CS 4910: Intro to Computer Security

Cryptographic Tools IV

Instructor: Xi Tan

What we already know

- Symmetric cryptography tools
  - Stream cipher
  - Block cipher
  - DES, AES
  - Block cipher modes
- Public Key Cryptography
  - RSA
  - Diffie-Hellman Key Exchange
- Message Integrity
  - MAC, Hash functions (MD5, SHA-1, SHA-2, SHA-3)

Today

Cryptographic tools

- Overview
- Symmetric Key Cryptography
- Public Key Cryptography
  - Diffie-Hellman Key Exchange
- Message Integrity and Digital Signatures
- Summary

Digital Signature

Digital Signatures

- A digital signature scheme is a method of signing messages stored in electronic form and verifying signatures
- Digital signatures can be used in very similar ways conventional signatures are used
  - paying by a credit card and signing the bill
  - signing a contract
  - signing a letter
- Unlike conventional signatures, we have that
  - digital signatures are not physically attached to messages
  - we cannot compare a digital signature to the original signature

Digital Signatures

- Digital signatures allows us to achieve the following security objectives:
  - authentication
  - integrity
  - non-repudiation

- Note that this is the main difference between signatures and MACs
  - a MAC cannot be associated with a unique sender since a symmetric shared key is used

Digital Signatures

- It is meaningful to consider the following attack models
  - key-only attack
  - known message attack
  - chosen message attack

- Adversarial goals might be
  - total break
  - selective forgery
  - existential forgery

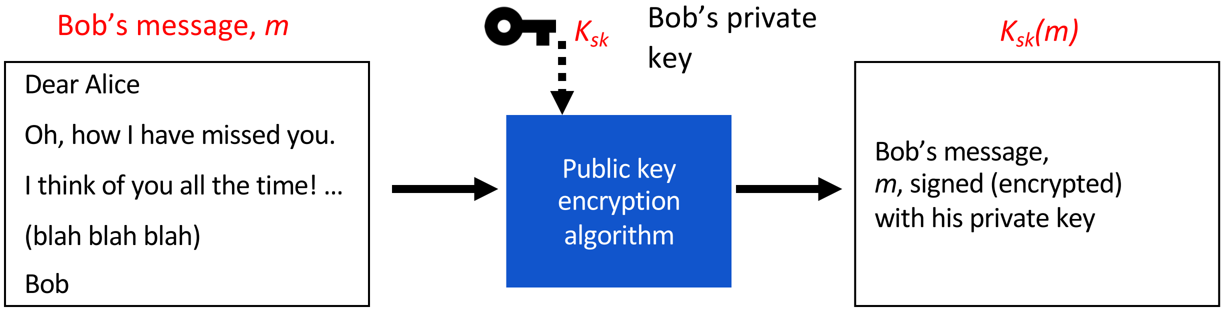
Digital Signatures

- A digital signature scheme consists of key generation, message signing, and signature verification algorithms
  - key generation creates a public-private key pair (*pk, sk*)
  - signing algorithm takes a messages and uses private signing key to output a signature
  - signature verification algorithm takes a message, a signature on it, and the signer's public key and outputs a yes/no answer

Digital Signatures

Simple digital signature for message *m*:

- Bob signs *m* by encrypting with his private key K_{sk}, creating “signed” message, $K_{sk}(m)$



Plain RSA Signatures

- Plain RSA signature is similar to plain RSA encryption
  - create a key pair as before: public $pk = (e, n)$ and private sk = d
  - signing of message m using sk is done as $\sigma = m^d \bmod n$
  - verification of signature σ on message m using pk is performed as $\sigma^e \bmod n \ ?= m$

Digital Signatures

- Plain RSA is not a secure signature scheme
  - both existential and selective forgeries are easy
  - the "hash-and-sign" paradigm is used in many constructions to achieve adequate security
    - e.g., compute *h(m)* and then continue with plain RSA signing of *h(m)*
  - this additionally improves efficiency
  - the hash function must satisfy all three security properties
    - preimage resistance
    - weak collision resistance
    - strong collision resistance

