

Password User Authentication

- 1) **What NOT to do or the basics of password attacks.**
- 2) **Cryptographic hash functions and salt**
- 3) **asymmetric key cryptography (aka Public/Private key crypto)**
- 4) **What to do and not do selecting passwords, locking out and resetting passwords – social engineering attacks**
- 5) **Crackers, password database dumps, key loggers. (exercises)**
- 6) *Is there any place for passwords in a well designed system?*

User Authentication

.... using a password

Why the password and not biometrics or hardware keys...?

- Simple and elegant, on the surface at least.
- Most important merely because it is the most widely used authentication method.
- Storing or transmitting passwords although apparently simple can be problematic.
- Depends on cryptography ... we need to know a little about this but we leave most of the details for someone else.
- It seems simple – we will follow the approach of the book and see how many places there are to stumble. It is partly a historical journey but illustrates the difficulties of creating and maintaining secure procedures.

Underlying Premise

- The user and the authenticating agent share a secret – the password.

Basic Procedure

- The user supplies:
 - a key (the *username*) and
 - the shared secret (the *password*)
- The authenticating agent:
 - uses the *username* key to look up its copy of the *password* from its database
 - if the password matches - the user is authenticated

Problem 1

1.1 We clearly cannot store the secret in plain text unless you want to be a target.



1.2 It is pointless to encrypt it. If we need a key to decrypt it then the password storage problem becomes a key storage problem.

Solution 1

We use a **cryptographic hash function** (aka *message digest function* – think MD5) to compute a hash of the password.

The function takes input data (often an entire message), and outputs a short, fixed length hash, and does so as a one-way function. The authenticating agent stores $f_{\text{hash}}(\textit{password})$ in its database instead of passwords. It is not possible to compute the password from the hash, so they can be stored without encryption. In Unix systems they were easily accessible by all users for many years.

For good cryptographic hash functions, collisions (two plaintexts which produce the same hash) are extremely rare. Hence a comparison of a hashed password can be safely used for authentication.

- The user supplies:
 - a key (the *username*) and
 - the shared secret (the *password*)

- The authenticating agent:
 - computes $f_{\text{hash}}(\textit{password})$ and
 - uses the key to look up its stored copy of the $f_{\text{hash}}(\textit{password})$ from its database
 - if they match then the user is authenticated

Problem 2

The original cryptographic hash function *crypt()* was computationally expensive – taking at least a second on a fast machine of the day. How big a problem is this?

Solution 2

A simple solution is to feed the output of the hash function as input a number of times to increase the complexity or use a better hash function.

Problem 3

 A bit of code from J. Viega:

```
char *get_new_crypted_password() {
    char *pw1,*pw2,*crypted_pw;
    while (1) {
        pw1 = read_line("Enter password: ");
        pw2 = read_line("reenter to confirm: ");
        if (strcmp(pw1,pw2)) {
            fprintf(stdout,"Passwords differ!\n");
            fprintf(stdout,"Try again.\n");
            free(pw1);
            free(pw2);
            continue;
        }
        break;
    }
    crypted_pw = crypt(pw1,pw1);
    free(pw1);
    free(pw2);
    return crypted_pw;
}
```

Solution 3

```
char *get_new_crypted_password() {
    char *pw1,*pw2,*crypted_pw1,*crypted_pw2,*salt;
    salt = get_random_salt();
    while (1) {
        pw1 = read_line("Enter password: ",0); /* turn off echo */
        crypted_pw1 = crypt(pw1,salt);
        while (*pw1) *pw1++ = 0;          /* the clear text password from memory */
        pw2 = read_line("reenter to confirm: ",0);
        crypted_pw2 = crypt(pw2,salt);
        while (*pw2) *pw1++ = 0;          /* the clear text password from memory */
        if (strcmp(crypted_pw1,crypted_pw2)) {
            fprintf(stdout,"Passwords were not the same! Try again.\n");
            continue;
        }
        break;
    }
    free(crypted_pw2);
    free(salt);
    return crypted_pw1;
}
```

Turn off echo

Wipe the memory holding the unencrypted password as quickly as possible to decrease the window of opportunity of memory scanners. Less of a problem with 'modern' processors and operating systems.

Prevent critical sections of memory from being swapped from virtual memory to the paging files where nefarious and cunning evil doers will take advantage of opportunity. In Linux *mlock()* will do the trick, in Windows® VirtualAlloc with MEM_PHYSICAL flag can be used.

