

# **CSE 410/565: Computer Security**

Instructor: Dr. Ziming Zhao

# Cryptographic Topics Covered

- What we've discussed so far:
  - symmetric encryption
  - message authentication codes
  - hash functions; password hashing
  - public-key encryption
  - digital signatures
- Today's agenda:
  - public key certificates; public key infrastructure
  - (pseudo) random numbers and generators

# **Public Key Certificates**

# Secure Communication

- As previously discussed, we want to use fast **symmetric key cryptography** for secure communication
- When there is no pre-established relationship and shared key, public-key **cryptography** is used to agree on the key
  - the idea is for one party A to choose a key  $k$  and send it encrypted to another party B using B's public key
    - A sends  $\text{Enc}_{\text{pk}_B}(k)$  to B
  - this logic forms the basis of different protocols used in practice (e.g., TLS)
- The question of (public) **key authenticity** arises

# Public Keys and Trust



Alice

public key  $pk_A$   
secret key  $sk_A$

Bob

public key  $pk_B$   
secret key  $sk_B$

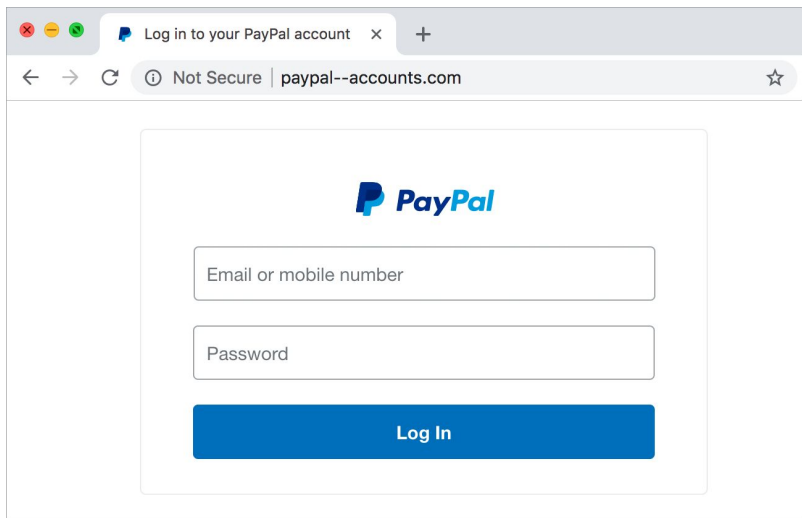
- If we want to use public-key cryptography, we are facing the **key distribution problem**
  - how/where are public keys stored?
  - how do I obtain someone's public key?
  - how can Bob know or "trust" that  $pk_A$  is indeed Alice's public key?

# Public-Key Certificates

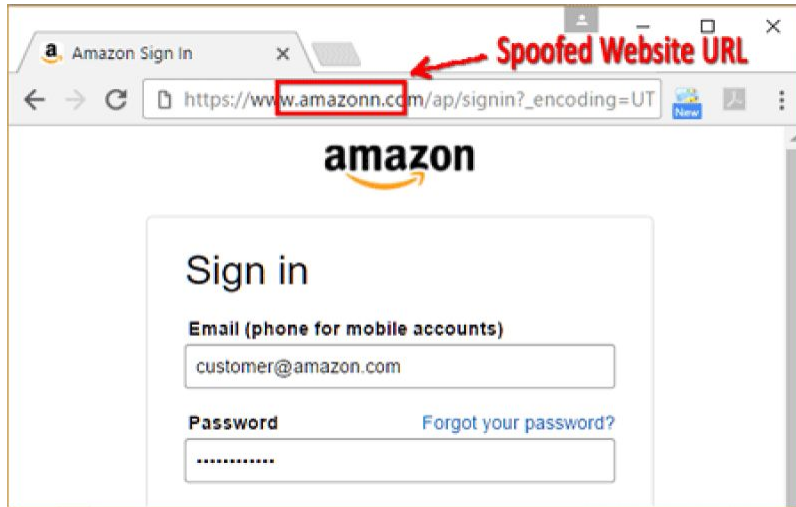
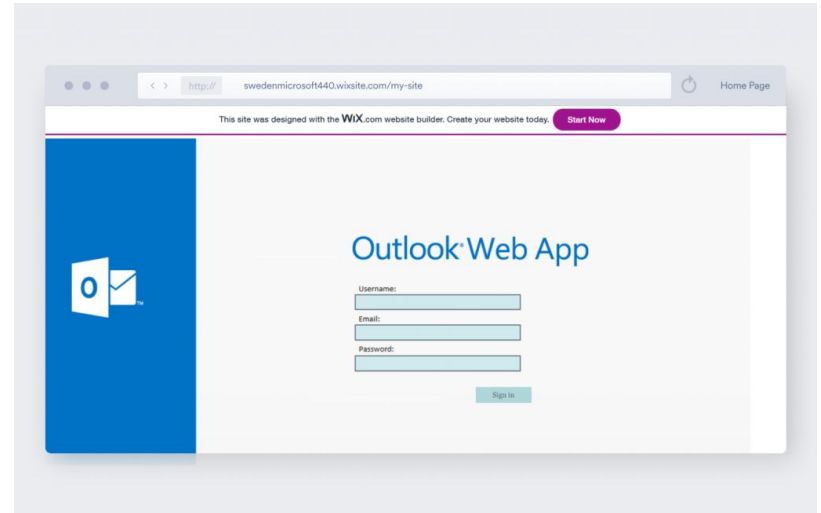
- Distribution of public keys can be done
  - by public announcement
    - a user distributes her key to recipients or broadcasts to community
  - through a publicly available directory
    - can obtain greater security by registering keys with a public directory
- Both approaches don't protect against forgeries
- Digital certificates are used to address this problem
  - a certificate **binds identity (and/or other information) to a public key**

