Overview

Today:
- HW 7 quiz
- Public Key Algorithms - ideas, math, and RSA

Next:
- Spring Break! Have fun!
- If you want to be productive:
  - Work on project phase 3
  - Read Sections 2.4-2.6, 2.8, 10.1, 10.2

Recall Basic Idea

- Keys the same or different?
  - Different: “Public-key crypto”

- Some algorithms: RSA, ElGamal, ECC
- Best feature: Simpler key management (can send to a stranger)
- Worst feature: Slow (1-2 Mbps for 2048-bit RSA; others are a little faster...)

Pay with 1234 5678 9012 3456
Public Key Crypto
Where do the keys come from?

Mathematical/Computational Properties
- \( KPG(R) \rightarrow (PU, PR) \) is efficiently computable (polynomial time)
- For all messages \( M \), \( D(PR, E(PU, M)) = M \) (decryption works)
- Computing \( PR \) from \( PU \) is computationally infeasible (we hope!)

Generally: \( PR \) has some "additional information" that makes some function of \( PU \) easy to compute (which is hard without that info) - this is the "trapdoor secret"

How can this be possible?

To get a sense of how trapdoor secrets help:

Problem: How many numbers \( x \in \{1, n-1\} \) have \( \gcd(x, N) > 1 \) for \( N = 32,501,477 \)?
(or: how many have a non-trivial common factor with \( N \)?)

How could you figure this out?
How long would it take to compute?
What if \( N \) were 600 digits instead of 8 digits?

What if I told you the prime factorization of \( N \) is 5,407 * 6,011?
How can this be possible?

To get a sense of how trapdoor secrets help:

**Problem:** How many numbers $x \in \{1, n-1\}$ have gcd$(x, N) > 1$ for $N = 32,501,477$?

(or: how many have a non-trivial common factor with $N$?)

How could you figure this out?

How long would it take to compute?

What if $N$ were 600 digits instead of 8 digits?

What if I told you the prime factorization of $N$ is $5,407 \cdot 6,011$?

- 5,406 multiples of 6,011 share the factor 6,011 with $N$
- 6,010 multiples of 5,407 share the factor 5,407 with $N$
- No numbers in common between these two sets (prime numbers!)

So... $5,406 + 6,010 = 11,416$ numbers share a factor with 32,501,477

The factorization of $N$ is a “trapdoor” that allows you to compute some functions of $N$ faster

Using Public Key Crypto in the JCA

Generating a keypair:

```java
public static KeyPair genRSAKey(int bits) {
    KeyPair kp = null;
    try {
        RSAPrivateKeySpec privSpec = new RSAPrivateKeySpec(bits, RSAKeyGenerator.getValidSpecSet(RSA));
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(privSpec);
        kp = kpg.genKeyPair();
    } catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException e) {
        System.err.println("Does - basic RSA key generation failed (?)");
    }
    return kp;
}
```

kp.getPublic() gives PublicKey (IS-A Key, so can be used to initialize a Cipher in ENCRYPT_MODE)
kp.getPrivate() gives PrivateKey (IS-A Key, so can be used to initialize a Cipher in DECRYPT_MODE)

Otherwise works just like Cipher with a symmetric cipher algorithm!

Related Notion - Key Agreement

Original idea - before public key encryption

**Alice**
Randomness (R)

KeyPair Generator (KPG)

PubKey (PU_A) PrivKey (PR_A)

Generate Secret (GS)

Secret (S)

**Bob**
Randomness (R)

KeyPair Generator (KPG)

PubKey (PU_B) PrivKey (PR_B)

Generate Secret (GS)