Problem 4

```
char *get_password( char *salt) {

    char *crypted_pw,*c;

    char *password = read_line(("Enter password: ",0);

    if ( !salt ) { /* If user was not found then salt == 0 */
        /* zero out the memory used for the password */
        while ( *password) *password++ = 0 ;
        return password;
    }

    c = crypt(password,salt);
    crypted_pw = (char *)malloc(strlen(c)*sizeof(char));
    strcpy(crypted_pw,c);

    /* zero out the memory used for the password */
    while ( *password) *password++ = 0 ;
    return crypted_pw;
}
```

```
int main(){

    char *user_name;
    user_name = get_user_name();

    if (username) salt = get_salt();
    else salt=0;

    if (get_password(salt) ==get_stored_password(user_name))
        fprintf(stderr,"Authentication successfull");
    else
        fprintf(stderr,"Authentication failed!");
}
```

Note the code in the book does not indicate directly that it is the username or the password is incorrect. ✓

However, it gave the game away by not hashing the password for invalid usernames. It will return quicker for invalid users – might as well print INVALID USER ID.

It should zero out encrypted passwords before freeing the memory.

Maybe failed logins attempts occur faster than a human can type?

Problem 5

What do we do when we detect a large number of failed logins?

Challenge Response

Challenge-response authentication is a family of protocols in which one party presents a question ("challenge") and another party must provide a valid answer ("response") to be authenticated. A simple version may hash the challenge to provide a response, using a shared secret as the salt for the hash function. The shared secret can also be modified after authentication by some mutually agreed algorithm to ensure eavesdroppers will not be able to replay the authentication-response conversation.

Simple Example mutual authentication sequence

Server sends a unique challenge value **sc** to the client
Client generates unique challenge value **cc**
Client computes $cr = \text{hash}(cc + sc + \text{secret})$
Client sends **cr** and **cc** to the server
Server calculates the expected value of **cr** and ensures the client responded correctly
Server computes $sr = \text{hash}(sc + cc + \text{secret})$
Server sends **sr**
Client calculates the expected value of **sr** and ensures the server responded correctly

Can a session like this be hijacked by a man in the middle?

How do you set up the shared secret initially without a secure communication channel?

