Encryption

Announcements

- Partner policies
  - You will be able to switch partners between projects
  - You will have the chance to give feedback about your partner in case they didn't do their fair share
- Please fill out this form if you want to be assigned a group
  - https://forms.gle/EgkRcfH8E23FAEyy5
  - (by Sunday noon eastern pls)

Advice for partners

- Making friends is good. Picking a partner you haven't worked with before lets you make a new friend.

- Learn pair programming techniques
  - Pair programming != having a partner
  - Great resource: Google "martin fowler pair programming“
  - Or you know, take 481

- Also use the Slack for communication if you’re stuck

Announcements

- SSD forms: send to Kevin, Emily, and Andrew
  - (I've received around 4 so far)

- Must be in advance of Exam 1

Review: Sessions

- **Sessions** allow us to build statefulness on top of our web service
  - **Session** is a server-side data structure
  - **Cookie** is a client-side file

- **Cookies** work in tandem with a **Session** to maintain state across multiple request-response cycles
  - You login to a site, the server sets up a session for you (e.g., uniqname)
  - The server sets a cookie in your browser so it knows you're associated with the 'uniqname' session it created for you on each request
- **Flask** lets you handle sessions with built-ins

One Slide Summary: Encryption

- **Encryption** is the act of **transforming** a message from **plaintext** to **ciphertext** to *hide its contents* from **adversaries**
- **Encryption** works by applying a **key** to a **plaintext** to produce the **ciphertext**
  - **Symmetric encryption** uses *one key* to **encrypt** and **decrypt** a message
    - "symmetric" with respect to the key.
  - **Asymmetric encryption** uses *separate keys* to **encrypt** and **decrypt** a message
- We use encryption to achieve:
  - **Confidentiality:** adversaries cannot understand the ciphertext
  - **Authenticity:** we know who sends the message and who receives it
  - **Integrity:** we can tell if the message has been tampered with

Encryption

- Transform a message so that only authorized users/machines can read it

From Tidus

Secret Message:
"Zanarkand was pretty cool."

Encryption Algorithm

Encrypted Message:
"Wyhyngyht fyc bnaddo luum."