Digital Signatures

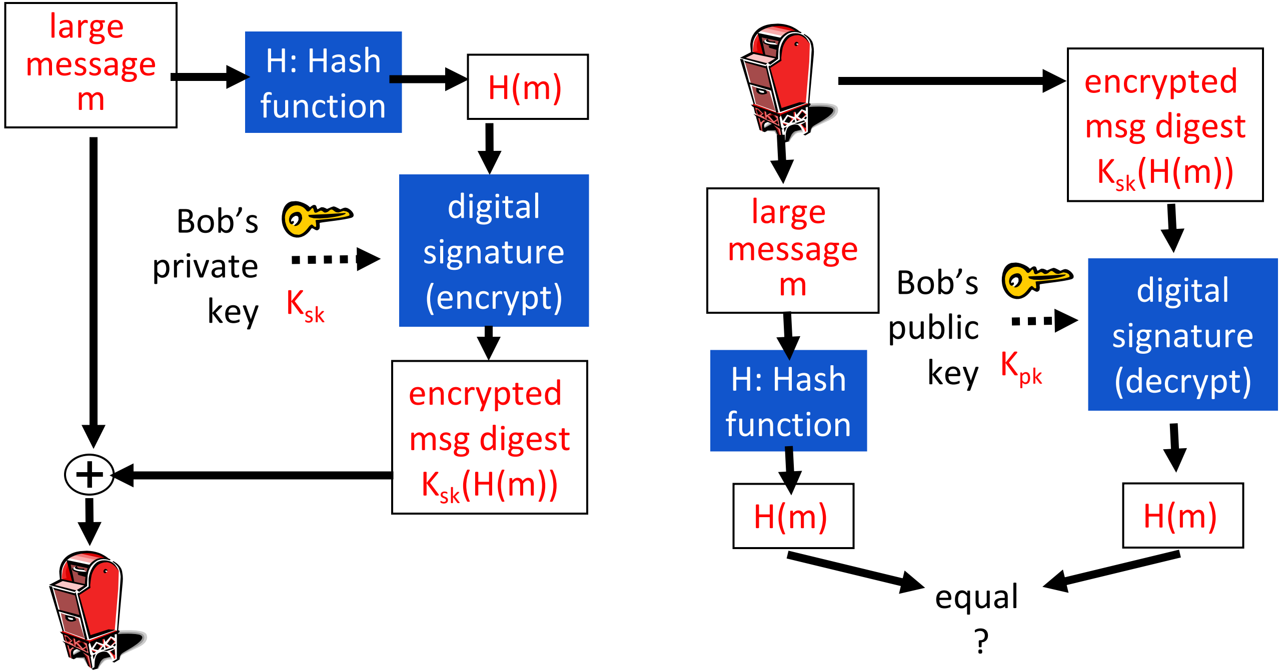
RSA signatures

- key generation
    - choose prime *p* and q, compute $n = pq$
    - choose prime *e* and compute *d* so that $ed \bmod (p - 1)(q - 1) = 1$
    - signing key is *d*, verification key is (e, n)
- message signing
    - given *m*, compute $h(m)$
    - output $\sigma = h(m)^d \bmod n$
- signature verification
    - given *m* and σ, first compute $h(m)$
    - check whether $\sigma^e \bmod n \mathrel{?=} h(m)$

Digital signature = signed message digest

Bob sends digitally signed message:

Alice verifies signature and integrity of digitally signed message:



Digital Signatures (more)

- Suppose Alice receives msg *m*, digital signature $K_{sk}(m)$
- Alice verifies *m* signed by Bob by applying Bob's public key K_{pk} to $K_{sk}(m)$ then checks $K_{pk}(K_{sk}(m)) = m$.
- If $K_{pk}(K_{sk}(m)) = m$, whoever signed *m* must have used Bob's private key.

Alice thus verifies that:

- ✓ Bob signed *m*.
- ✓ No one else signed *m*.
- ✓ Bob signed *m* and not *m'*.

Non-repudiation:

- ✓ Alice can take *m*, and signature $K_{sk}(m)$ to prove that Bob signed *m*.

