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 $g^{ab} \mod p$

\[ \begin{align*}
  &g^a \\
  &\quad g^b \\
\end{align*} \]

Alice \hspace{2cm} Bob
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)

$g^a$

$g^b$, $(\text{sig}_{SB}(g^b, g^a))_K$

$(\text{sig}_{SA}(g^a, g^b))_K$

• $\text{sig}_{SA}$, $\text{sig}_{SB}$ 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 $\rightarrow$ S: A, B, $n_A$
  - 2 S $\rightarrow$ A: $(K_{AB}, n_A, L, \text{sig}_{AS}(K_{AB}, A, L))_{KB}$
  - 3 A $\rightarrow$ B: $(K_{AB}, A, L)_{KB}$, $(A, T_A)_{KB}$
  - 4 B $\rightarrow$ A: $(T_A)_{KB}$
• Encryption is not necessary for message 1
• Message 2 requires encryption, $K_{AS}$ needs to remain secret
• Encryption in message 3 & 4 proves knowledge of $K_{AS}$
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
  - $A \rightarrow B: \{\text{Message 1}\}_K$
  - $B \rightarrow A: \{\text{Message 2}\}_K$
  - 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
  - $A \rightarrow B: \{K'\}_K$
  - $B \rightarrow A: \{\text{“Terminal type: ”}\}_K$

- **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 $K_a, K_a^{-1}$
  - $A \rightarrow B: \{K_a\}_K$
  - $B \rightarrow A: \{\{K'\}_a\}_K$
  - 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: \( A \rightarrow B: \{ g^a \}_K \)
- 2: \( B \rightarrow A: \{ g^b \}_K \)
- \( K' = H( g^{ab} ) \)
- 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

**Administrivia**

- 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?

```c
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