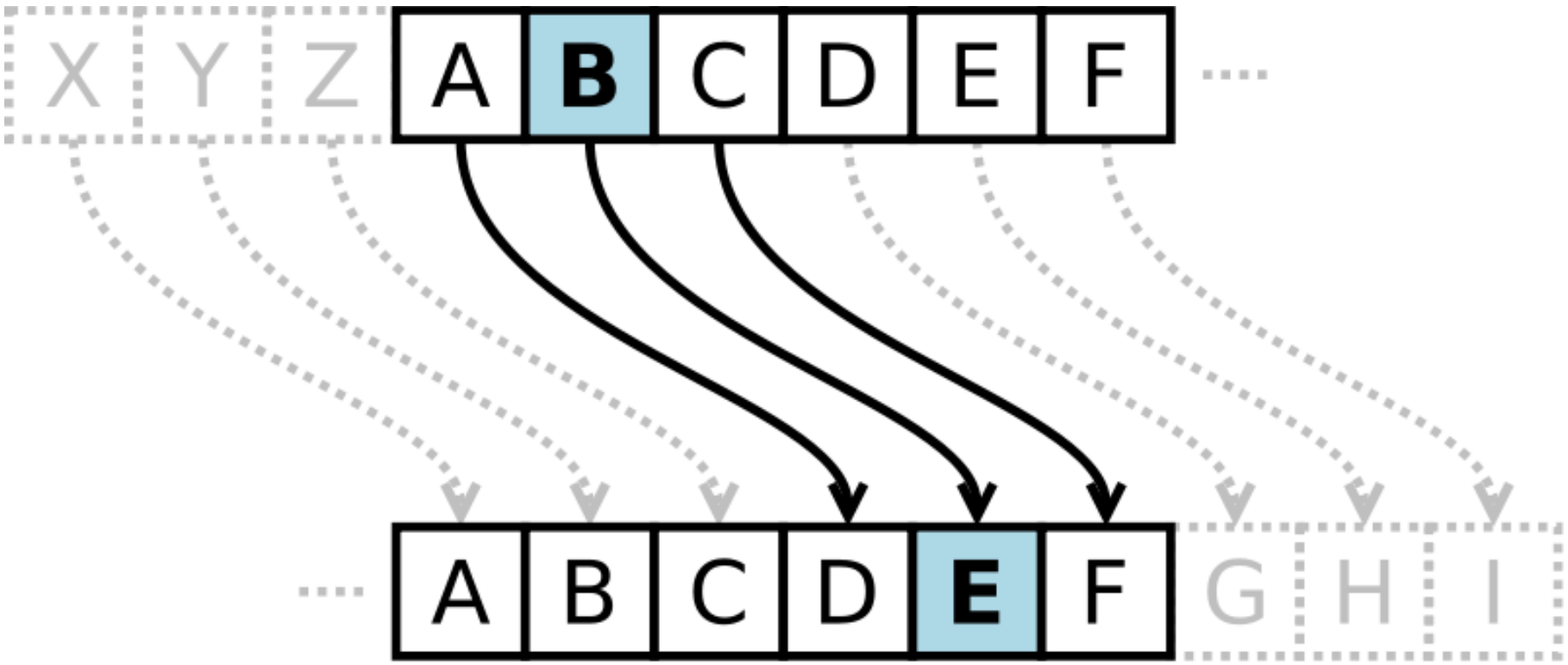
Decryption Algorithm

To Yuna

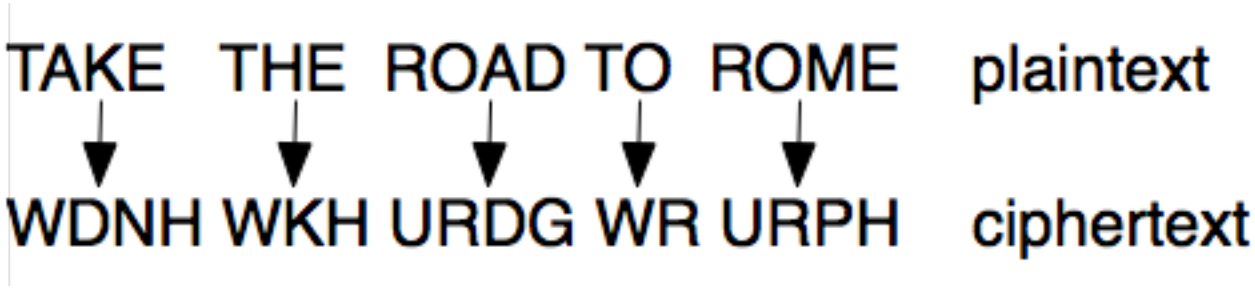
Secret Message:
"Zanarkand was pretty cool."