Digital Signature Standard (DSS)

- Digital Signature Standard (DSS) or Digital Signature Algorithm (DSA) was adopted as a standard in 1994
  - its design was influenced by prior ElGamal and Schnorr signature schemes
  - it assumes the difficulty of the discrete logarithm problem
  - no formal security proof exists

Digital Signature Standard (DSS)

- DSS was published in 1994 as FIPS PUB 186
  - it was specified to hash the message using SHA-1 before signing
  - it was specified to produce a 320-bit signature on a 160-bit hash
- The current version is FIPS PUB 186-4 (2013)
  - DSA can now be used with a 1024-, 2048-, or 3072-bit modulus
  - the message size is 320, 448, or 512 bits
- Signing and signature verification involve:
  - hashing the message
  - computing a couple of modulo exponentiations on both longer and shorter sizes

Digital Signature Standard (DSS)

- Thorough evaluation of security of a signature scheme is crucial
  - often a message can be encrypted and decrypted once and long-term security for the key is not required
  - signatures can be used on legal documents and may need to be verified many years after signing
  - choose the key length to be secure against future computing speeds

Bit Security

- All constructions studied so far rely on the fact that an adversary is limited in computational power
  - if it has more resources than we anticipate, cryptographic algorithms can be broken
- Today, 112–128-bit security is considered sufficient
  - this means approximately that for 128-bit security, 2^{128} operations are needed to violate security with high probability
- This translates into the following parameters
  - symmetric key encryption: the key size is at least 112 bits
  - hash functions: the hash size is at least 224 bits
  - public key encryption: the modulus is at least 2048 bits long

Public key certificates

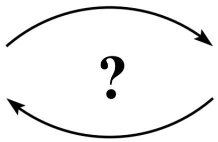
Public-key certificates

- 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 $Enc_{pk_B}(k)$ 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

- Motivation: Trudy plays pizza prank on Bob
  - Trudy creates e-mail order:
    Dear Pizza Store, Please deliver to me four pepperoni pizzas. Thank you, Bob
  - Trudy signs order with her private key
  - Trudy sends order to Pizza Store
  - Trudy sends to Pizza Store her public key, but says it's Bob's public key.
  - Pizza Store verifies signature; then delivers four pizzas to Bob.
  - Bob doesn't even like Pepperoni

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

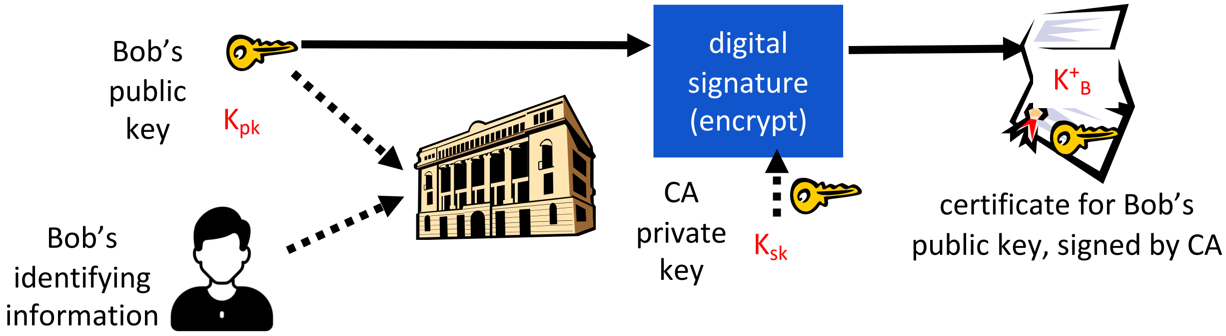
- Distribution of public keys can be done
  - by public announcement
    - a user distributes her key to recipients or broadcasts to community
  - through a publicly available directory
    - can obtain greater security by registering keys with a public directory

