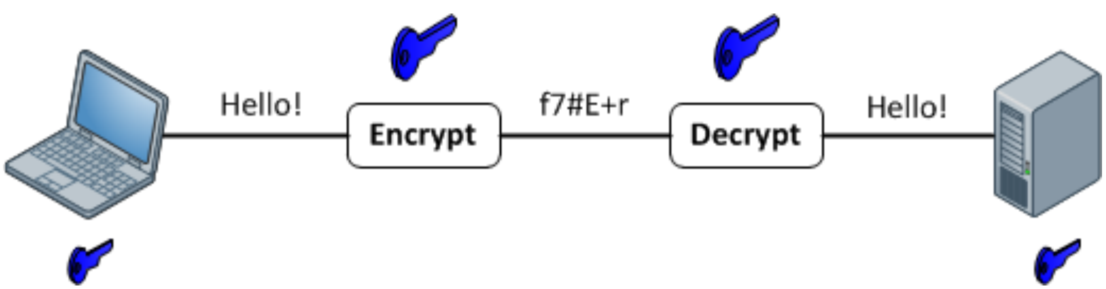
- Communication: What properties would you like?

# Desirable properties

- Confidentiality
  - Adversary should not understand message
- Sender authenticity
  - Message is really from the purported sender
- Message integrity
  - Message not modified between send and receive
- Freshness
  - Message was sent “recently”
- Anonymity
  - Attacker should not know that we are communicating

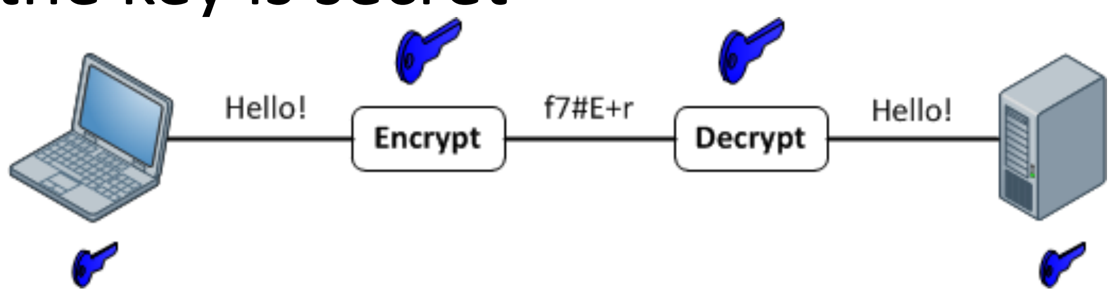
# Encryption as a function

- Plaintext message string
- Encryption key $K_{enc}$
- Decryption key $K_{dec}$
- ciphertext = encrypt (plaintext, $K_{enc}$)
- plaintext = decrypt (ciphertext, $K_{dec}$)

