Password Authentication & Random Number Generation

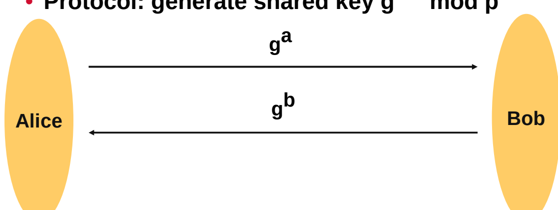
Dawn Song
dawnsong@cs.berkeley.edu

Review

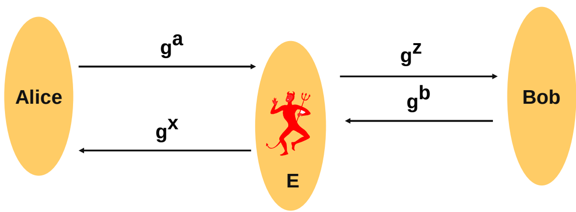
- PKI
- Authentication and Key Establishment Protocols
- Diffie-Hellman

Diffie-Hellman Key Agreement

- Public values: large prime p, generator g
- Alice picks secret random value a
- Bob picks secret random value b
- Protocol: generate shared key g^{ab} mod p



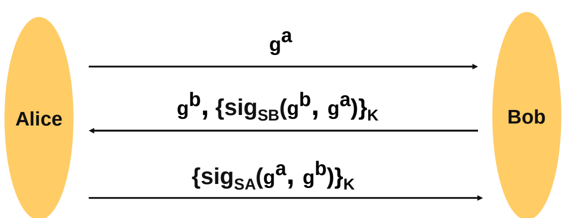
Man-in-the-middle Attack for DH Key Agreement



Attack:

- **A & B may believe they share a session key, but in fact, A & E share g^{ax} , B & E share g^{bz}**
- **How to fix it?**

Station-to-Station Protocol (STS)



- **sig_{SA} , sig_{SB} represent signatures of A & B**
- **Session key K = g^{ab}**

Kerberos Protocol

- Symmetric-key setting
  - Each user shares a symmetric key with key server
- A & S share K_{AS}, B & S share K_{BS}, L is the lifetime of the ticket, T_A is timestamp referring to A's clock, n_a is a nonce generated by A
- 1 A → S: A, B, n_a
- 2 S → A: $\{ K_{AB} , n_a, L, B \}_{K_{AS}}$, $\{ K_{AB} , A, L \}_{K_{BS}}$
- 3 A → B: $\{ K_{AB} , A, L \}_{K_{BS}}$, $\{ A, T_A \}_{K_{AB}}$
- 4 B → A: $\{ T_A \}_{K_{AB}}$
- **Encryption is not necessary for message 1**
- **Message 2 requires encryption, K_{AB} needs to remain secret**
- **Encryption in message 3 & 4 proves knowledge of K_{AB}**

Password-Based Authentication

- Setting
  - Alice and Bob know password P
  - They want to establish common key based on shared secret P
- Approach 1
  - K = H(P)
  - Use K to encrypt / authenticate communication
  - A → B: $\{\text{Message 1}\}_K$
  - B → A: $\{\text{Message 2}\}_K$
  - What's wrong with this approach?
- Goal: prevent eavesdropper from performing a dictionary attack to guess password

Simple Password Authentication

- Same setting as before
- Protocol
  - K = H(P)
  - Pick key K' at random
  - A → B: $\{ K' \}_K$
  - B → A: $\{ \text{"Terminal type: "} \}_{K'}$
- Dictionary attack possible?
  - Yes! Pick candidate password P
  - Compute K, decrypt K', and verify that message matches "Terminal type: "

EKE Basic Idea

- Observation: low entropy passwords enable dictionary attacks
- Countermeasures
  - Encrypt random values with password-based key
  - Public-key crypto establishes high-entropy session key
- Simple example
  - K = H(P), choose random key pair K_A, K_A^{-1}
  - A → B: $\{ K_A \}_K$
  - B → A: $\{ \{ K' \}_{K_A} \}_K$
  - Using K' as session key, dictionary attack possible?

EKE DH Protocol

- Large prime p, generator g
- K = H(P), A picks random a, B picks random b
- 1: A → B: $\{ g^a \}_K$
- 2: B → A: $\{ g^b \}_K$
- K' = H(g^{ab})
- Use K' as session key for secure communication
- Dictionary attacks?

Summary

- EKE is very nice and useful protocol
- Many variants exist: SPEKE, SRP, PDM, …
- Unfortunately, extensive patents on EKE and SPEKE prevented so far use of any of these protocols
  - Lucent owns EKE patent, demands exorbitant licensing fees
  - EKE patent: "Cryptographic protocol for secure communications", U.S. Patent #5,241,599, filed 2 October 1991, issued 31 August 1993.

Administravia

- Hw1 out
- Start looking for group partner

Random Number Generation

- Many crypto protocols require parties to generate random numbers
  - Key generation
  - Generating nonces
- How to generate random numbers?
  - Step 1: how to generate truly random bits?
  - Step 2: crypto methods to stretch a little bit of true randomness into a large stream of pseudorandom values that are indistinguishable from true random bits (PRNG)

Case Study

- Random number generation is easy to get wrong
- Can you spot the problems in this example?

```
unsigned char key[16];

srand(time(null));
 for (i=0; i<16; i++)
    key[i] = rand() & 0xFF;

where

static unsigned int next = 0;
void srand(unsigned int seed) {
   next = seed;
}

int rand(void) {
   next = next * 1103515245 + 12345;
   return next % 32768;
}
```


Real-world Examples

- X Windows "magic cookie" was generated using rand()
- Netscape browsers generated SSL session keys using time & process ID as seed (1995)
- Kerberos
  - First discover to be similarly flawed
  - 4 yrs later, discovered flaw with memset()
- PGP used return value from read() to seed its PRNG, rather than the contents of buffer
- On-line poker site used insecure PRNG to shuffle cards
- Debian Openssl package generates predictable pseudorandom numbers

Lessons Learned

- **Seeds must be unpredictable**
- **Algorithm for generating pseudorandom bits must be secure**