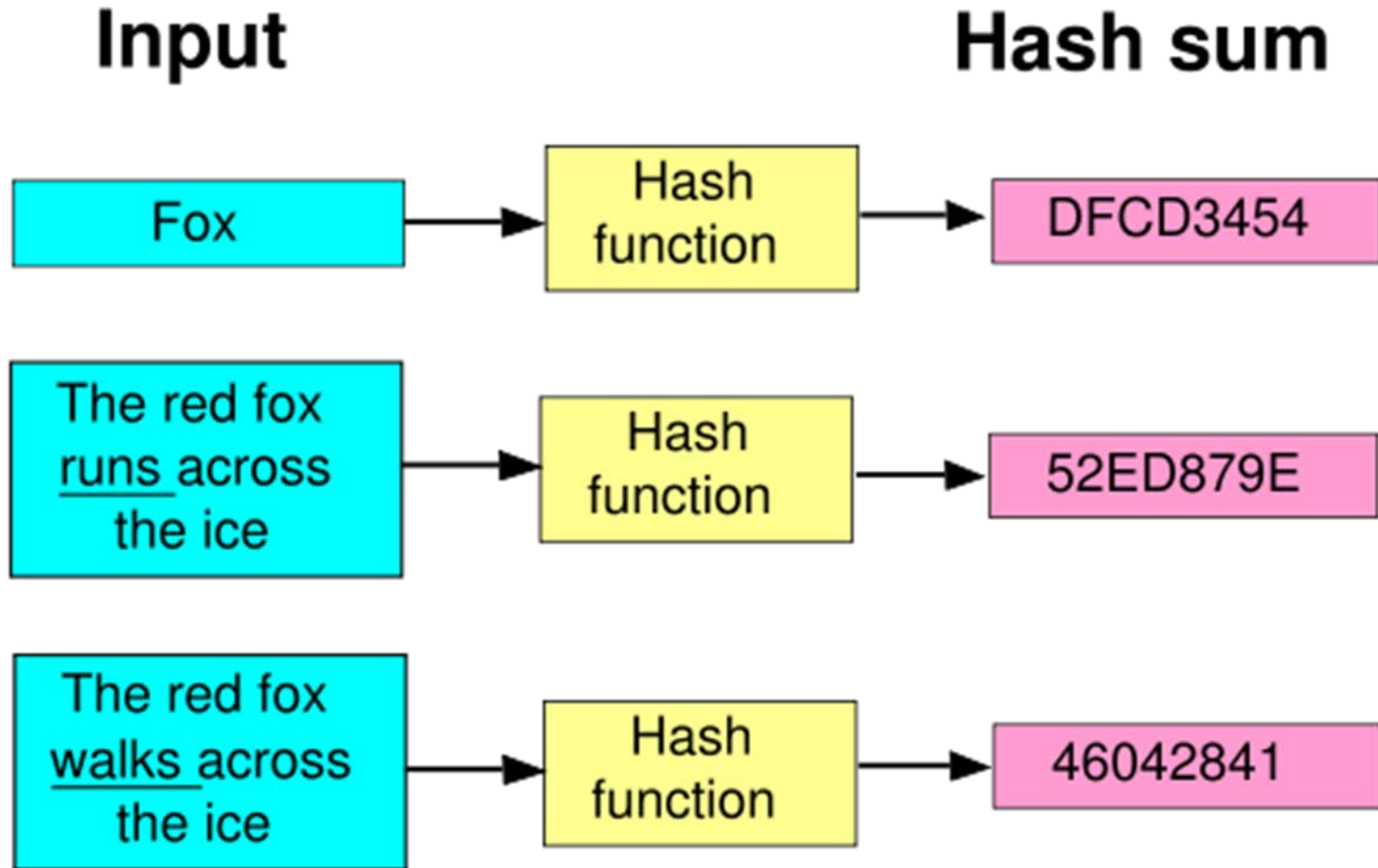
where

sc is the server generated challenge
cc is the client generated challenge
cr is the client response
sr is the server response

- We need to store user passwords so they can log back in later, *however*:
- Don't store passwords in plain text
 - They can be easily read and that's bad!
- Don't (just) store encrypted passwords
 - This makes the passwd storage problem a key hiding problem.

- Store cryptographic hashes of passwords
- Use a salt when hashing
- Make the hash run slow
- Run the hash several times (hash the hash of the hash)
- Encourage your users not to pick guessable passwords

Cryptographic Hash Functions



Cryptographic Hash Functions

Have 3 main properties

- It is very hard if not impossible to get from the hash to the input that produced the hash.
- Should not be fast to compute.
- Collisions do not occur, even when two sets of input are closely related.

Salt

A salt is a string of (random) bits that we append to the password before hashing.

Each bit doubles the search space.

Precomputed dictionaries must be recomputed, which can be very costly.

Salt – an example

User is known to use one of 200.000 words as his password.

System uses a 32-bit salt.

Attacker must precompute hashes for all passwords with all 4.294.967.296 possible salts.

Total:

$$2^{32} \times 200000 \text{ hashes} = 8.58993459 \times 10^{14}$$

Around 800 trillion combinations

Salt – storing

We can depend on a static salt and try to hide it somewhere in the code.