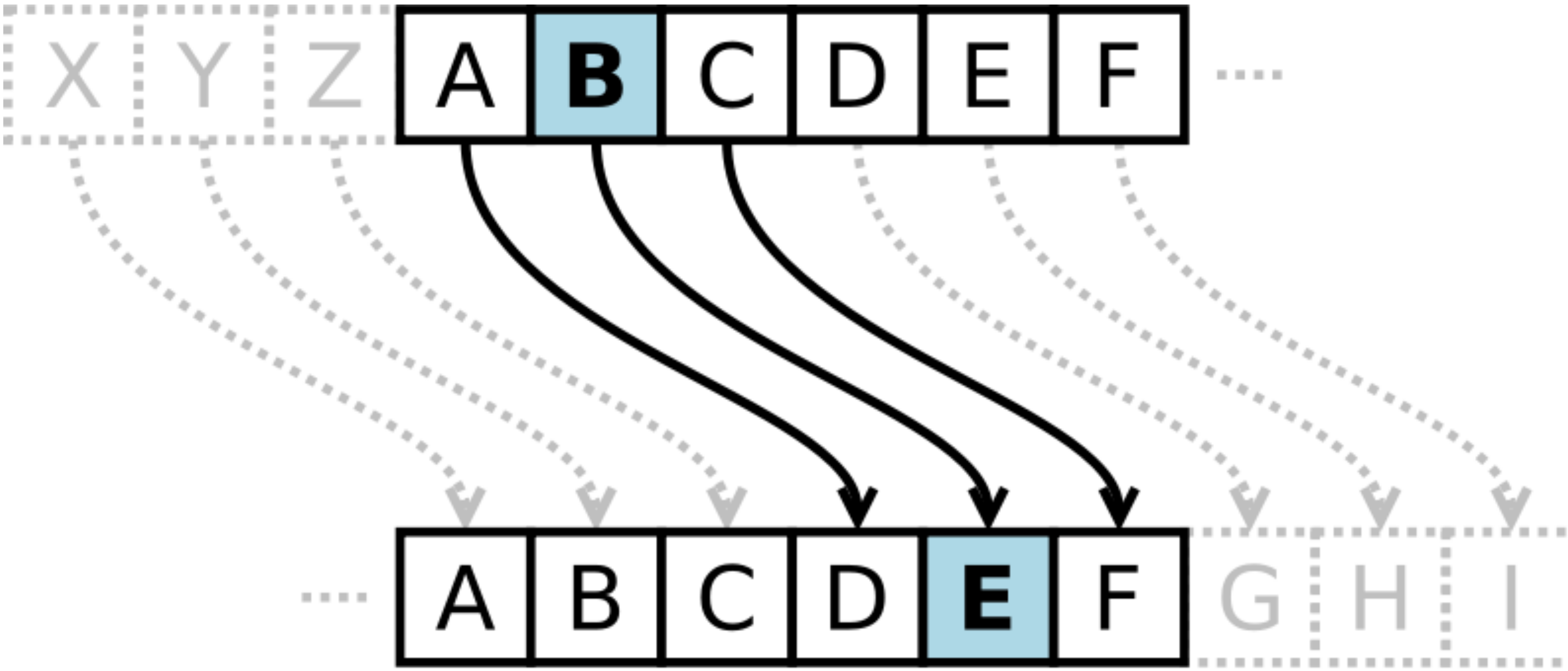


# Encryption in words

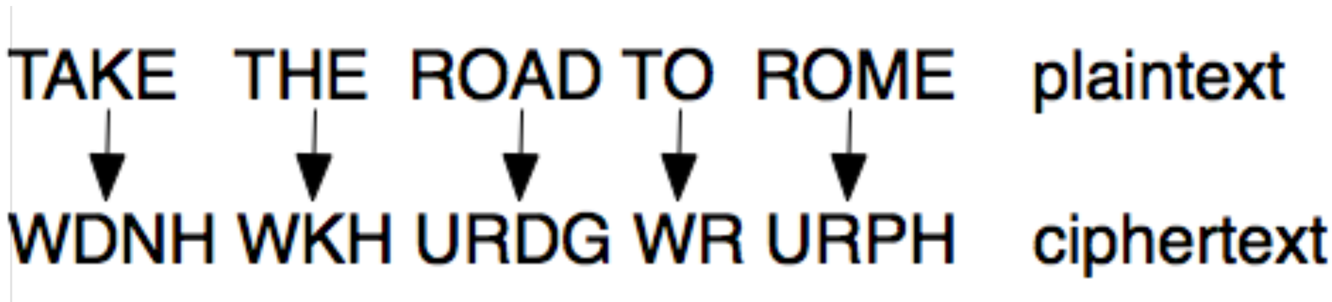
- Encryption applies a reversible function to some piece of data, yielding something unreadable
- Decryption recovers the original data from the unreadable encryption-output
- The encryption/decryption algorithm assumed known; the key is secret



# A brief history



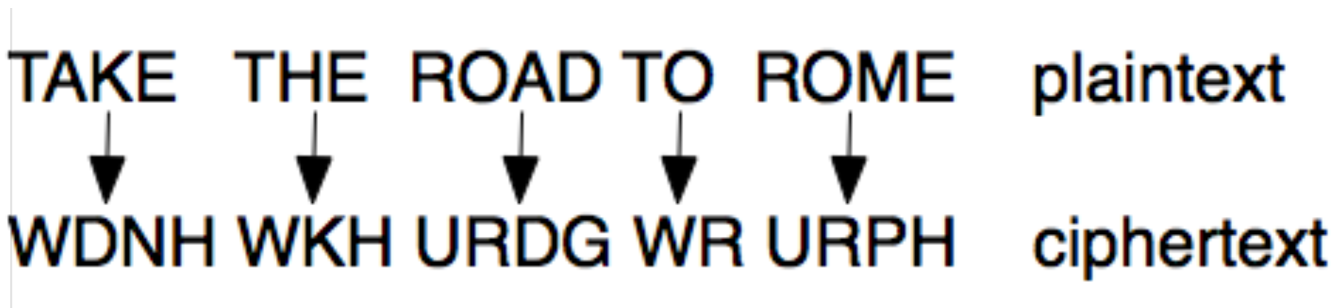
# A brief history




- How secure is this?

- If you found the ciphertext (inscribed on a piece of papyrus or something), how would you break it?

# Substitution ciphers



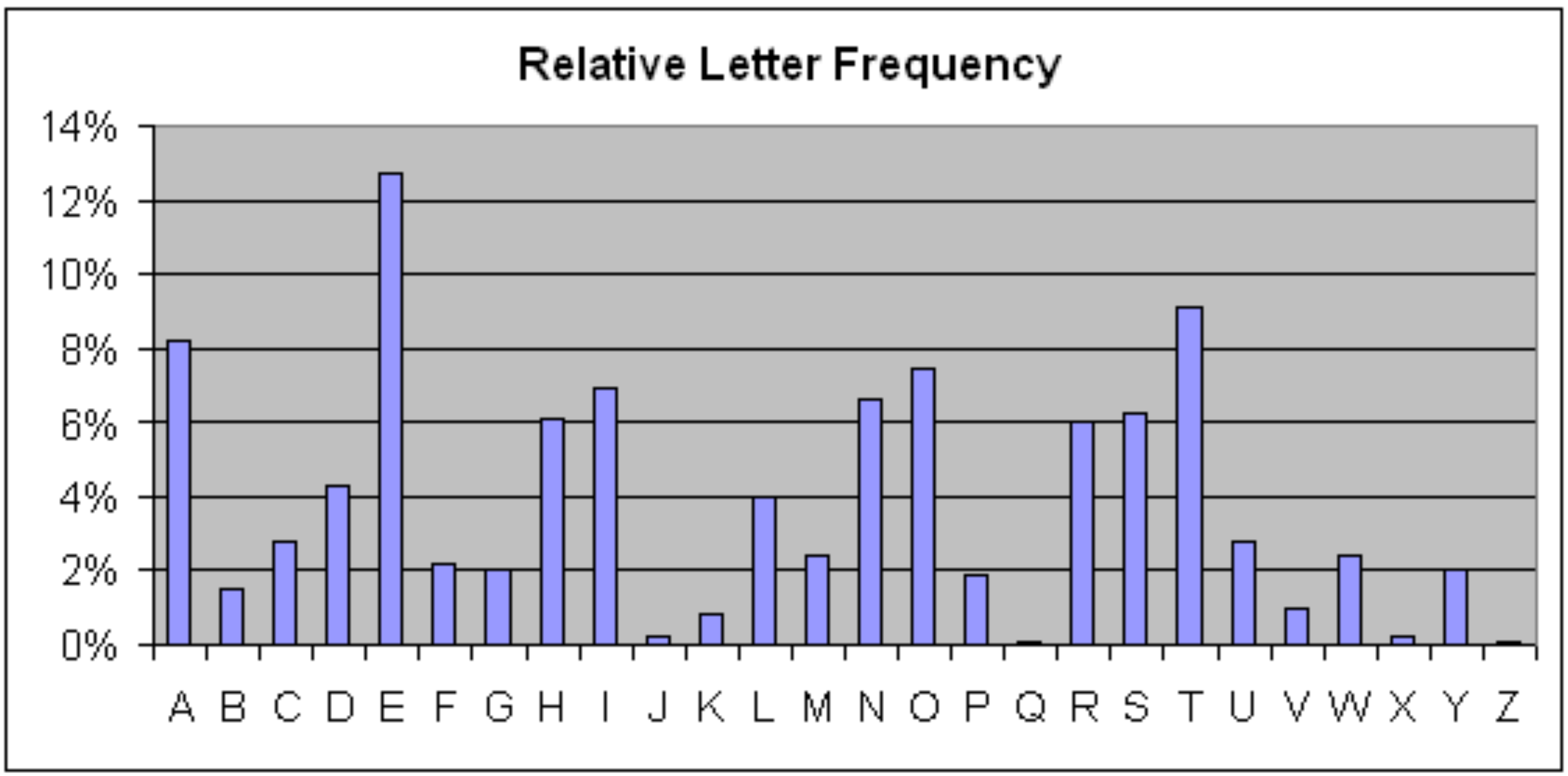

- No need to shift 3 chars
  - You could do 2!  Or even 4!
- You also don't have to shift the alphabet at all.  Just arbitrary 1:1 mapping of alphabet chars, using a substitution table