Secret (S)
**Related Notion - Key Agreement**

Original idea - before public key encryption

<table>
<thead>
<tr>
<th>Alice</th>
<th>Randomness (R)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KeyPair Generator (KPG)</td>
</tr>
<tr>
<td></td>
<td>PubKey (Pu_A), PrivKey (Pr_A)</td>
</tr>
<tr>
<td></td>
<td>Generate Secret (GS)</td>
</tr>
<tr>
<td>Secret (S)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bob</th>
<th>Randomness (R)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KeyPair Generator (KPG)</td>
</tr>
<tr>
<td></td>
<td>PubKey (Pu_B), PrivKey (Pr_B)</td>
</tr>
<tr>
<td></td>
<td>Generate Secret (GS)</td>
</tr>
<tr>
<td>Secret (S)</td>
<td></td>
</tr>
</tbody>
</table>

Properties:
- PK KeyPair properties
- GS(Pu_A, Pr_B) = GS(Pu_B, Pr_A)
- Infeasible to compute S from only public keys (Pu_A and Pu_B)

Note: No message - so can send no information.

But two sides get a shared secret.

---

**Using Key Agreement in the JCA**

For algorithm “DH” (Diffie-Hellman)

Generating a keypair requires first generating public parameters:

```java
AlgorithmParameterGenerator params = AlgorithmParameterGenerator.getInstance("DH");
params.init(new HashMap());
AlgorithmParameters params = params.generateParameterSpec();
AlgorithmParameterSpec spec = params.getParameterSpec(AlgorithmParameterSpec.class);
```

But: Parameter generation can be slow so often done in advance and saved.

Weakness recently found for this, however... be cautious!

Given parameters, Alice and Bob can do key agreement:

```java
KeyPairGenerator bkg = KeyPairGenerator.getInstance("DH");
bkg.initialize(params);
KeyPair aliceKP = bkg.generateKeyPair();
KeyAgreement aliceKA = KeyAgreement.getInstance("DH");
aliceKA.init(aliceKP.getPrivate());
byte[] aliceS = aliceKA.generateSecret();
```

Bob’s public key (received)

---

**Key Sizes for Public Key Systems**

From NIST publication 800-57a

**Issue:** PK algorithms based on mathematical relationships, and can be broken with algorithms that are faster than brute force.

We spent time getting a feel for how big symmetric cipher keys needed to be

How big do keys in a public key system need to be?

---

**Table 1: Comparing strengths**

<table>
<thead>
<tr>
<th>Security</th>
<th>Deterministic key algorithms</th>
<th>ECC (e.g., DSA-DH)</th>
<th>ECC (e.g., RSA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>1.223</td>
<td>1.284</td>
<td>1.252</td>
</tr>
<tr>
<td>768</td>
<td>1.300</td>
<td>1.389</td>
<td>1.357</td>
</tr>
<tr>
<td>512</td>
<td>1.369</td>
<td>1.476</td>
<td>1.442</td>
</tr>
<tr>
<td>256</td>
<td>1.438</td>
<td>1.568</td>
<td>1.530</td>
</tr>
</tbody>
</table>

From NIST pub 800-57a.
Weakness in long-term fixed DH parameters

From 2015 ACM Conference on Computer and Communication Security:

Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice


*IDA Pen Test Team — MRA Research (CNG) CNRS and University of Rennes

ABSTRACT

We investigate the security of Diffie-Hellman key exchange as used in Transport Layer Security (TLS) — the most common protocol for securing internet communications — and find it to be vulnerable to weak keys.

To test our Internet-scale measurements, we implemented a number of key exchange plugins for the Firefox browser that allow us to measure Diffie-Hellman key exchanges. To test our hypothesis, we measured key exchanges from tens of thousands of clients and servers. We then analyzed key exchanges using a standard technique to determine which parameters were used.

We found that the real-world deployments of TLS are vulnerable to weak keys. In particular, we found that keys used in over 20% of all key exchanges were vulnerable to weak keys. We also found that keys used in over 10% of all key exchanges were vulnerable to weak keys.

We call for the adoption of the Diffie-Hellman or RSA encryption methods as a means of mitigating this vulnerability.

We also call for the adoption of the Diffie-Hellman or RSA encryption methods as a means of mitigating this vulnerability.