# Public-Key Certificates

- (Root of trust) Assume there is a trusted central authority  $CA$  with a known public key  $pk_{CA}$
- $CA$  produces certificate for Bob as  $cert_B = sig_{CA}(pk_B || \text{Bob})$
- Bob distributes  $(pk_B, cert_B)$
- Alice can verify that her copy of Bob's key is genuine
- This technique is used in many applications
  - TLS/SSL, ssh, email, IPsec, etc.



# Phishing Websites





# Website Identity

- When you go to a site that uses HTTPS (connection security), the website's server uses a **certificate** to prove the website's **identity** to browsers, like Chrome.
- Anyone can create a certificate claiming to be whatever website they want. To help you stay on safe on the web, a good browser requires websites to use certificates from trusted organizations.

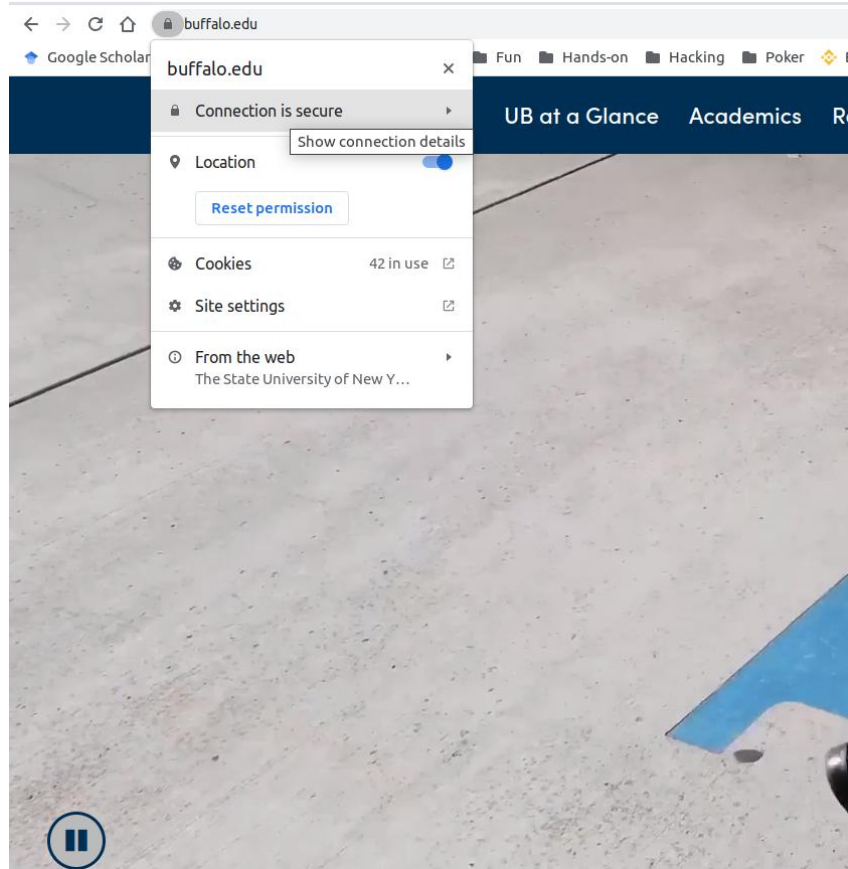
# X.509 Identity Certificates

- Distinguished Name of user
  - C=US, O=Lawrence Berkely National Laboratory, OU=DSD, CN=Mary R. Thompson
- DN of Issuer
  - C=US, O=Lawrence Berkely National Laboratory, CN=LBNL-CA
- Validity dates:
  - Not before <date>, Not after <date>
- User's public key
- Signed by CA