# Frequency analysis

- Substitution ciphers are vulnerable to frequency analysis
  - Letter
  - Word
  - Common phrases
- Frequency analysis discovered in $9^{th}$ century

# Frequency analysis

- Frequency analysis: count the frequency of each letter in the cipher text
- Compare against frequency of letters in English



# Polygram cipher

- Translate n-grams, not chars

| plaintext | ciphertext |
|---|---|
| AAA | QWE |
| AAB | RTY |
| AAC | ASD |

- How big is the substitution table?

# Polygram cipher

- Translate n-grams, not chars

| plaintext | ciphertext |
|---|---|
| AAA | QWE |
| AAB | RTY |
| AAC | ASD |

- How big is the substitution table?
  - $A^n$ entries, where A is size of alphabet
  - A=26,n=3; 17576 entries
  - A=100,n=6; 1T entries
- Still vulnerable, but requires more text

# Substitution rules

- Don’t store table explicitly; derive table rows using substitution rule
  - E.g., s XOR k, where k is key
  - Remember: security level depends on size of key
  - Key of len b => $2^b$ possible keys

# Substitution rules

- XOR “flips a bit” for input bits that correspond to key's `1`
  - Correspond to a 0?  No change

```
0000000001010101    plaintext
1011010010011100    key
1011010011001001    XOR
```

- Encrypted string should ideally show no pattern for frequency analysis attack
- Use key long enough to make ciphertext appear random

# Agenda

- Overview
- **Symmetric encryption**
- Asymmetric encryption
  - Public key infrastructure
- Cryptographic hash functions
- Summary

# Symmetric encryption

- The key in traditional crypto is used to encode the substitution rule
  - Needed to encrypt and decrypt
  - AES uses this technique
- Both sides need the same key
  - One side uses the key to encrypt
  - Other side uses the key to decrypt
  - Important that key is kept secret by each side
- This is called *symmetric encryption*

# AES (Advanced Encryption Standard)

- Originally known as Rijndael, after its authors Vincent Rijmen and Joan Daemen
- Designed and standardized by NIST competition, long public comment/discussion period. Winner among 15 finalists.
- Widely believed to be secure, but we don’t know how to prove its security.
- 128-bit block size
- Variable key size (128 or 256 bits)

# AES construction

- "Round-based" with ten rounds
- Split key into ten subkeys
- Perform ten rounds of substitution/permutation operations, each time with a different subkey

# Foot-Shooting Prevention Agreement

I, ____________ , promise that once
Your Name

I see how simple AES really is, I will <u>not</u> implement it in production code even though it would be really fun.

This agreement shall be in effect until the undersigned creates a meaningful interpretive dance that compares and contrasts cache-based, timing, and other side channel attacks and their countermeasures.

X ______________________ ______
Signature Date



http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html

# AES round

- Input: 128 bit plaintext, 128-bit subkey
- Output: 128 bit ciphertext
- Picture as operations on a 4x4 grid of 8-bit values

1. Non-linear substitution
   - Run each byte thru a substitution function
2. Shift rows
   - Circular-shift each row, *i*th row shifted by *i*
3. Linear-mix columns
   - Matrix operations, invertible
4. Key-addition
   - XOR each byte with byte of round subkey

| | | | |
|---|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ |
| $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ |

# AES

- 10 rounds
- Reversible
- Small changes to the input result in big changes to the output

# DES

- DES  = Data Encryption Standard
- Earlier symmetric encryption algorithm
- Early 70's to late 90's
- Now insecure
- Modified version Triple DES (3DES)
- Superseded by AES

- Interesting podcast episode about DES
  - Darknet Diaries Episode 12: Crypto Wars
  - https://darknetdiaries.com/episode/12/

# AES vs. DES

- AES: designed to run fast in software (8-bit embedded through 64-bit)
- DES: specifically designed to run slow in software
  - There's a 64-bit reordering (swap low/high bits)
  - Cryptographically meaningless, but slows down any software implementation

# AES vs. DES

- AES: Designed by two Belgian cryptographers, open NIST competition, no secrets in its design (late 1990’s)
- DES: Designed by IBM (with “help” from NSA), meant for commercial uses (1970’s)
  - NSA genuinely helped (made DES resistant to differential cryptanalysis)
  - Academics were worried about hidden weaknesses in DES (mysterious S-Boxes were mysterious)
  - When differential cryptanalysis was discovered by Biham and Shamir, DES was already resistant to it!

# Symmetric encryption summary

- $K_{enc} = K_{dec}$
- Tends to be fast to compute
- Symmetric encryption provides *confidentiality*
  - Adversary should not understand message
- Symmetric encryption can provide *message integrity**
  - Nobody modified the message in between send and receive
  - *If the message is "recognizable" it's probably OK.  To be really sure, you have to send a MAC (message authentication code) along with the encrypted plaintext.