A brief history

- Caesar cipher rotates alphabet by 3



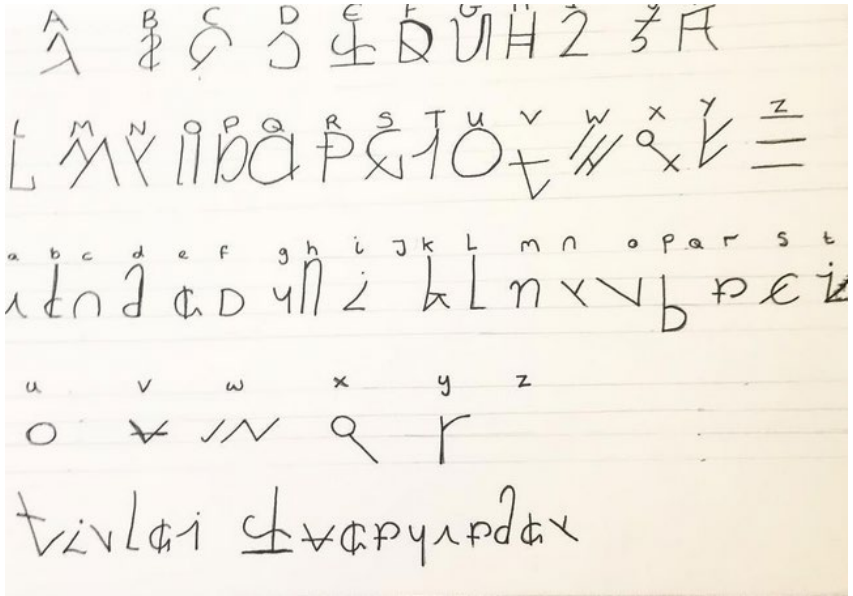
A brief history



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

Substitution ciphers

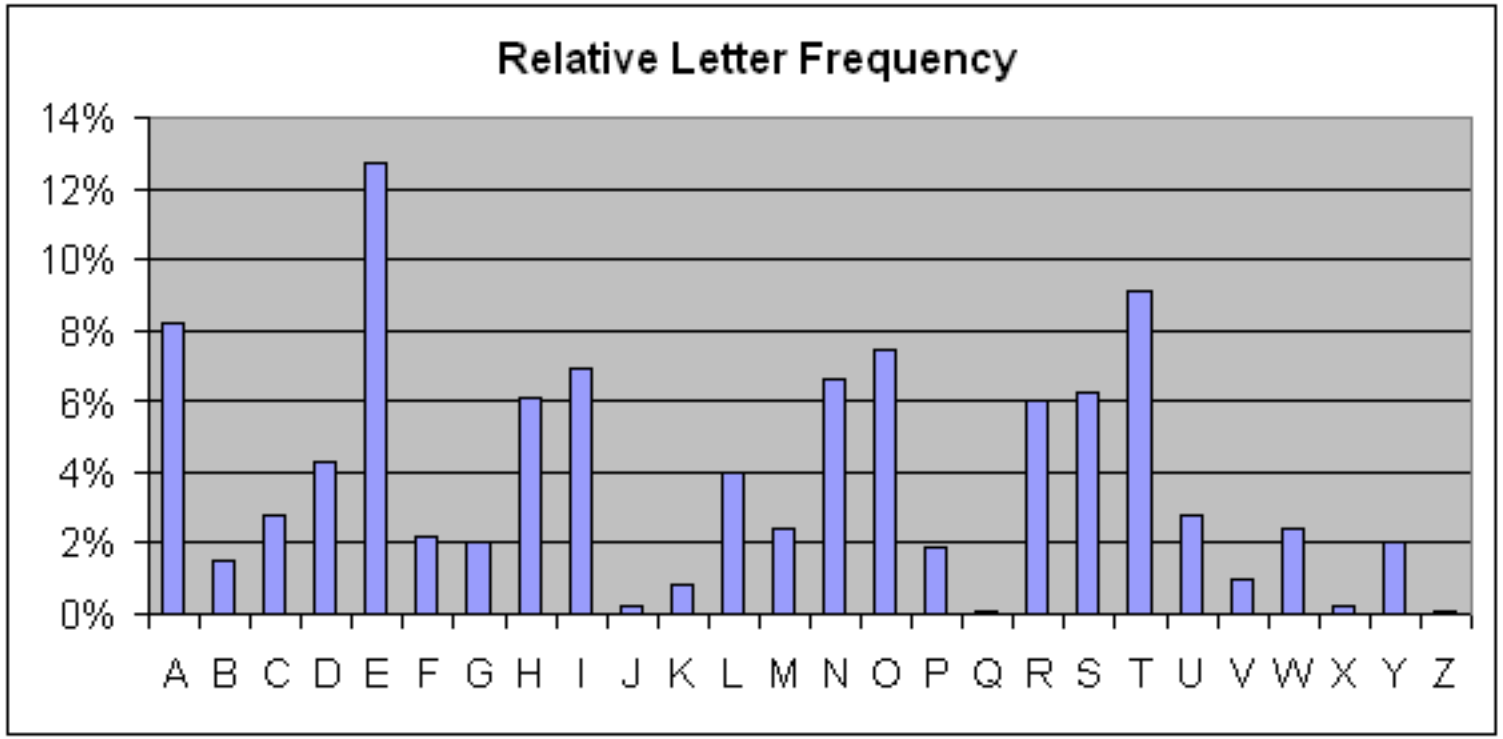
TAKE	THE	ROAD	TO	ROME	plaintext
↓	↓	↓	↓	↓	
WDNH	WKH	URDG	WR	URPH	ciphertext