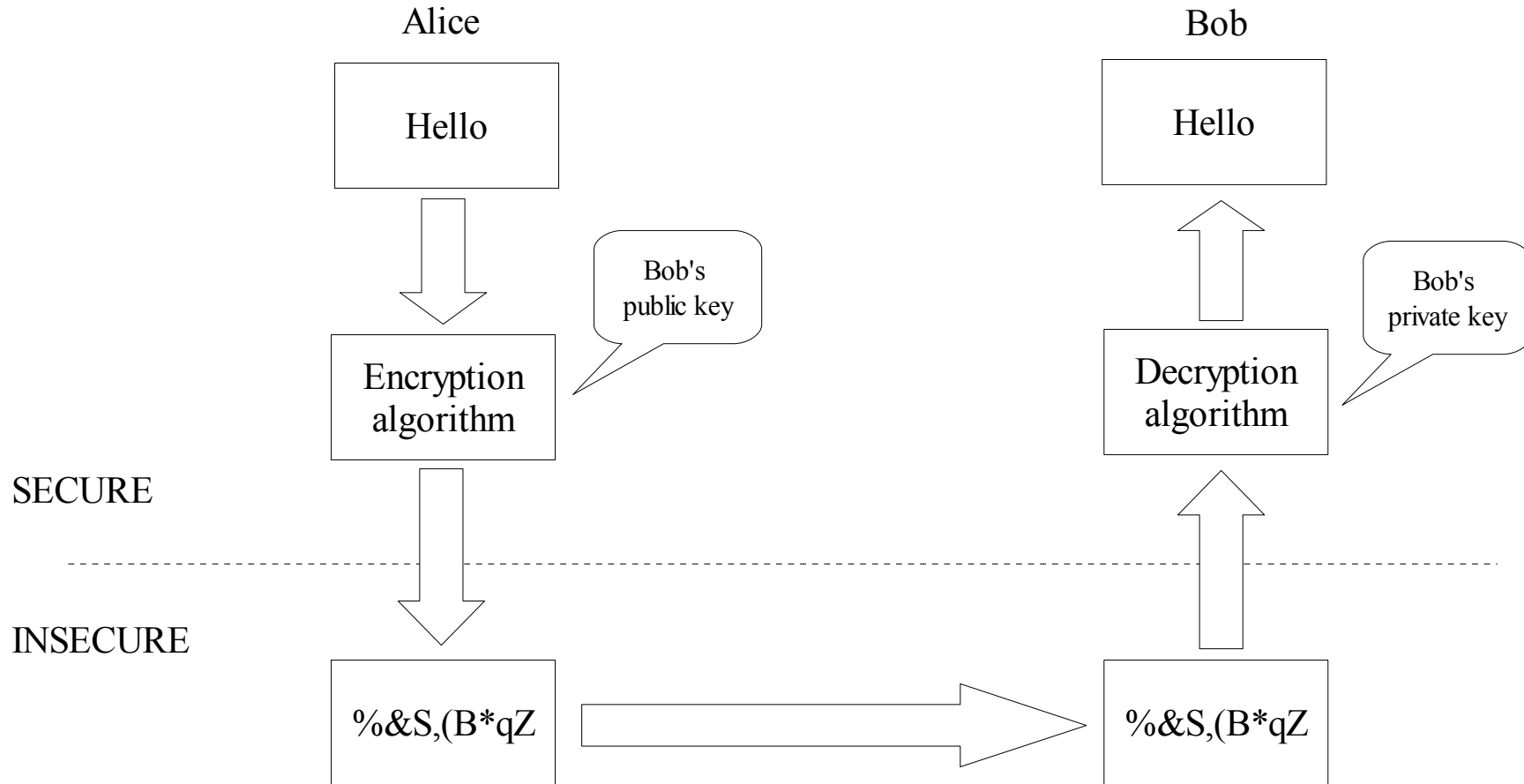
Or we can generate a new, random salt for each new user and store the salt in the clear along