# Agenda

- Overview
- Symmetric encryption
- **Asymmetric encryption**
  - Public key infrastructure
- Cryptographic hash functions
- Summary

# Symmetric and asymmetric

- Symmetric encryption: both sides have the same key
- Key distribution is the weak link
    - Hard to revoke
    - Disastrous if “codebook” is compromised
    - Hard to distribute (requires initial out-of-band secure exchange)
    - Impossible for the Web
- All of this changed in the 1970s

# Asymmetric encryption

- Asymmetric encryption uses a pair of keys
- AKA public key cryptography
- Each party has:
  - A **public** key, which is published freely
  - A **private** key, which is shared with no one
- A message encrypted with one can be decrypted with the other
- You can't derive one from the other
- Original idea due to Diffie, Hellman, but RSA (Rivest, Shamir, Adelman) popular

# Confidentiality

- Asymmetric encryption provides *confidentiality*
  - Adversary should not understand message
- Encrypt with public key, decrypt with private key

- Anyone can encrypt
  - ciphertext = encrypt (plaintext, $K_{alice\_public}$)
- Only Alice can decrypt
  - plaintext = decrypt (ciphertext, $K_{alice\_private}$)

- Sometimes called *sealing* a message

# Sender authenticity

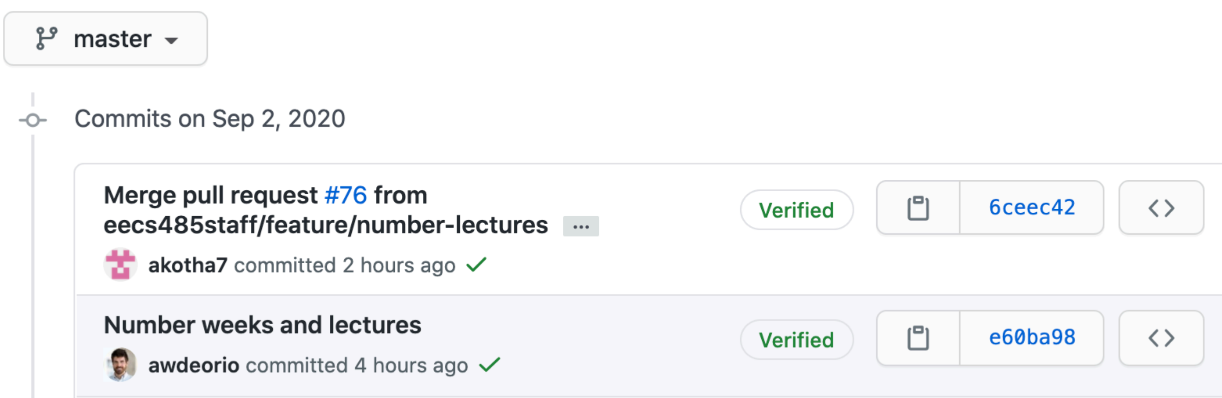
- Asymmetric encryption provides *sender authenticity*
  - Message is really from the purported sender
- Encrypt with private key, decrypt with public key

- Only Alice can encrypt
  - ciphertext = encrypt (plaintext, $K_{alice\_private}$)
- Anyone can decrypt
  - plaintext = decrypt (ciphertext, $K_{alice\_public}$)

- Sometimes called *signing* a message

# Asymmetric encryption examples

- Encrypted email
  - Only the recipient can decrypt
- SSH keys
  - GitHub knows it's you when you git push
- Verified commits on GitHub
  - Only the developer team modified this code



# Combining properties

- We can combine the two previous examples to gain all three desirable properties

- Confidentiality
  - Adversary should not understand message
- Sender authenticity
  - Message is really from the purported sender
- Message integrity
  - Message not modified between send and receive

*If the message is "recognizable" it's probably OK. To be really sure, you have to send a MAC (message authentication code).

# Combining properties

## Alice sends a message to Bob

1. ciphertext = encrypt(plaintext, K_bob_public)
   - Seal
2. signed_ciphertext = encrypt(ciphertext, K_alice_private)
   - Sign

## Bob receives message from Alice

1. ciphertext = decrypt(signed_ciphertext, K_alice_public)
   - Verify signed ciphertext, anyone can do this
2. plaintext = decrypt(ciphertext, K_bob_private)
   - Decrypt ciphertext, only Bob can do this

# How does it work?

- Public key cryptography relies on trapdoor functions
  - A function that is easy to compute, but hard to invert without special information
  - “Easy” and “hard” meant computationally
- Poor trapdoor functions
  - Add 2
  - Multiply by 3
- Difficult to find good trapdoor functions in practice
- Most popular one is related to prime factorization
  - Others possible

# Trapdoor functions

- n = p*q, where p and q are primes
  - Given p and q, easy to compute n
  - Given n, very hard to find p and q

- Public, private keys require original primes to compute
  - Only product of primes is ever exposed
  - Computationally extremely challenging to recover original primes

# From asymmetric to symmetric

- Asymmetric encryption is slow
- Symmetric is fast
- Use asymmetric to communicate key for symmetric
- Then, continue with symmetric encryption

# Agenda

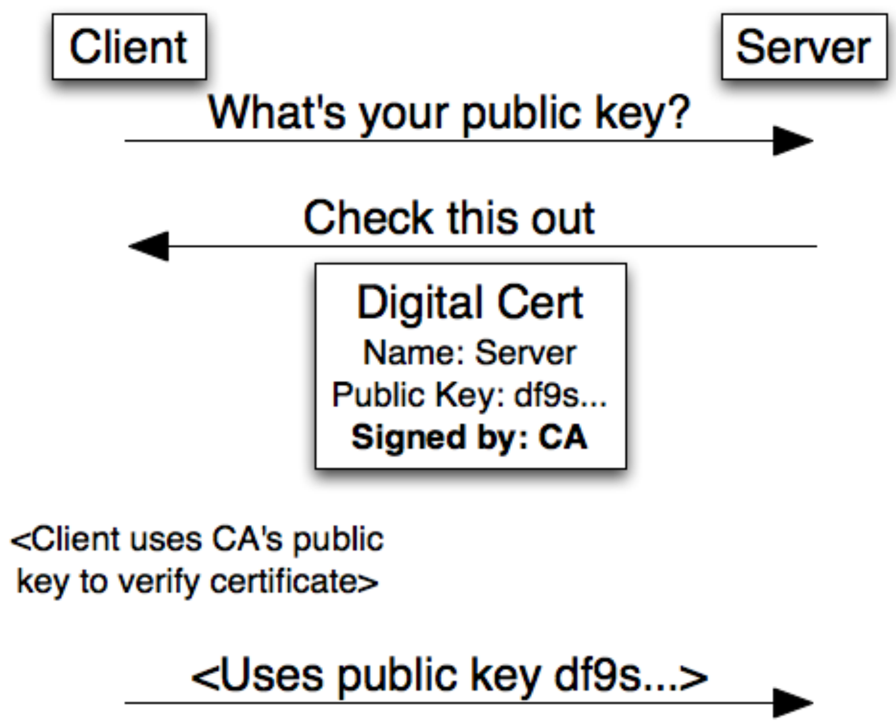
- Overview
- Symmetric encryption
- Asymmetric encryption
  - **Public key infrastructure**
- Cryptographic hash functions
- Summary