- No need to shift 3 chars
  - *Rotate* by arbitrary number of characters
- You can also use a map from plaintext char to cipher char

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

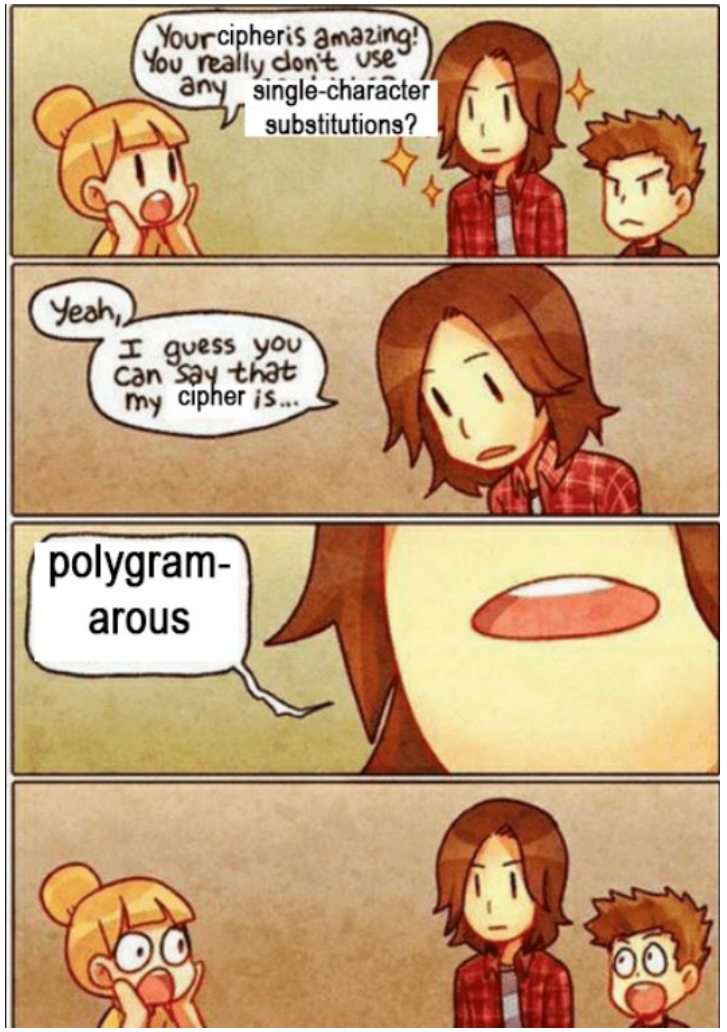
- How big is the substitution table?

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

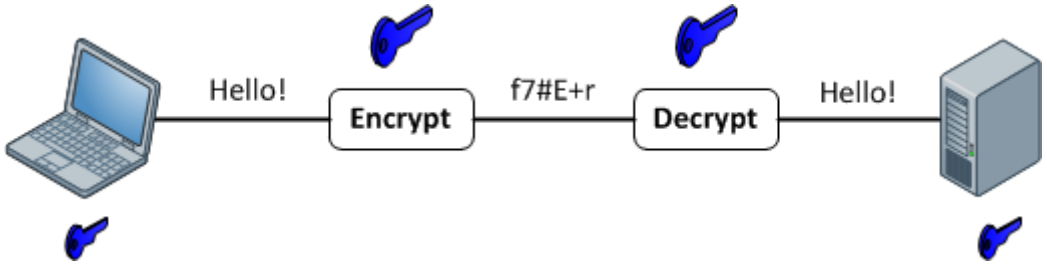


Encryption in words

- **Encryption** applies a **reversible function** to some piece of data **(plaintext),** yielding something unreadable **(ciphertext)**