# Certificate Authority

- A trusted third party - must be a secure server
- Signs and publishes X.509 Identity certificates
- Revokes certificates and publishes a Certification Revocation List (CRL)
- Many vendors
  - OpenSSL - open source, very simple
  - Netscape - free for limited number of certificates
  - Entrust - Can be run by enterprise or by Entrust
  - Verisign - Run by Verisign under contract to enterprise
  - RSA Security - Keon servers

# Website Identity



# Website Identity

Certificate Viewer: www.buffalo.edu ×

**General** Details

**Issued To**

Common Name (CN)	www.buffalo.edu
Organization (O)	<Not Part Of Certificate>
Organizational Unit (OU)	<Not Part Of Certificate>

**Issued By**

Common Name (CN)	R3
Organization (O)	Let's Encrypt
Organizational Unit (OU)	<Not Part Of Certificate>


**Validity Period**

Issued On	Wednesday, January 18, 2023 at 3:44:14 PM
Expires On	Tuesday, April 18, 2023 at 4:44:13 PM








**Fingerprints**

SHA-256 Fingerprint	7E 31 E6 7D 33 92 8A 54 BF 6E 8C 6B 15 D4 30 7B 94 04 0E 73 54 9C 48 3F 37 62 A5 EA EA 3C 44 92
SHA-1 Fingerprint	BE E4 B5 48 7F 0B 5C 77 EA 38 F4 4F FD D1 CD B5 29 A1 63 72






# Website Identity

 **Settings** 🔍 Search settings



---

-  You and Google
-  Autofill
-  **Privacy and security**
-  Appearance
-  Search engine
-  Default browser
-  On startup

---

-  Languages
-  Downloads
-  Accessibility
-  System
-  Reset settings

---

-  Extensions [🔗](#)
-  About Chrome

---

← Manage certificates

Your certificates   Servers   Authorities   Others

You have certificates on file that identify these certificate authorities [Import](#)

org-AC Camerfirma S.A.	▼
org-AC Camerfirma SA CIF A82743287	▼
org-ACCV	▼
org-Actalis S.p.A./03358520967	▼
org-AddTrust AB	▼
org-AffirmTrust	▼
org-Agencia Catalana de Certificacio (NIF Q-0801176-I)	▼
org-Amazon	▼
org-AS Sertifitseerimiskeskus	▼

## Self-signed certificate?

- Technically, anyone can create their own SSL certificate by generating a public-private key pairing and including all the information mentioned above. Such certificates are called self-signed certificates because the digital signature used, instead of being from a CA, would be the website's own private key.
- But with self-signed certificates, there's no outside authority to verify that the origin server is who it claims to be. Browsers don't consider self-signed certificates trustworthy and may still mark sites with one as "not secure," despite the https:// URL. They may also terminate the connection altogether, blocking the website from loading.

# **Public Key Infrastructure**



# Public Key Infrastructure

- Possibly the biggest challenge in public-key cryptography is ensuring the **authenticity of public keys**
  - Alice wants to encrypt a message for Bob using his public key
  - but Alice doesn't know Bob personally
- We have already seen that public keys can be managed through the use of **certificates**
  - there is a trusted certification authority with a known public key
  - the CA issues certificates by signing a user's identity along with her public key
  - Bob's public key  $pk_B$  is authentic if it matches his key in  $cert_B = sig_{CA}(B, pk_B)$

# Public Key Infrastructure

- A **public-key infrastructure** (PKI) is a system for managing trust in the public keys through the use of certificates
  - it is the basis of a pervasive security infrastructure whose services are implemented and delivered using public-key techniques
- Ideally, a PKI should function without active participation of the user
  - when we say that a network user Alice performs various operations, it implies that her software does this
  - Alice might not be even aware of the PKI-related procedures
- A **PKI's goal** is to eliminate the need for users to share precomputed symmetric keys and enable the use of public-key cryptography

# Public Key Infrastructure

- There are many **components to a PKI**
  - **certificate issuance**
    - before a certificate can be issued, the identity and credentials of the user must be verified using non-cryptographic means
    - a secure procedure must be used to generate the public and private keys for the certificate's owner
  - **certificate revocation**
    - this is done before a certificate's expiration date under unforeseen circumstances
    - for example, if a private key is lost or stolen
    - additional infrastructure is need to recognize revoked certificates

# Public Key Infrastructure

- PKI components (cont.)
  - key backup/recovery/update
    - key backup refers to secure storage of users' private keys by the administrator of the PKI
    - key recovery is a protocol that allows a lost or forgotten key to be restored or re-activated
    - key update occurs when a key is to be changed (e.g., a certificate is about to expire)
  - timestamping
    - the times at which a key is issued, revoked, or updated may be important
    - such timestamps are often included in the certificate