- 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

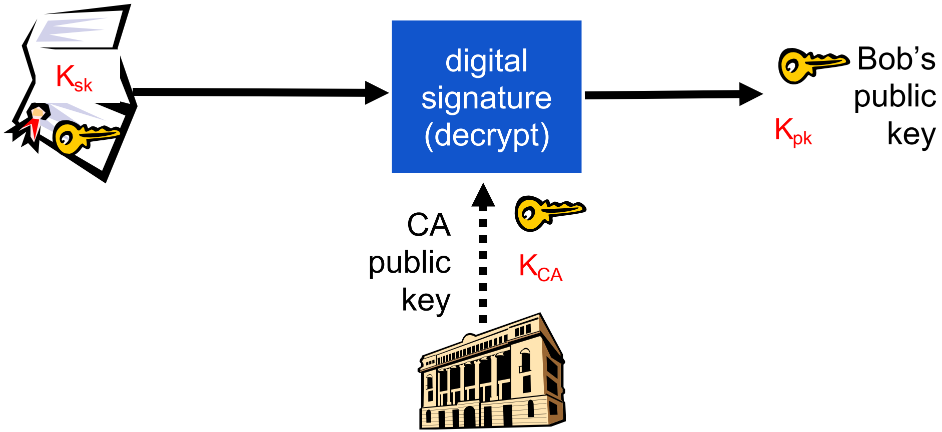
Certification Authorities

- Certification authority (CA): binds public key to particular entity, *E*.
- *E* (person, router) registers its public key with CA.
  - Bob provides "proof of identity" to CA.
  - CA creates certificate binding E to its public key.
  - certificate containing Bob's public key digitally signed by CA – CA says "this is Bob's public key"



Certification Authorities

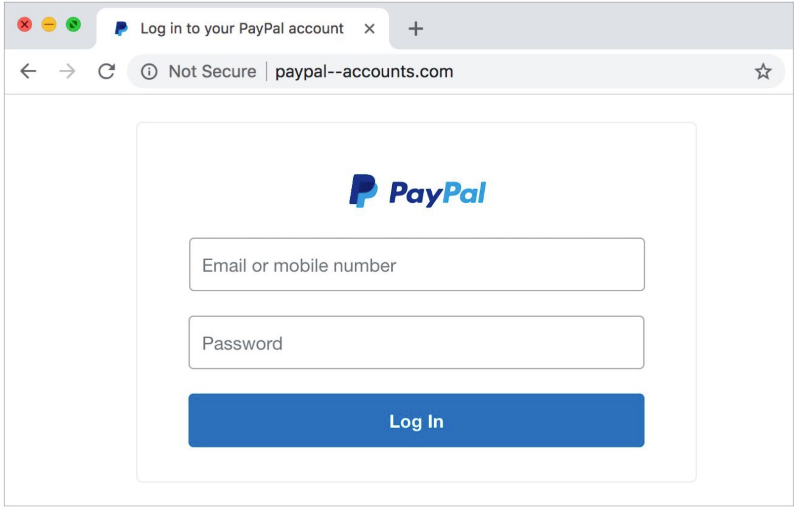
- When Alice wants Bob's public key:
  - gets Bob's certificate (Bob or elsewhere).
  - apply CA's public key to Bob's certificate, get Bob's public key