- **Decryption** recovers the original plaintext data from the ciphertext

- The encryption/decryption *algorithm* assumed known; the *key* is secret



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

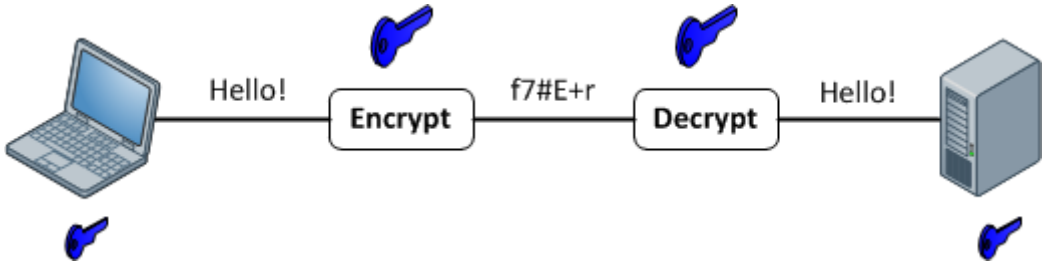
- Communication: What properties would you like?

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
- Recipient authenticity
  - Message is really read by the purported recipient

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})



Agenda

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

Symmetric Encryption

- Encryption and decryption key are the same
- Advantage: fast
- Disadvantage: how do you share the key?
  - Symmetric key encryption of the key?
    - Turtles all the way down…

AES: Advanced Encryption Standard

- 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

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

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

What makes a good symmetric cipher?

- Ideally: Nobody knows how to break it
  - **Fundamentally unbreakable** – no mathematical formula that gives you a statistical edge
    - Really difficult to achieve in practice

- “breakable”: if I have ciphertext, find the plaintext faster than guessing keys

- Is AES secure? No formal proof, but we haven't found any effective attacks yet.
  - Has the NSA? Would they tell us if they did?

Thought question

- A simple symmetric cipher:
  - Key: a number n
  - Algorithm: shift each letter forwards n places in the alphabet.
- e.g.
  - If key is 2: A -> C, Y -> A
  - HELLO -> JGNNQ

- Why would this not make a good cipher?

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

Asymmetric encryption

- Problem for symmetric: how to share key
- **Different keys** for encryption and decryption

- Keep decryption key private
- Share encryption key with everyone

Designing asymmetric cryptography

- Need mathematical way to make two related keys

- Hard as possible to figure out private key
  - Assume everyone knows public key

- Idea: "trapdoor functions"
  - b = f(a)
  - Make it hard to find *a* given *b*.
    - Recall: **fundamentally unbreakable**

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

- *NB: Modern cryptography depends on the assumption that we don't know how to factor primes!*

Building asymmetric cryptography

2020

$2^2 \cdot 5 \cdot 101$

- RSA algorithm (you don't need to know the details)
  - Uses prime factorization to derive matching public and private keys

Pick (large) prime integer p, q, compute $n = pq$

This makes n *part* of the *public* key.

Next, find $lcm(p-1, q-1)$ (called $\lambda(n)$)

$\lambda(n)$ is *part* of the *private* key.

Pick coprime value for e between 1 and $\lambda(n)$

e is also part of the *public* key.

Find $d \times e \equiv 1 \pmod{\lambda(n)}$ (*modular congruence*)

d is part of the *private* key

Private key:

$\lambda(n), p, q, \boldsymbol{d}$

Public Key:

n, e

Encryption of "m":

$m^e = c$

Decryption of "c":

$c^d = (m^e)^d = m$

From asymmetric to symmetric

- Asymmetric encryption is slow
  - Think of all that exponentiation and big integers
- Symmetric is fast
  - XORs galore

- Use asymmetric to communicate a symmetric key
- Then, continue with symmetric encryption

- “PGP” Pretty Good Privacy

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

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

Agenda

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

Public key infrastructure

- Asymmetric cryptography depends on sharing public keys
- How do you know that Amazon's public key actually comes from Amazon?
- Thought question: What could an attacker do to Amazon’s public key as it’s sent to you?