# Public key infrastructure

- How do you get a public key?
  - Read it out of the phone book, off a billboard, off a business card, from an email signature line
  - But there are lots of possible public keys
- What if the public key is faked?
  - Attacker Mallory distributes a fake public key for Bob
  - Alice sends message to Bob, encrypted with fake key
  - Mallory uses own private key to decrypt
- The Public Key Infrastructure (PKI) distributes public keys safely, via certificates
- A *certificate* is signed item that contains org's public key

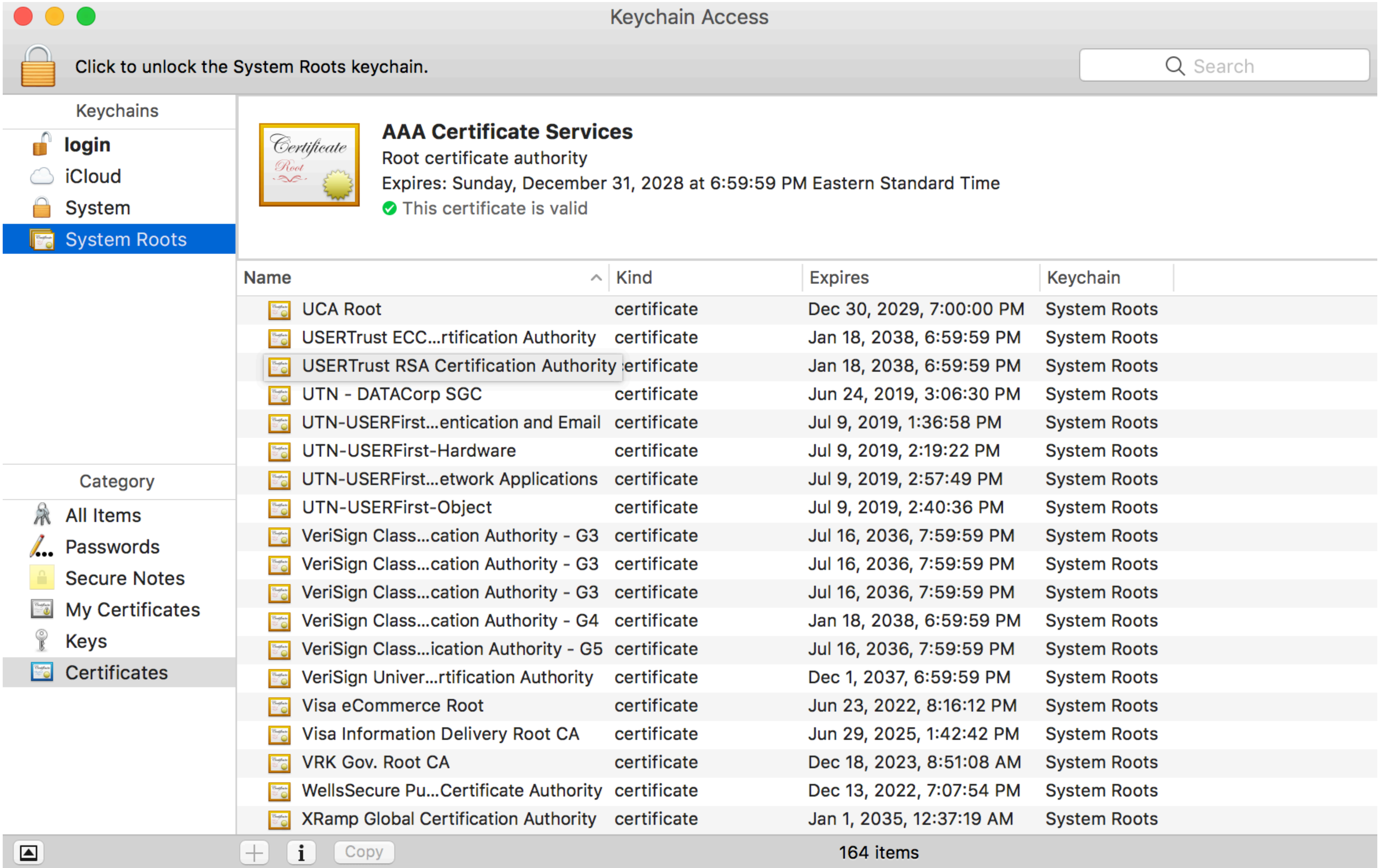
# Public key infrastructure

- What if the server is not authentic?
- How can we verify the certificate?
- Where is the weakest link?