with usernames and password hashes

LM Hash

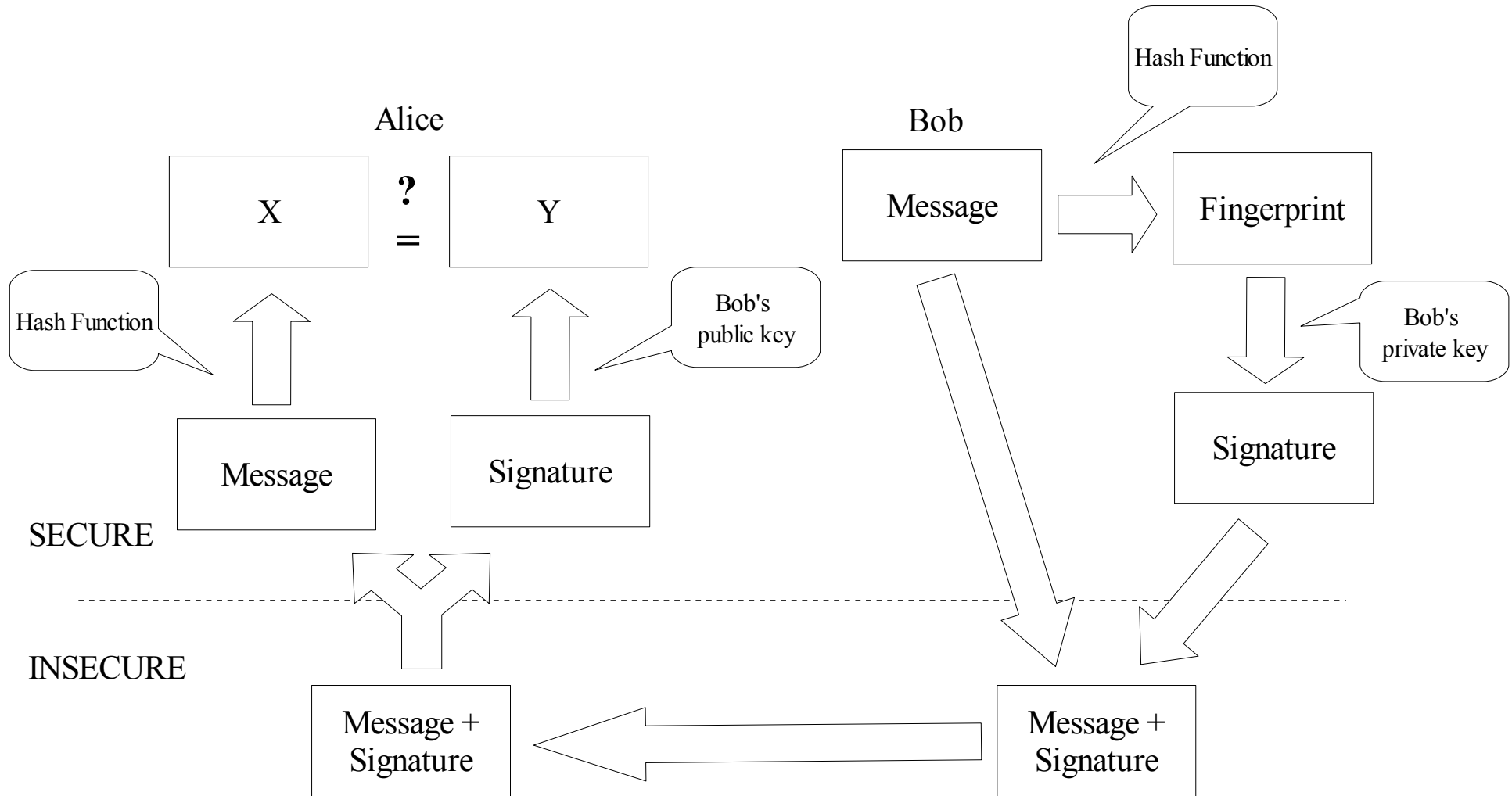
Example of bad hash algorithm is LMHash for Windows

