Authentication and Random Number Generation

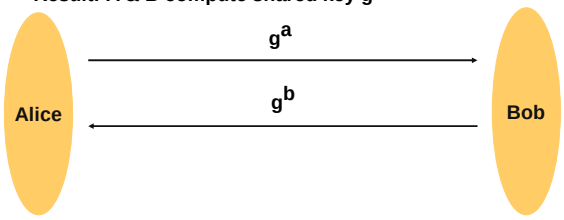
Dawn Song
dawnsong@cs.berkeley.edu

Review

- **Zero-knowledge proof**
  - **Challenge-response**
  - **Knowledge extractor**
- **Protocols for authentication and key agreement**
  - **Need careful design**

Diffie-Hellman Key Agreement

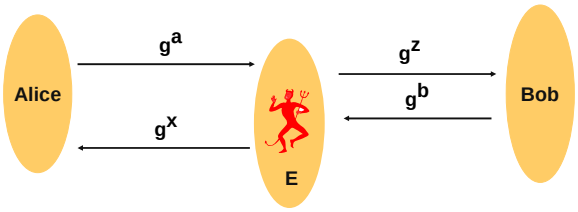
- **Public values: large prime p, generator g**
- **Alice picks secret random value a, sends g^a**
- **Bob picks secret random value b, sends g^b**
- **Result: A & B compute shared key g^{ab}**



Diffie-Hellman Assumption

- large prime p, generator g
- Computational Diffie-Hellman assumption: Given g^a , g^b , it is hard to compute g^{ab}
- Decisional Diffie-Hellman assumption: Given g^a , g^b , it is hard to distinguish between g^{ab} and a random element r (in mod p)
- Related to Discrete Log problem, but not known to be equivalent

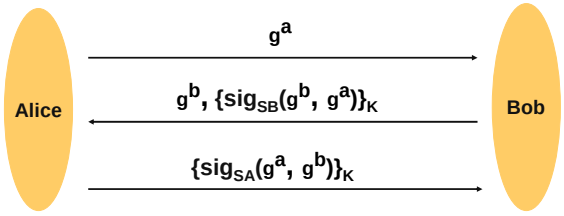
Man-in-the-middle Attack for DH Key Agreement



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

- Involves a 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: $\{ K_{AB}, A, L \}_{K_{BS}}$, $\{ A, T_A$

Administrative Matter

- Office hour this week moves to tomorrow 4pm
- svn ready
- Project description out

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

Lessons Learned

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

Generating Pseudorandom Numbers

- **True random number generator (TRNG)**
  - **Generates bits that are distributed uniformly at random, so that all outputs are equally likely, with no patterns, correlations, etc.**
- **Cryptographically secure pseudorandom number generator (CS-PRNG)**
  - **Taking a short true-random seed, and generates long sequence of bits that is computationally indistinguishable from true random bits**

CS-PRNG

- **CS-PRNG: cryptographically secure pseudorandom number generator**
  - **G: maps a seed to an output G(S)**
    - **E.g., G: $\{0,1\}^{128}$ -> $\{0,1\}^{1000000}$**
  - **Let K denote a random variable distributed uniformly at random in domain of G**
  - **Let U denote a random variable distributed uniformly at random in range of G**
  - **G is secure if output G(K) is computationally indistinguishable from U**

TRNG (I)

- **TRNG should be random and unpredictable**
- **Bad choices**
  - **IP addresses**
  - **Contents of network packets**
  - **Process IDs**
- **Some sources of randomness**
  - **High-speed clock**
  - **Soundcard**
  - **Keyboard input**
  - **Disk timings**

TRNG (II)

- **How to convert non-uniform sources of randomness into TRNG?**
  - **Use a cryptographic hash function, such as SHA1**
  - **Suppose x is a value from an imperfect source, or a concatenation of values from multiple sources, and it is impossible for an attacker to predict the exact value x except with probability $1/2^n$**
  - **Then hash(x) truncated to n bits should provide a n-bit value that is uniformly distributed**

Conclusion

- **Authentication & key-agreement**
  - **If not well designed, attacker can impersonate, learn session key, etc.**
  - **Diffie-Hellman key agreement**
  - **Kerberos key agreement**
  - **Need to be able to detect attacks in flawed protocols**
- **Random number generation**
  - **Common mistakes**
  - **TRNG & CS-PRNG**