# Public Key Infrastructure

- Once a PKI is built and operational, it allows various applications to be built on top of it
  - such applications can be called **PKI-enabled services**
- Examples of **PKI-enabled services** include
  - **secure communication**
    - secure email protocols include Secure Multipurpose Internet Mail Extensions (S/MIME) and Pretty Good Privacy (PGP)
    - secure web service access is provided through Secure Sockets Layer (SSL) or Transport Layer Security (TLS)
    - secure virtual private networks (VPNs) use the Internet Protocol Security (IPsec) protocols

# Public Key Infrastructure

- Examples of **PKI-enabled services** (cont.)
  - **access control**
    - access control provides the means of managing user privileges through authentication, authorization, and delegation
    - access control normally requires user authentication via a password or cryptographic identification scheme
  - **privacy architecture**
    - a privacy architecture permits the use of anonymous or pseudonymous credentials
    - such credentials (or certificates) allow an individual to show membership or another property without specifying their identity

# Access control/Authentication without Password

Registration Stage

Alice

Server

Generate a pair of  
public and private keys

Upload Alice's public key



# Access control/Authentication without Password

## Authentication Stage

Alice

Server

I am Alice, and I want to be authenticated



Encrypt random number  $S$   
using Alice's private key



Here it is  $C = \text{Enc}(\text{PrivateK}, S)$



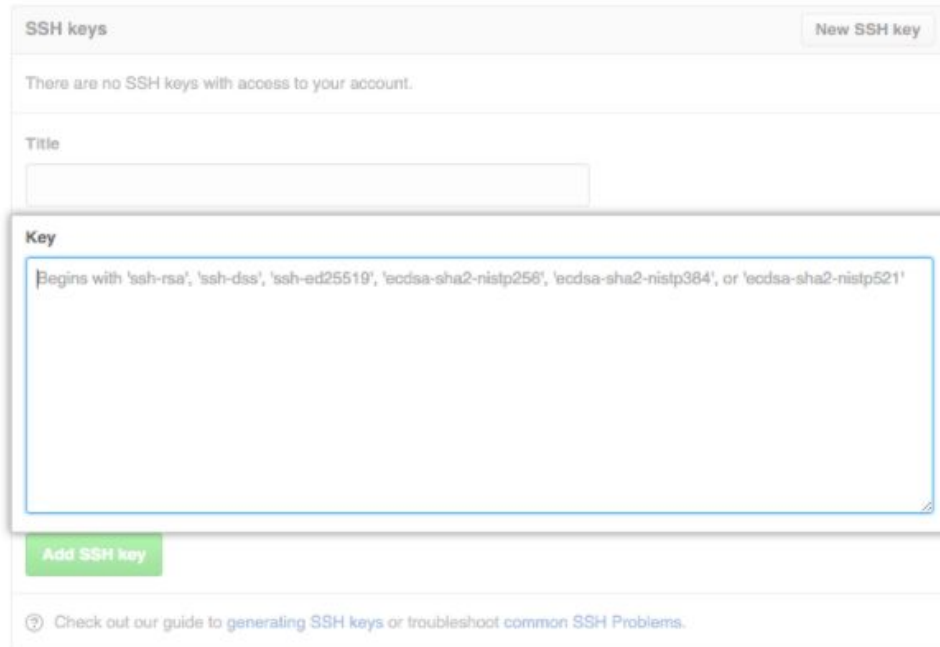
Verifies if  
 $\text{Dec}(\text{PublicK}, C) == S$



# GitHub/SSH Example

Push code without inputting GitHub password each time.

- Login GitHub website with Username/Passwd
- Add RSA public key in your profile



The screenshot shows the GitHub 'SSH keys' management interface. At the top, there is a header 'SSH keys' and a 'New SSH key' button. Below this, a message states 'There are no SSH keys with access to your account.' A 'Title' input field is present. The main section is titled 'Key' and contains a large text area with a placeholder that reads: 'Begins with 'ssh-rsa', 'ssh-dss', 'ssh-ed25519', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', or 'ecdsa-sha2-nistp521''. Below the text area is a green 'Add SSH key' button. At the bottom, there is a link to a guide: 'Check out our guide to generating SSH keys or troubleshoot common SSH Problems.'