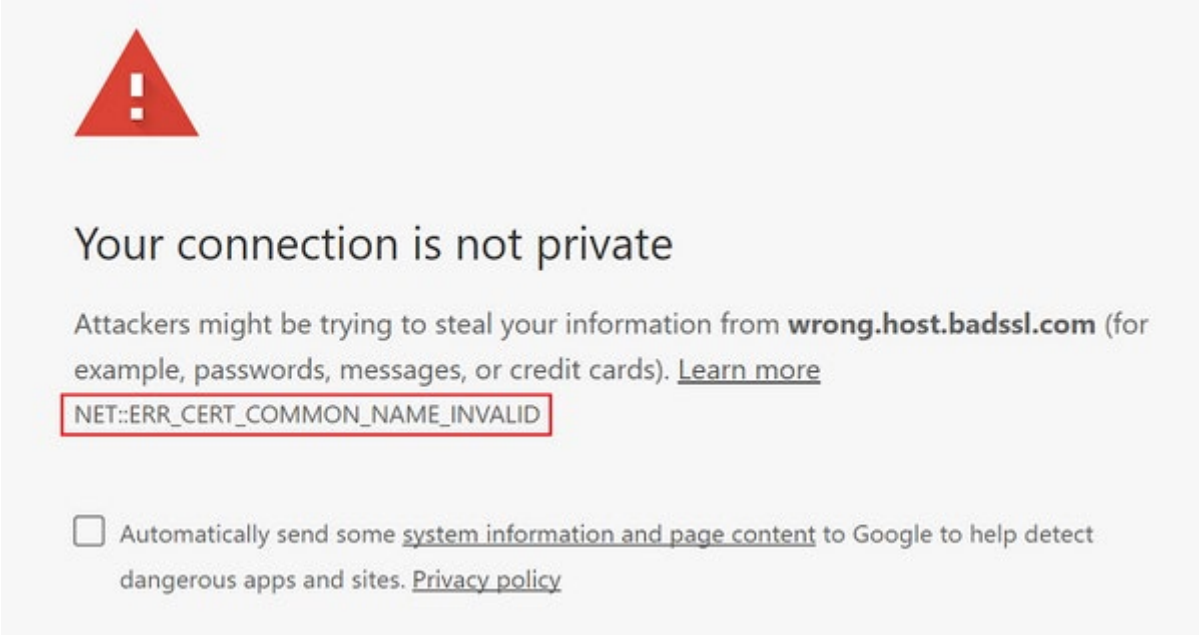
Certificate Authorities

- There are companies called "Certificate Authorities"
  - Verisign, Let's Encrypt (co-founded by Michigan prof Alex Halderman)

- They'll sign a certificate for you
  - Your public key will look legit to modern browsers!

Let's Encrypt

Certificate Authorities and Validation



Keychain Access

Click to unlock the System Roots keychain.