- # The user's password is converted to uppercase.
- # This password is either null-padded or truncated to 14 bytes.
- # The "fixed-length" password is split into two 7-byte halves.
- # These values are used to create two DES keys, one from each 7-byte half, by converting the seven bytes into a bit stream, and inserting a zero bit after every seven bits. This generates the 64 bits needed for the DES key.
- # Each of these keys is used to DES-encrypt the constant ASCII string "KGS!@#\$%", resulting in two 8-byte ciphertext values.
- # These two ciphertext values are concatenated to form a 16-byte value, which is the LM hash.

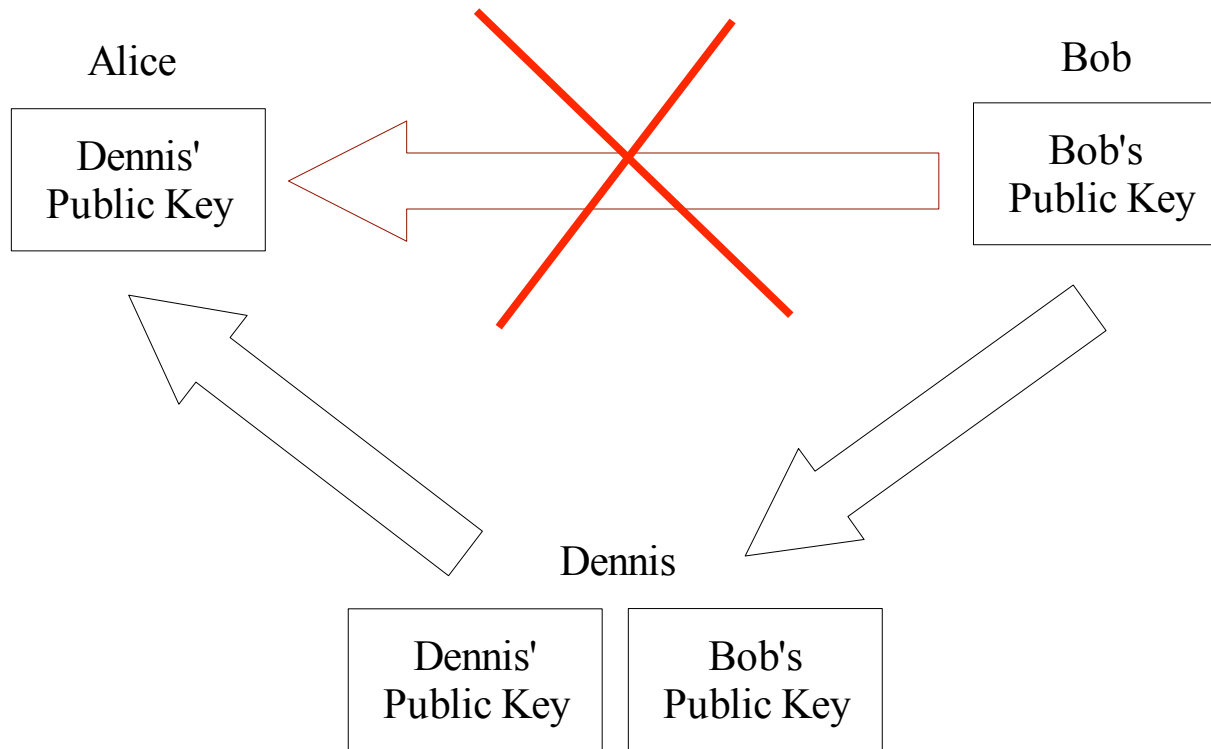
Public Key Cryptography



Digital Signature



Man In The Middle



Digital Certificate

Attributes:

- The public key being signed
- A name, which can refer to a person, a computer or organization
- A expiration date
- The location (URL) of a revocation center
- etc.

The certificate is signed by a trusted third party, i.e. it is encrypted with the private key of the trusted third party.

Trusted Third Party

More than 50 certificates are trusted by the most popular web browsers.

A web site obtains a certificate by applying to a certificate provider. This public certificate is served to any web browser that connects to the web site.

If the obtained certificate is signed by one of the owners of the certificates your browser already trusts, your browser will also trust the web site.

Guidelines

- Protocols
- Password Resets
- Password Choice
- Other

Michael Howard, David LeBlanc and John Viega:
'19 Deadly sins of Software Security'

Guidelines for Choosing Protocols

- Use strong password protocol
- Establish an SSL/TLS connection
- Otherwise - cryptography

Guidelines for Password Resets

- Be careful with locking out users in case of too many bad password attempts (DoS)
- Password reset questions
- The Paris Hilton Hijacking
- E-mail – temporary, new, random password better than human involved.
- Customer support should not be allowed to give password over the phone – or mail

Guidelines for Password Choice

- Usability >< Security
- Give ideas of how to choose passwords.
- Check for weak password when it is chosen - CrackLib
- If change of password is mandatory – keep old password in database

Other guidelines

- Only one single message for failed log in
- Log failed password attempts – not the failed passwords
- Do not store plaintext passwords
- Make users change their account passwords first time they use the application
- Use strong salted cryptographic one-way function based on hash for password storage
- No echo when users type password
- Use one-time password for access from untrustworthy systems

Password User Authentication

- 1) **What NOT to do or the basics of password attacks.**
- 2) **Cryptographic hash functions and salt**
- 3) **asymmetric key cryptography (aka Public/Private key crypto)**
- 4) **What to do and not do selecting passwords, locking out and resetting passwords – social engineering attacks**
- 5) **Crackers, password database dumps, key loggers. (exercises)**
- 6) *Is there any place for passwords in a well designed system?*
- 7) *Phishing, eavesdropping (wireline,airwaves,eyes,cameras),browser single sign-on, securing the comms channel with SSL, can you think of any other related issues.*

Resources:

- Building Secure Software – *John Viega and Gary McGraw*, Addison Wesley 2002.
- 19 Deadly Sins of Software Security – *Michael Howard, David LeBlanc and John Viega*, McGraw Hill Osborne 2005.
- Innocent Code – *Sverre Huseby*, Wiley 2004.
- Code Hacking – *Richard Conway and Julian Cordingley*, Charles River Media 2004.
- Cryptography and Network Network Security – *William Stallings*, Pearson 2006.
- Protecting RAM Secrets with Address Windowing Extensions – *Jason Coombs*, Dr. Dobb's Journal, October 18, 2004.
- New Directions in Cryptography - *Whitfield Diffie and Martin E. Hellman*, IEEE International Symposium on Information Theory in Ronneby, Sweden, June 21–24, 1976
- Address Windowing Extensions and Microsoft Windows 2000 Datacenter Server - <http://msdn.microsoft.com/en-us/library/ms810461.aspx>, March 30, 1999
- Password Authentication with Insecure Communication- *Leslie Lamport*, SRI International, Communications of the ACM November 1981, Volume 24, Number 11.
- Wikipedia – password authentication, challenge response protocol, cryptographic hash, asymmetric cryptography.