# Certificates

- **Certificates** are important building blocks of PKIs
- It is generally assumed that there is a **trusted CA**
  - each user has access to an authentic copy of the CA's public key
- In its simplest form, a certificate is a CA's signature on an identity and the identity's public key
- A valid CA's signature of this form is treated as a confirmation that the public key belongs to that identity
  - i.e., it is assumed that the CA verifies the identity before signing any given key
- **X.509 v3** is a popular type of certificate

# Certificates

- X.509 certificates contain the following fields:
  - data
    - version
    - serial number
    - signature algorithm
    - issuer name
    - validity period
    - subject name
    - subject public key and public key algorithm
    - optional fields
  - signature on all of the above field

# Certificates

- X.509 certificates were originally defined using X.500 names for subjects
- **X.500 names** have a hierarchical format:
  - C = US
  - O = University at Buffalo
  - OU = Department of Computer Science and Engineering
  - CN = Ziming Zhao
- here C denotes country, O denotes organization, OU denotes organizational unit, and CN denotes common name
- This format ensures that **everyone has a unique global name**
  - it could be used as a global phone directory
- No global X.500 directory exists today

# Certificates

- Example X.509 certificate

## Data:

Version: 3

Serial Number: 32:73:4D:44:25:6E:73:C2

Certificate Signature Algorithm: PKCS #1 SHA-256 With RSA Encryption

Issuer: CN = Google Internet Authority G3

O = Google Trust Services

C = US

## Validity:

Not Before: September 25, 2018, 7:43:00 AM GMT

Not After : December 18, 2018, 7:43:00 AM GMT

Subject: C = US, ST = California, L = Mountain View,

O = Google LLC,

CN = www.google.com

# Certificates

- Example X.509 certificate (cont.)

Subject Public Key Info:

Public Key Algorithm:

Algorithm Identifier: Elliptic Curve Public Key

Algorithm Parameters: ANSI X9.62 elliptic curve prime256v1 (aka secp256r1, NIST P-256)

Subject's Public Key:

Key size: 256 bits

Base point order length: 256 bits

Public value: 04 5a cd 02 99 d4 9a a0 ab 0f 9f c0 9e d1 c5 2d ...

Certificate Signature Value:

Size: 256 Bytes / 2048 Bits

88 14 32 8e 8f 32 95 8f d4 a4 85 0d 2f 55 23 78 4b 58 6d cf b4 5e 1c 2b a8 3f  
1c 6c 83 7e 6a fa ...

# Certificates

- **Certificate life-cycle management** has several phases, which include:
  - registration
  - key generation (and backup), key distribution
  - certificate issuance
  - certificate retrieval and validation
  - certificate expiration
- Additionally, we might have:
  - key update
  - key recovery
  - certificate revocation

# Certificates

- During the **validation**, the following steps take place
  - verify the integrity and authenticity of the certificate by verifying the CA's signature
    - it is assumed that the CA's keys is known a priori or can be reliably verified using external resources
  - verify that the certificate has not expired
  - verify that the certificate has not been revoked
  - if applicable, verify that the certificate's usage complies with the policy constraints specified in the certificate's optional fields



# Certificates

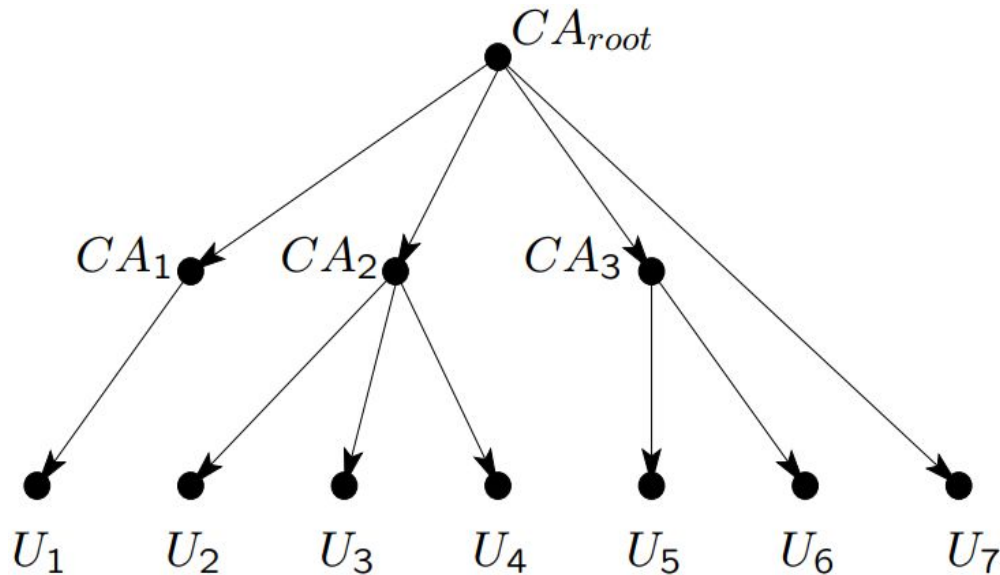
- A PKI needs a mechanism to verify that a certificate has not been revoked prior to its expiration date
- A **certificate revocation list** (CRL) is the most common technique
  - a CRL is a list of the serial numbers of certificates that are revoked but not expired
  - this list is prepared by the CA and is signed
  - it is updated periodically and is made available at a public directory
- For efficiency reasons, **delta CRLs** can be used
  - instead of containing all revoked certificates, delta CRLs contain the changes since the most recent previously issued CRL or delta CRL
- Alternatively, an **online certificate status protocol** can be used to query a certificate's status in real time