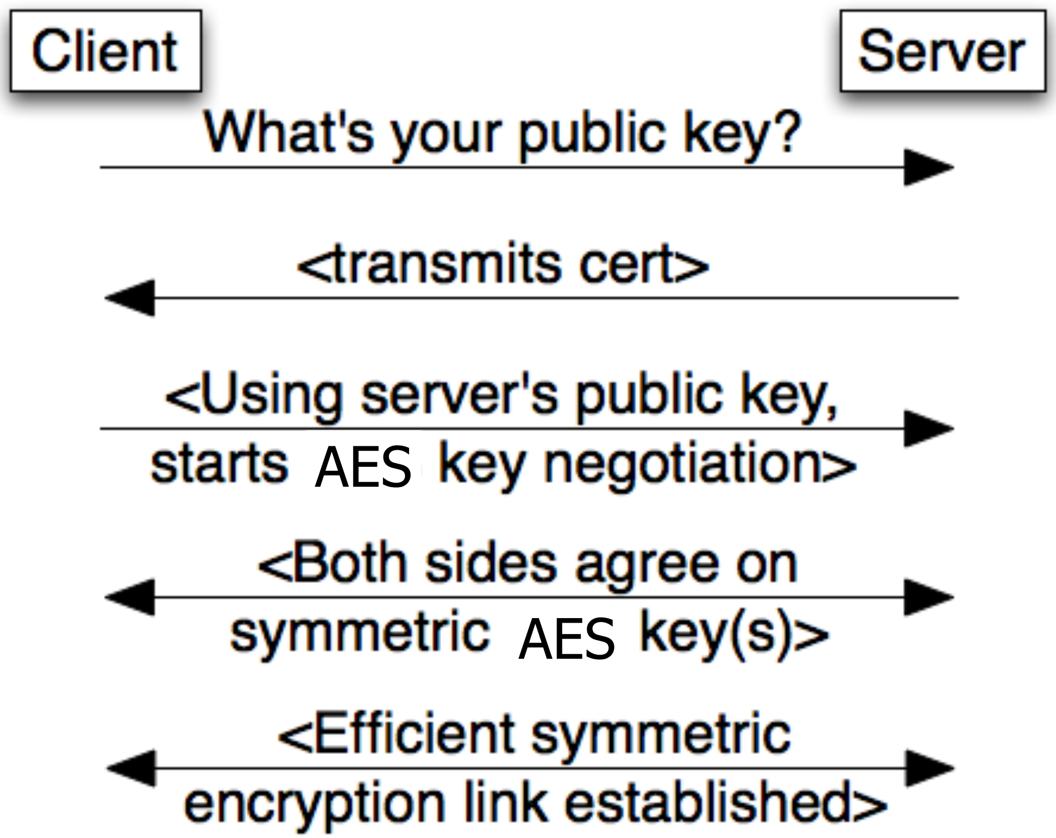
# Certificate authorities

- Verify identities and public keys
- Public keys for big Certificate Authorities (Verisign, Thawte, lots of others) are built into browsers
- There can be a chain of certificate signing
- You can start signing certs today! But you probably won't be built into Chrome
- Different cert "strengths" depending on level of identity verification


Keychain Access
Click to unlock the System Roots keychain.
Search
Keychains
login
iCloud
System
System Roots
AAA Certificate Services
Root certificate authority
Expires: Sunday, December 31, 2028 at 6:59:59 PM Eastern Standard Time
This certificate is valid
Name
Kind
Expires
Keychain
UCA Root certificate Dec 30, 2029, 7:00:00 PM System Roots
USERTrust ECC...rtification Authority certificate Jan 18, 2038, 6:59:59 PM System Roots
USERTrust RSA Certification Authority certificate Jan 18, 2038, 6:59:59 PM System Roots
UTN - DATACorp SGC certificate Jun 24, 2019, 3:06:30 PM System Roots
UTN-USERFirst...entication and Email certificate Jul 9, 2019, 1:36:58 PM System Roots
UTN-USERFirst-Hardware certificate Jul 9, 2019, 2:19:22 PM System Roots
UTN-USERFirst...etwork Applications certificate Jul 9, 2019, 2:57:49 PM System Roots
UTN-USERFirst-Object certificate Jul 9, 2019, 2:40:36 PM System Roots
VeriSign Class...cation Authority - G3 certificate Jul 16, 2036, 7:59:59 PM System Roots
VeriSign Class...cation Authority - G3 certificate Jul 16, 2036, 7:59:59 PM System Roots
VeriSign Class...cation Authority - G3 certificate Jul 16, 2036, 7:59:59 PM System Roots
VeriSign Class...cation Authority - G4 certificate Jan 18, 2038, 6:59:59 PM System Roots
VeriSign Class...ication Authority - G5 certificate Jul 16, 2036, 7:59:59 PM System Roots
VeriSign Univer...rtification Authority certificate Dec 1, 2037, 6:59:59 PM System Roots
Visa eCommerce Root certificate Jun 23, 2022, 8:16:12 PM System Roots
Visa Information Delivery Root CA certificate Jun 29, 2025, 1:42:42 PM System Roots
VRK Gov. Root CA certificate Dec 18, 2023, 8:51:08 AM System Roots
WellsSecure Pu...Certificate Authority certificate Dec 13, 2022, 7:07:54 PM System Roots
XRamp Global Certification Authority certificate Jan 1, 2035, 12:37:19 AM System Roots
Category
All Items
Passwords
Secure Notes
My Certificates
Keys
Certificates
Copy
164 items

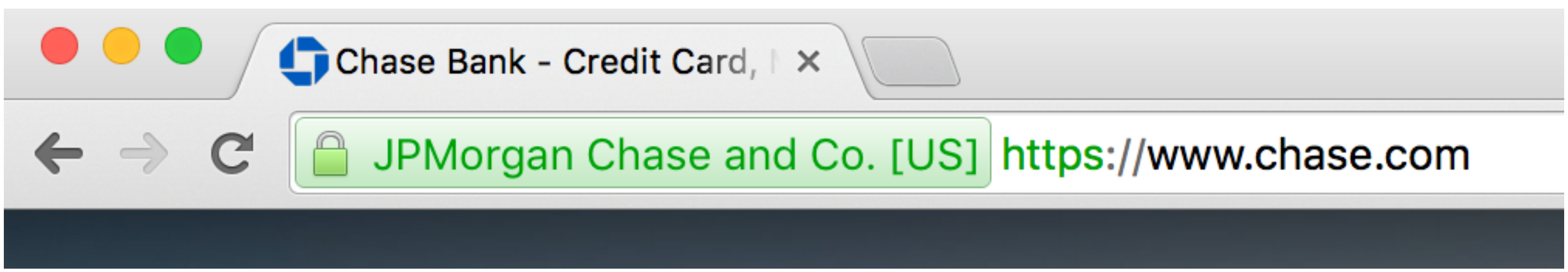
# Client server interaction

- Little public-key-encrypted data
- Browser verifies validity of certificate



# TLS/SSL

- Transport Layer Security / Secure Sockets Layer
- Commonly, https://
- Encryption of all content that goes into TCP payload