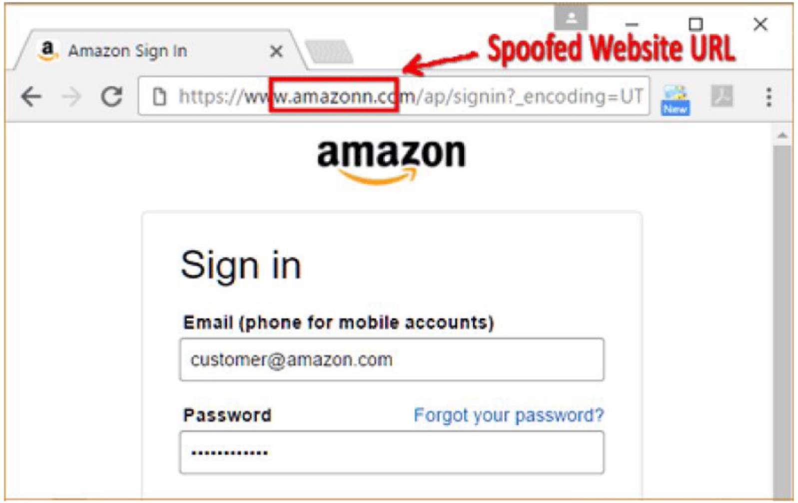


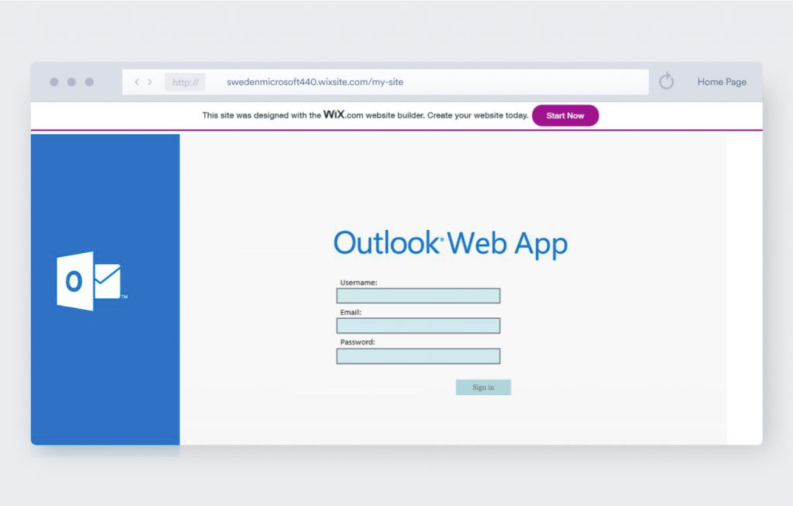
Public-key certificates

- (Root of trust) Assume there is a trusted central authority CA with a known public key pk_{CA}
- CA produces certificate for Bob as $cert_B = sig_{CA}(pk_B || Bob)$
- 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.