# Trust Models

- Often there is **more than one CA** that can sign certificates
  - a certificate can be verified by following a certificate path from a trusted CA to a given certificate
  - each certificate in the path is signed by the owner of the previous certificate in the path
  - if all certificates in the path can be verified, the last certificate is considered to be valid
- A **trust model** specifies the way in which a certificate path should be constructed, e.g.,
  - for example, strict hierarchy; networked PKIs; web browser model; and user-centric model (or web of trust)

# Trust Models

- **Strict hierarchy model**
  - there is a single root CA that has a self-signed self-issued certificate
    - the root CA is called **trust anchor**
  - the root CA may issue certificates for other CAs
  - any CA can issue certificates for end users



# Trust Models

- **Strict hierarchy model**
  - what is the meaning of a directed edge?
  - usage example: Alice would like to verify a certificate of user  $U_3$ 
    - $U_3$  sends to Alice
    - Alice performs certificate path validation

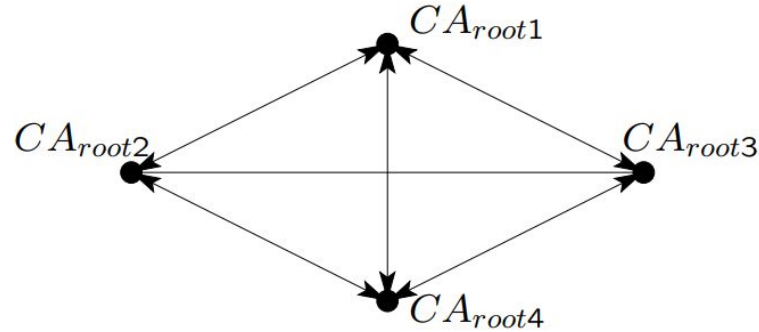
# Trust Models

- Networked PKIs model

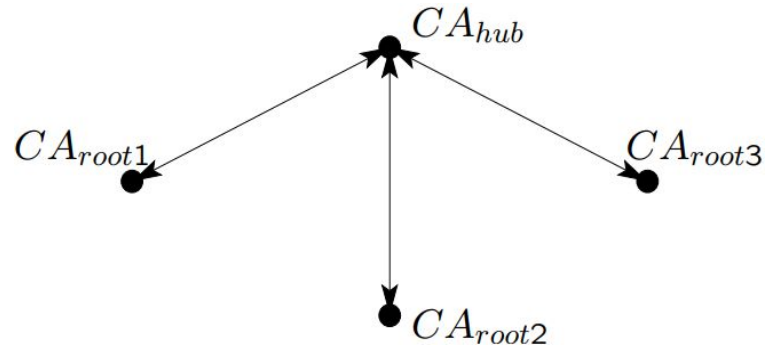
- sometimes it might be desirable to connect root CAs of two or more different PKI domains
  - this can be called **PKI networking** and it creates one large PKI with users in different domains
- cases when one CA signs the certificate of another CA are called **cross-certification**
- in **mesh configuration**, there are several root CAs and all of them cross-certify one another
  - the number of cross-certifications for n root CAs is \_\_\_\_\_
- in **hub-and-spoke configuration**, root CAs each cross-certify independently with a new hub CA
  - the number of cross-certifications for n root CAs is \_\_\_\_\_

# Trust Models

- Networked PKIs model (cont.)
  - example mesh configuration



- example hub-and-spoke configuration



# Trust Models

- Networked PKIs model (cont.)

- to validate Bob's certificate, Alice must be able to find a path of certificates from her trust anchor  $CA_{rooti}$  to Bob
  - this process is called **path discovery**
- assume that Bob's certificate is signed by  $CA_{rootj}$
- what path does Alice construct in **mesh configuration**?
- in **hub-and-spoke configuration**?

# Trust Models

- **Web browser model**

- most web browsers come preconfigured with a set of independent root CAs
- all of them are considered to be trusted by a user of the browser
- **security considerations** for this model:
  - users normally don't have information about the security of these pre-configured root CAs
  - the list is editable, but many users are not even aware of it
  - there might be no mechanism to revoke a root CA from a web browser
  - there might be no automated way to update root CAs' certificates
  - if permitted, most users choose to proceed with expired certificates



# Trust Models

- **Pretty Good Privacy (PGP)**

- PGP is used for email, where each user is her own CA
- a PGP certificate contains an email address (UID), a public key (PK), and one or more signatures on this (UID, PK) pair
- when a Alice creates her key, she first self-signs it:

$\text{cert}(\text{Alice}) = (\text{data}, \text{sig}_A(\text{data})), \text{ where}$