# TLS/SSL

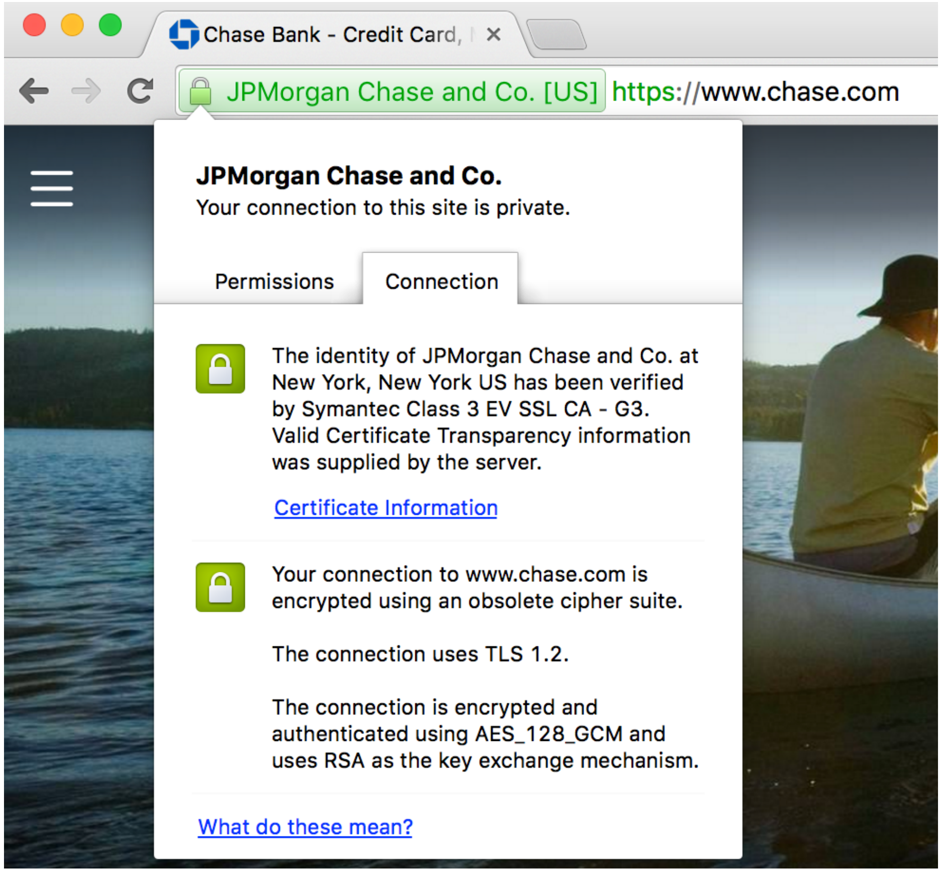
- SSL usually implemented by the server
- Two common production web servers are Nginx and Apache
- Server decrypts HTTPS traffic, then proxies dynamic page requests to backend
  - Server could be Nginx or Apache, etc.
  - Backend is `gunicorn` in EECS 485 AWS deployment

# HTTPS example

- 1. Hello
- Client sends hello message to server
  - Includes supported cipher algorithms and SSL version
- Server sends hello message to client
  - Includes selected cipher algorithm and SSL version

# HTTPS example

- 2. Certificate exchange
- Server proves its identity to the client
- Server sends SSL certificate and public key
- Clients checks certificate against stored CAs



# HTTPS example

- 3. Key exchange
- Client generates random key to be used for later symmetric encryption
- Client encrypts this key using the server's public key
  - Remember, only the server will be able to decrypt this message using the server's private key

- Then, traffic is encrypted with symmetric encryption using the agreed-upon key

# HTTPS example

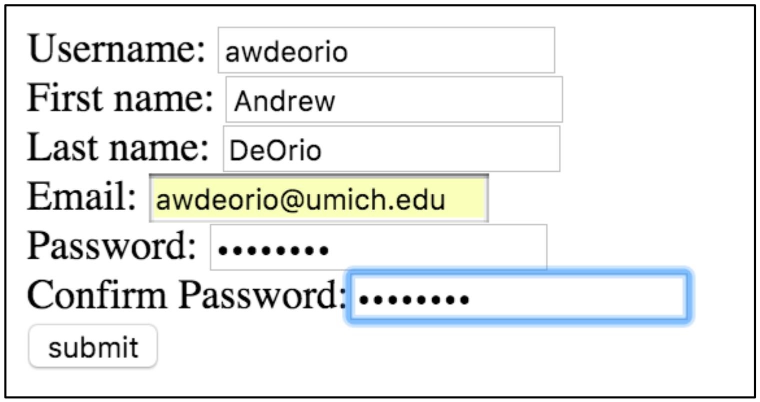
```
$ openssl s_client -connect www.google.com:443
 ---
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google
Inc/CN=www.google.com
   i:/C=US/O=Google Inc/CN=Google Internet Authority G2
 1 s:/C=US/O=Google Inc/CN=Google Internet Authority G2
   i:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
 2 s:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
   i:/C=US/O=Equifax/OU=Equifax Secure Certificate Authority
---
Server certificate
-----BEGIN CERTIFICATE----
...
```

# Agenda

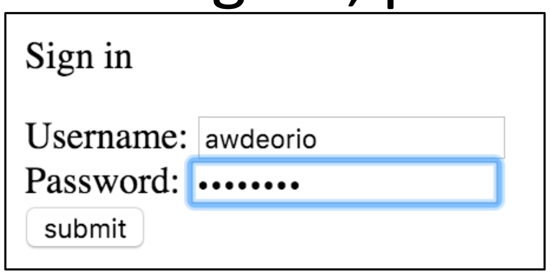
- Overview
- Symmetric encryption
- Asymmetric encryption
  - Public key infrastructure
- **Cryptographic hash functions**
- Summary

# Hashing passwords

- Bad idea: server stores password in database




- User logs in, password plain text compared to db




- What if someone gets a copy of the db?

