CSE 410/565 Computer Security
Spring 2022

Lecture 6: Public Key Certificates, Random Numbers

Department of Computer Science and Engineering
University at Buffalo

Cryptographic Topics Covered

- What we've discussed so far:
  - symmetric encryption
  - message authentication codes
  - hash functions
  - public-key encryption
  - digital signatures
- We finish with:
  - public key certificates for secure channel establishment
  - (pseudo)random numbers and generators

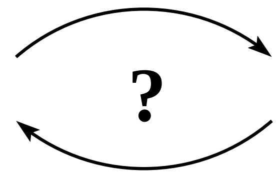
Secure Communication

- 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 $\mathsf{Enc}_{\mathsf{pk}_B}(k)$ 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

- Assume there is a central authority CA with a known public key pk_{CA}
- CA produces certificate for Bob as $\mathsf{cert}_B = \mathsf{sig}_{CA}(\mathsf{pk}_B || \mathsf{Bob})$
- Bob distributes $(\mathsf{pk}_B, \mathsf{cert}_B)$
- 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.

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, $\mathsf{PRG} : \{0,1\}^k \to \{0,1\}^{\ell(k)}$ 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 $\mathsf{Enc}_k(0), \mathsf{Enc}_k(1), \ldots$, 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

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