$\text{data} = (\text{UID} = \text{alice@buffalo.edu}, \text{PK} = 0xBEEF1234)$

- later, other users may add their signatures to Alice's key, so she has  
 $\text{cert}(\text{Alice}) = (\text{data}, \text{sig}_A(\text{data}), \text{sig}_B(\text{data}), \text{sig}_C(\text{data}), \dots)$

# Trust Models

- **Pretty Good Privacy (PGP)** (cont.)
  - when Bob wants to sign Alice's key, he needs to
    - retrieve a copy of Alice's key (from Alice or a key server)
    - verify Alice's identity
    - ensure that the key he has for Alice is the same as what Alice has
      - this is done by comparing the fingerprints of Alice's key and Bob's version of Alice's key
    - sign Alice's key
    - send the updated certificate to Alice or a key server
  - if Bob performed all checks, he is likely to trust the key
  - he can encrypt emails to Alice or receive signed emails from her

# Trust Models

- **Pretty Good Privacy (PGP)** (cont.)
  - Alice keeps a collection of certificates she obtained from different sources in a data structure called a **keyring**
  - she is able to declare how much she trusts the owner of a certificate
    - often the choices are implicitly trusted, completely trusted, partially trusted, or untrusted
  - Alice's own key is implicitly trusted
  - if Alice is convinced that Bob's public key is valid and Bob would not to sign invalid keys, she declares Bob's key completely trusted
  - she can also set how the trust of unknown keys is computed
    - e.g., she can set that keys with the distance larger than 3 from her should not be trusted

# Trust Models

- PGP web of trust
  - each user can set the trust of a key at signing time and also choose how trust of unsigned keys is computed
  - key servers are a convenient way of store and access keys
    - examples: keyserver.pgp.com, pgp.mit.edu, etc.
    - different key servers synchronize their datasets

# S/MIME Client Certificates

- There are other certificate or PKI architectures for email
- UB offers [client certificates](#) for signing and encryption
  - the key is generated by UBIT and is signed by UB's key
  - you can trust that signed emails from UB accounts were sent by the account owners
  - see <https://email.buffalo.edu/ClientCertificate.html> for more information

# Future of PKI

- There are several difficulties that prevent a large-scale deployment of PKIs
  - the first problem is who should be responsible for development, maintenance, and regulation of PKIs
  - the second problem is what standards should be used in PKIs
  - the third problem is that different PKIs are needed in different environments
  - also, a lack of PKI-compatible applications is slowing the deployment of PKIs
    - this is a chicken-and-egg problem

# Are There Alternatives?

- **Emerging technologies** might provide alternative solutions to having PKI
  - one example is **contract signing using blockchain technology**
    - blockchain offers immutable storage organized by time
    - contract signing could be realized using digital signatures
      - this requires having an authentic copy of the signer's key as otherwise the security guarantees don't hold
    - by reliably storing one's consent to executing the contract on the blockchain, we could have similar legal protection
    - laws and regulations often lag behind technological changes

# Summary

- **Public key infrastructure** targets solving the problem of public-key management between users who don't know each other
  - this is often addressed through the use of certificates
  - many different architectures for building trust exist
  - different applications use different models
    - multiple CA certificates on the web
    - web of trust in PGP
- No large-scale public PKI is currently in place



# Random Numbers

# Random Numbers

- All cryptographic constructions that are non-deterministic or produce key material require **randomness**
  - choosing symmetric key as a random string
  - choosing large prime and other numbers for public-key constructions
  - choosing padding or other means of randomizing encryption
- What do we expect from a **random bit sequence**?
  - **uniform distribution**: all possible values are equally likely
  - **independence**: no part of the sequence depends on its other parts
- Where do we find randomness?

# Random Numbers

- Randomness can be gathered from **physical, unpredictable processes**
- Example **sources of true randomness**
  - least significant bits of time between key strokes
  - noise from a mouse, video camera, and microphone
  - variation in response times of raw read requests from a disk
- Amount of required randomness may not be small
  - example: choosing a 1024-bit prime
- Instead of a **true random number generator (TRNG)** we can use a **pseudo-random number generator (PRNG)**

# Pseudo-Random Numbers

- A **pseudo-random generator** is an algorithm that
  - takes a short value, called a seed, as its input
  - produces a long string that is statistically close to a uniformly chosen random string
  - for a  $k$ -bit long seed, a PRG has period of at most  $2^k$  bits
  - formally,  $\text{PRG} : \{0, 1\}^k \rightarrow \{0, 1\}^{\ell(k)}$  for some  $\ell(k) > k$
- The **security requirement** is that a computationally bounded adversary cannot tell the output of a PRG apart from a truly random string of the same size
  - in practice, a number of statistical tests are used to test the strength of a PRG

# Pseudo-Random Numbers

- PRGs are deterministic
  - the output is always the same on the same seed
  - for cryptographic purposes, it is crucial that the seed is hard to guess
    - i.e., use strong true randomness to generate a seed
- One of uses of a PRG is for **symmetric key stream ciphers**
  - two parties share a short key, which is used as a seed to a PRG
  - the resulting pseudo-random key string is used to encipher the data
  - portions of the pseudo-random string cannot be reused!