# Story time

- Rock You was a popular website in the late 2000’s
- Stored their passwords in plain text
- Got hacked
- You can download all the passwords https://www.kaggle.com/wjburns/common-password-list-rockyoutxt
- Find out if your info has been seen in a hack https://haveibeenpwned.com/

# Hashing passwords

- Better idea: server hashes password using a one-way hash function
- If someone gets the database, they don't get the passwords

# Hashing passwords

- Example: MD5
  - Insecure!  Compromise in ~seconds to ~hours
  - Collision attack: find two inputs that produce the same hash

- Example: 512 bit SHA-2
  - First published in 2001 by US National Institute of Standards and Technology (NIST)
  - Resistant to collision attacks

# Example

- Using SHA-512 to hash a password

```
import hashlib
m = hashlib.sha512('bob1pass')
password_hash = m.hexdigest()
print(password_hash)
```

```
af1bd47889bff89ccc889bc2aa61437c2ac90ee411618645bd
4adbca1e02f8a277729093ea8ac094d3265352b75b12af1b4a
50edd8fc5783cc0fac0411cde8c2
```

# Cracking passwords

- Brute force attack: try every possible password, hash it, see if it matches db entry
- Dictionary attack: try all the words in the dictionary
  - Actually, many dictionaries
- Example: John the Ripper (`john`) is an open source program for password cracking

# Rainbow tables

- Rainbow tables speed up brute force attacks with pre-computed tables
  - Example: download MD5 rainbow tables
  - Example: generate your own with RainbowCrack
- Compute (or download) the table once, use it many times on the same database of passwords
- Recover all the passwords

# Protecting against cracking

- Rainbow tables assume that all the passwords were "hashed the same way"
- Alter the way each password is hashed using a *salt*
- *Salt* is a random number appended to the password plain text
- Each password is encrypted with a different salt
- Store the salt with the password
- Now you would need a different rainbow table for every password!

# Example: hashing with a salt

- Using SHA-512 to hash a password with a salt

```
import hashlib
import uuid

algorithm = 'sha512'
password = 'bob1pass'
salt = uuid.uuid4().hex
m = hashlib.new(algorithm)
m.update((salt + password).encode('utf-8'))
password_hash = m.hexdigest()
print(algorithm, salt, password_hash)
```

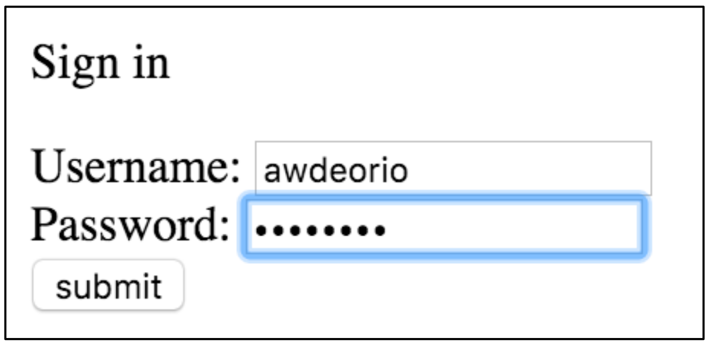
# Example: encrypting with a salt

- In practice, we store the algorithm, password and salt in the database

```
import hashlib
import uuid

algorithm = 'sha512'
password = 'bob1pass'
salt = uuid.uuid4().hex

m = hashlib.new(algorithm)
password_salted = salt + password
m.update(password_salted.encode('utf-8'))
password_hash = m.hexdigest()

print("$".join([algorithm,salt,password_hash]))

sha512$523bbfca143d4676b5ecfc8ee42aca6d$fae41640d635cb42c36
31e5a66a997e6f6ebfd25f6bb3f9777107d848c24bd2db9767242e803a8
81dbc5af73ddbf7ee80d1d855db2568061bfb2ca21fcf2dd5f
```

# Login

- User logs in



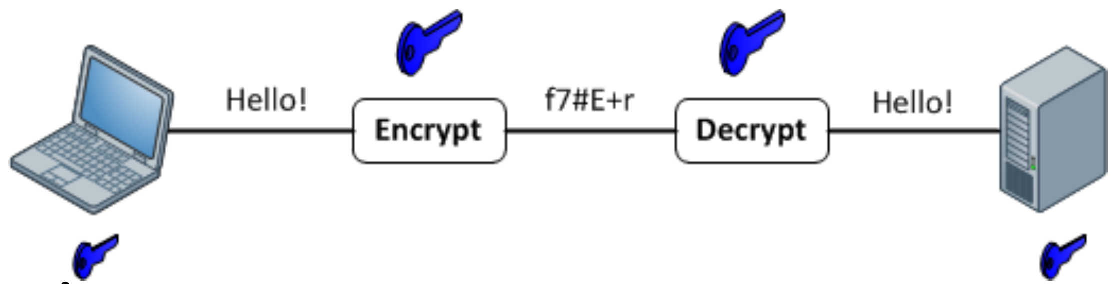

- Read password entry from database:
  sha512**$**<SALT>**$**<HASHED_PASSWORD>
- Compute sha512(<SALT> + input_password)
- Check if it matches <HASHED_PASSWORD>

# Agenda

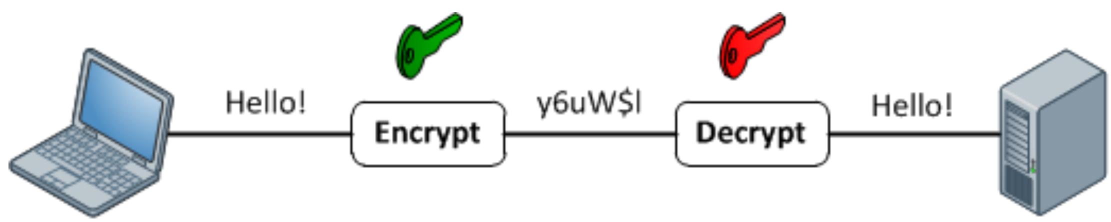
- Overview
- Symmetric encryption
- Asymmetric encryption
  - Public key infrastructure
- Cryptographic hash functions
- **Summary**

# Summary

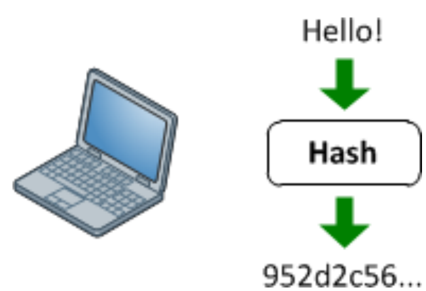
- Symmetric encryption
  - One key




- Asymmetric encryption
  - Two keys




- Cryptographic hash functions
  - No keys



# Summary: desirable properties

<u>Today</u>

- Confidentiality
  - Adversary should not understand message
- Sender authenticity
  - Message is really from the purported sender
- Message integrity
  - Message not modified between send and receive

# Summary: desirable properties

## Next time

- Freshness
  - Message was sent “recently”

## Later

- Anonymity
  - Attacker should not know that we are communicating
- Dark Web Lecture