Search

Keychains

- login
- iCloud
- System
- System Roots

Category

- All Items
- Passwords
- Secure Notes
- My Certificates
- Keys
- Certificates

AAA Certificate Services

Root certificate authority

Expires: Sunday, December 31, 2028 at 6:59:59 PM Eastern Standard Time

This certificate is valid

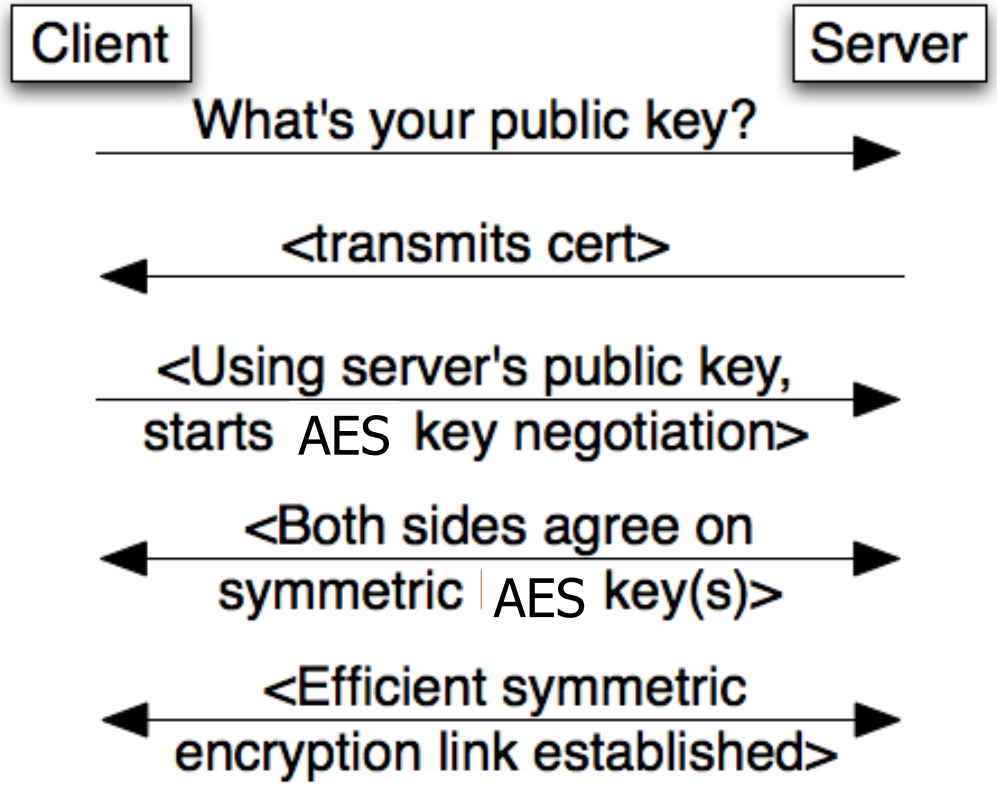
Name	Kind	Expires	Keychain
UCA Root	certificate	Dec 30, 2029, 7:00:00 PM	System Roots
USERTrust ECC...rtification Authority	certificate	Jan 18, 2038, 6:59:59 PM	System Roots
USERTrust RSA Certification Authority	ertificate	Jan 18, 2038, 6:59:59 PM	System Roots
UTN - DATACorp SGC	certificate	Jun 24, 2019, 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

Copy

164 items

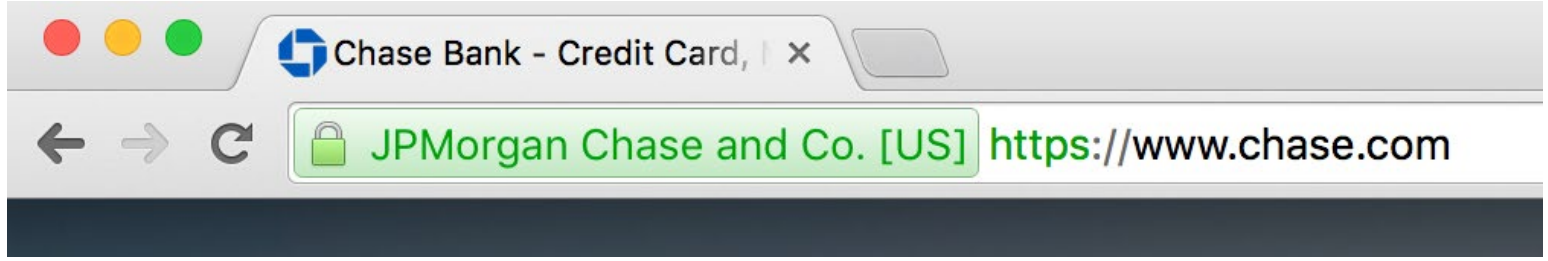
Client server interaction

- 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 so that the underlying HTTP request can be serviced
  - Server could be Nginx or Apache, etc.
  - Backend is `gunicorn` in EECS 485 AWS deployment