# Pseudo-Random Numbers

- Example of a PRG
  - symmetric block ciphers, such as AES, can be used as PRGs
  - given a key  $k$ , produce a stream as  $\text{Enc}_k(0), \text{Enc}_k(1), \dots$ , where Enc is block cipher encryption
- There are various tests that can be run on PRGs to determine how close the output to a uniformly chosen string
- Of particular importance to cryptographically secure PRG is the next-bit test
  - given  $m$  bits of a PRG's output, it is infeasible for any computationally-bounded adversary to predict the  $m + 1$ th bit with probability non-negligibly greater than  $1/2$

# Random and Pseudo-Random Numbers

- Regardless of how randomness was produced, it is absolutely **crucial** that you use good randomness
  - insufficient amount of randomness leads to predictable keys
  - this is especially dangerous for long-term signing keys
- **Examples of poor randomness** in cryptographic applications
  - CVE-2006-1833: Intel RNG Driver in NetBSD may always generate the same random number, Apr. 2006
  - CVE-2007-2453: Random number feature in Linux kernel does not properly seed pools when there is no entropy, Jun. 2007
  - CVE-2008-0166: OpenSSL on Debian-based operating systems uses a random number generator that generates predictable numbers, Jan. 2008

# Linux /dev/random and /dev/urandom

- Both /dev/random and /dev/urandom are devices to provide a cryptographically secure pseudorandom number generator.
- /dev/random blocks when there is not enough entropy available, which can cause performance issues in certain situations. Entropy refers to the amount of randomness that can be gathered from the environment, such as user input and hardware events, to generate secure random numbers.
- /dev/urandom does not block and will always generate random numbers using a cryptographic algorithm that uses a cryptographic key to generate random numbers. This means that /dev/urandom can generate random numbers much faster than /dev/random. However, in some situations, if there is not enough entropy available, /dev/urandom may use weaker sources of randomness, which can potentially reduce the security of the generated random numbers.



# Linux Random Number Generator 2.6.10

- The Linux random number generator is part of the kernel of all Linux distributions and is based on generating randomness from entropy of operating system events.
- The output of this generator is used for almost every security protocol, including TLS/SSL key generation, choosing TCP sequence numbers, and file system and email encryption.

# Linux Random Number Generator 2.6.10

## Analysis of the Linux Random Number Generator

Zvi Gutterman

Safend and The Hebrew University of Jerusalem

Benny Pinkas

University of Haifa

Tzachy Reinman

The Hebrew University of Jerusalem

### Abstract

*Linux is the most popular open source project. The Linux random number generator is part of the kernel of all Linux distributions and is based on generating randomness from entropy of operating system events. The output of this generator is used for almost every security protocol, including TLS/SSL key generation, choosing TCP sequence numbers, and file system and email encryption. Although the generator is part of an open source project, its source code (about 2500 lines of code) is poorly documented, and patched with hundreds of code patches.*

*We used dynamic and static reverse engineering to learn the operation of this generator. This paper presents a description of the underlying algorithms and exposes several security vulnerabilities. In particular, we show an attack on the forward security of the generator which enables an adversary who exposes the state of the generator to compute previous states and outputs. In addition we present a few cryptographic flaws in the design of the generator, as well as measurements of the*

by breaking the Netscape implementation of SSL [8], or predicting Java session-ids [11].

Since a physical source of randomness is often too costly, most systems use a pseudo-random number generator. The state of the generator is seeded, and periodically refreshed, by entropy which is gathered from physical sources (such as from timing disk operations, or from a human interface). The state is updated using an algorithm which updates the state and outputs pseudo-random bits.

This paper studies the Linux pseudo-random number generator (which we denote as the LRNG). This is the most popular open source pseudo-random number generator, and it is embedded in all running Linux environments, which include desktops, servers, PDAs, smart phones, media centers, and even routers.

**Properties required of pseudo-random number generators.** A pseudo-random number generator must be secure against external and internal attacks. The attacker is assumed to know the code of the generator, and might have partial knowledge of the entropy used for refreshing the generator's state. We list here the most basic

IEEE S&P 2006

# Conclusions

- It is important to understand what **security guarantees** are expected from a cryptographic tool
- It is important to use **constructions** that have been proven secure or are widely believed to be secure
- The use of strong **randomness** is critical
- **Implementing** cryptographic constructions is hard!
  - bugs exist even in well-known and widely used cryptographic libraries
  - e.g., the Heartbleed Bug