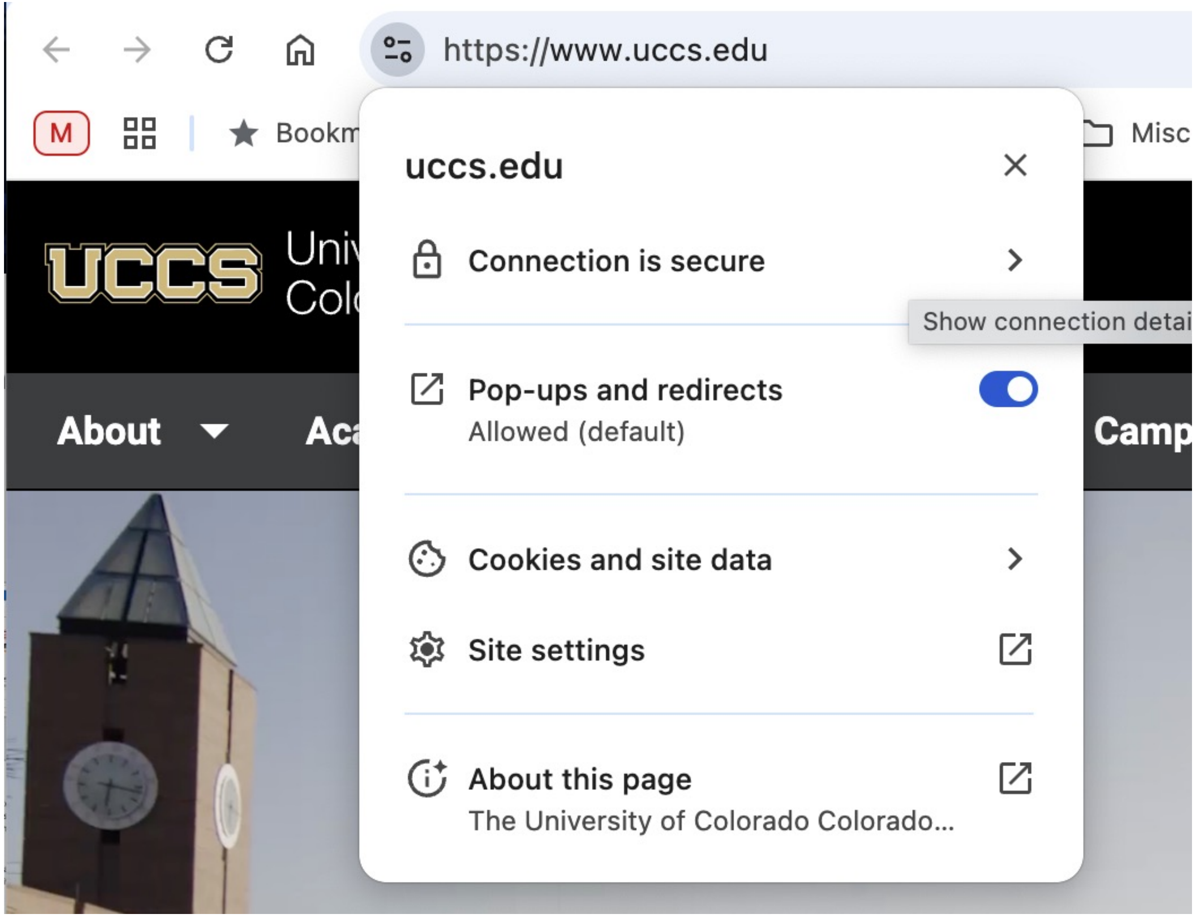
X.509 Identity Certificates

- 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

Web Identity



Web Identity

Certificate Viewer: uccs.edu

General | Details

Issued To

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

Issued By

Common Name (CN)	R10
Organization (O)	Let's Encrypt
Organizational Unit (OU)	<Not Part Of Certificate>

Validity Period

Issued On	Monday, February 3, 2025 at 1:14:59 PM
Expires On	Sunday, May 4, 2025 at 2:14:58 PM

SHA-256 Fingerprints

Certificate	dbd41dd17bb3996015c7cb48113a2a441fc39a1375fcbd07a0f65 5d9a3641cb0
Public Key	4be24b936561d7af029cec7051413108f24e9bee9bc4f59807aa 38efa1959e3a

Web Identity

Certificate Manager

- Local certificates
- Your certificates
- Chrome Root Store

Chrome Root Store

The Chrome Root Store contains certificates from Certificate Authorities trusted by the Chrome Root Program, and is continually reviewed on an ongoing basis. Learn more

Trusted Certificates

Export

Actalis Authentication Root CA	55926084EC963A64B96E2ABE01C...
Amazon Root CA 3	18CE6CFE7BF14E60B2E347B8DFE...
Amazon Root CA 2	1BA5B2AA8C65401A82960118F80...
Amazon Root CA 1	8ECDE6884F3D87B1125BA31AC3F...
Amazon Root CA 4	E35D28419ED02025CFA69038CD6...
Certum Trusted Network CA	5C58468D55F58E497E743982D2B...
Certum Trusted Network CA 2	B676F2EDDAE8775CD36CB0F63C...
Atos TrustedRoot 2011	F356BEA244B7A91EB35D53CA9AD...
Autoridad de Certificacion Firmaprofesional CIF A62634068	57DE0583EFD2B26E0361DA99DA9...

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, 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 $Enc_k(0)$, $Enc_k(1)$, . . ., 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 + 1th* bit with probability non-negligibly greater than 1/2

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 actual entropy collected by it, and a critical analysis of the use of the generator in Linux distributions on disk-less devices.

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 security requirements, using common terminology (e.g., of [3]). (A more detailed list of potential vulnerabilities appears in [14].)

IEEE S&P 2006

Conclusion

- 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

Summary

Summary (1 of 2)

- Three types of cryptography: secret-key, public key, and hash function

plaintext $\xrightarrow{K_S}$ ciphertext $\xrightarrow{K_S}$ plaintext

A) Secret key (symmetric) cryptography. SKC uses a single key for both encryption and decryption

plaintext $\xrightarrow{K_A}$ ciphertext $\xrightarrow{K_B}$ plaintext

B) Public key (asymmetric) cryptography. PKC uses two keys. One for encryption and the other for decryption.

plaintext $\xrightarrow{\text{hash function}}$ ciphertext

C) Hash function (one-way cryptography). The plaintext is not recoverable from the ciphertext

Summary (2 of 2)

- Application of the three cryptographic techniques for secure communication
  - Confidentiality
    - Encrypted message
  - End-Point Authentication (Both Alice and Bob)
    - Secure Key exchange: only Bob can decrypt session key
    - Digital signature: decrypting the digital signature with Alice's public key
      - Message was sent by Alice
  - Message Integrity
    - Hash value of her message