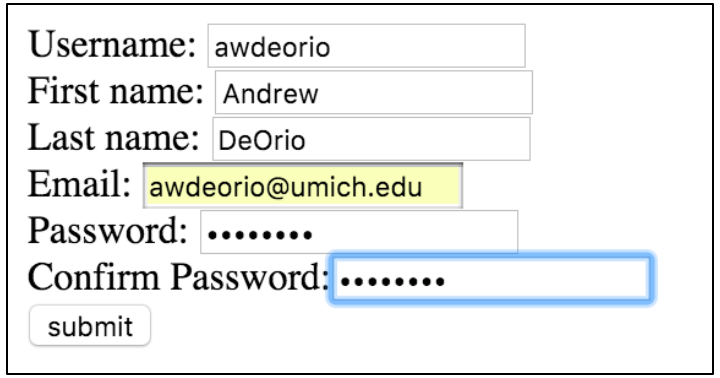
```python
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run(ssl_context=('cert.pem', 'key.pem'))
```

Agenda

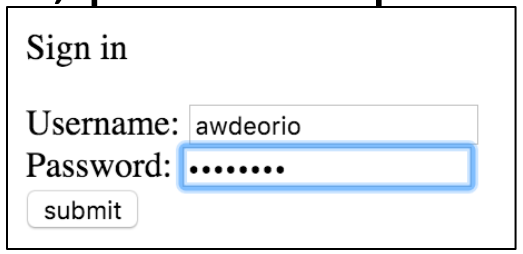
- Overview
- Symmetric encryption
- Asymmetric encryption
  - Public key infrastructure
- **Cryptographic hash functions**

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?

Hashing passwords

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

- Store hashed password
- User enters a password at login
  - Hash that entry, see if it matches the hash!

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)
```

```
af1bd47889bff89ccc889bc2aa61437c2ac90ee411618645bd4adbca1e02f8a2777
29093ea8ac094d3265352b75b12af1b4a50edd8fc5783cc0fac0411cde8c2
```

Partner question

- The most common passwords are:
  - 123456
  - 123456789
  - qwerty
  - password
  - 11111111
  - 12345678

- If someone gets a copy of my password database, what is a way they can work backwards from the hashes to the real passwords?

Rainbow tables

- Compute a table of passwords and hashes

Plaintext	MD5 Checksum
123456	e10adc3949ba59abbe56e057f20f883e
123456789	25f9e794323b453885f5181f1b624d0b
password	5f4dcc3b5aa765d61d8327deb882cf99
adobe123	7558af202997483d3afef3bb2b5a709d
12345678	25d55ad283aa400af464c76d713c07ad
qwerty	d8578edf8458ce06fbc5bb76a58c5ca4
1234567	fcea920f7412b5da7be0cf42b8c93759
111111	96e79218965eb72c92a549dd5a330112

- Can use this against many different databases

Protecting against rainbow tables

- 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)
```

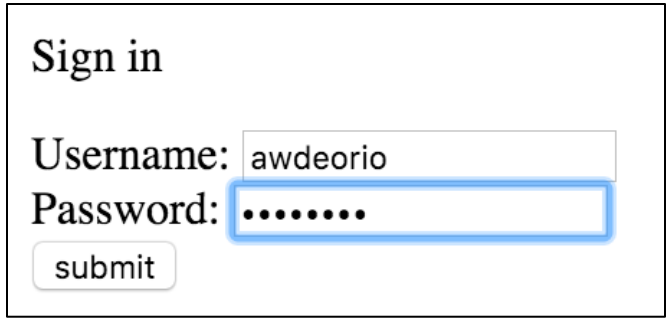
Example: encrypting with a salt

- 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$523bbfca143d4676b5ecfc8ee42aca6d$fae41640d635cb42c3631e5a66a997e6f6ebfd2
5f6bb3f9777107d848c24bd2db9767242e803a881dbc5af73ddbf7ee80d1d855db2568061bfb2ca
21fcf2dd5f
```

Login

- User logs in



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