CIS2326 Week 8 - Security

This handout examines various ways of improving security - attached is a tutorial problem.

**Mobile Code Threats** Infected 'Mobile code' can be input to a machine and the recipient may try to run it. Examples of this mobile code are java applets, 'agents' (which check information and postscript. One method of dealing with the code is via *Sandboxing*. This is an attempt to confine each piece of mobile code (an applet for example) to a limited range of virtual addresses enforced at run-time. Each sand-box shares the property that all of its addresses share some string of higher-order bits. Each applet is given two sandboxes, one for the code and one for the data (see diagram - part (a) from lecture). The code in part (b) checks that an instruction does not jump to an instruction or reference code outside its sandbox. An example of the kind of code checked is a ‘JMP’ instruction.

Other ways of checking applets are (i) to run them interpretively and not let them access the hardware (the approach taken by web browsers) (ii) only accept applets from trusted sources.

**Security and Secret Codes**

**Secret (Symmetric) Key Cryptography**
These have the property that given the encryption key it is easy to find the decryption key. This means that both the sender and the receiver of messages can encrypt or decrypt them. If the encryption key is a function, then there is an easily derived inverse of it!

**Public Key Cryptography**
This means that a sender knows the encryption ‘public’ key he/she does not know how to decrypt a message. Only the decrypter can do this - If the encryption key is a function, then there is NOT an easily derived inverse. An example of an implementation is if the public key is based on the product of two large prime numbers. To decrypt requires the factors - it is hard to factorise the product of two large primes.

**Digital Signatures**
(see also http://www.youdzone.com/signature.html) - from which this is taken!

With his private key and the right software, Bob can put digital signatures on documents and other data. A digital signature is a "stamp" Bob places on the data which is unique to Bob, and is very difficult to forge. In addition, the signature assures that any changes made to the data that has been signed cannot go undetected.

To sign a document, Bob’s software will crunch down the data into just a few lines by a process called "hashing". These few lines are called a message digest. (It is not possible to change a message digest back into the original data from which it was created.) Bob’s software then encrypts the message digest with his private key. The result is the digital signature. Finally, Bob’s software appends the digital signature to document. All of the data that was hashed has been signed. Bob now passes the document on to Pat.

First, Pat’s software decrypts the signature (using Bob’s public key) changing it back into a message digest. If this worked, then it proves that Bob signed the document, because only Bob has his private key. Pat’s software then hashes the document data into a message digest. If the message digest is the same as the message digest created when the signature was decrypted, then Pat knows that the signed data has not been changed.

Plot complication...

Doug (our disgruntled employee) wishes to deceive Pat. Doug makes sure that Pat receives a signed message and a public key that appears to belong to Bob. Unbeknownst to Pat, Doug deceitfully sent a key pair he created using Bob’s name. Short of receiving Bob’s public key
from him in person, how can Pat be sure that Bob’s public key is authentic?

It just so happens that Susan works at the company’s certificate authority centre. Susan
can create a digital certificate for Bob simply by signing Bob’s public key as well as some
information about Bob.

Now Bob’s co-workers can check Bob’s trusted certificate to make sure that his public
key truly belongs to him. In fact, no one at Bob’s company accepts a signature for which
there does not exist a certificate generated by Susan. This gives Susan the power to revoke
signatures if private keys are compromised, or no longer needed. There are even more widely
accepted certificate authorities that certify Susan.

Let’s say that Bob sends a signed document to Pat. To verify the signature on the
document, Pat’s software first uses Susan’s (the certificate authority’s) public key to check
the signature on Bob’s certificate. Successful de-encryption of the certificate proves that Susan
created it. After the certificate is de-encrypted, Pat’s software can check if Bob is in good
standing with the certificate authority and that all of the certificate information concerning
Bob’s identity has not been altered.

Pat’s software then takes Bob’s public key from the certificate and uses it to check Bob’s
signature. If Bob’s public key de-encrypts the signature successfully, then Pat is assured that
the signature was created using Bob’s private key, for Susan has certified the matching public
key. And of course, if the signature is valid, then we know that Doug didn’t try to change the
signed content. Although these steps may sound complicated, they are all handled behind
the scenes by Pat’s user-friendly software. To verify a signature, Pat need only click on it.
See also - tutorial problem attached on booking a cinema ticket.

**Salting Passwords**

This is taken from:

In very many - not to say almost all - Web applications user data is administered, from Web
forum to Web shop. These user data encompass login information of the users which contain
the password besides the user name - and this in plain text. A security leak par excellence.

Why is storing the user name and password in plain text a security leak? Well, imagine a
 cracker gaining system access through eventual OS or server software errors, and being able
to read the user database. As he now knows the user name and password of any arbitrary
user he can now log on as a 'real' user and do whatever he wants with the permissions for
that user - from ordering in the Web shop to character assassination on the forum. And you
are the operator...

How can this security risk be eliminated? Why should we roam far when a proven method
for safe storage of passwords exists since decades: under UNIX, user passwords are stored as
so called 'salted hashes'.

**Salted Hash?**

A hash is a numerical value of fixed length which unequivocally identifies files of arbitrary
length. An example of a hashing algorithm is SHA1, which already figured as the topic of an
ASP article (German). The reader might now say that saving the password as a hash would
be sufficient, but why is this wrong?

The reason for this is that usually so called 'Dictionary Attacks' are run against hashed
passwords - a good example being the MD5 hashed passwords of NT4. This is a Brute Force
attack: all entries in a dictionary were hashed using MD5 and those hash values then are
compared against the password database. Have a guess how quickly some passwords are
found this way.
The intention behind a Salted Hash is to have this type of attack fail by attaching a random value - the so called salt - to each password and only then compute the hash over password and salt. For comparison of the password the salt has to be stored alongside the salted hash, but the only vector of attack is to re-code the dictionary for each individually stored password with the salt - and this takes quite a long time.

Storing the Salted Hash As previously mentioned, we now need to store three fields instead of user name and password: user name, salt and the salted hash of the password. I also mentioned that when these data get into the hands of a cracker he will have a problem using standard attacks and most probably will look for an easier victim.

One point however must be kept in mind: it is now impossible to send a 'password reminder’ email - all that can be done is to generate and send a new password for the user. As a number of mistakes is made in this field, we will begin with the .NET code for generating a truly random password.

Generating Passwords - done right!
The entire class was created in the course of a (C# ASP.NET) community project together with another AspHeute author, namely Alexander Zeitler. In this case the question of how to generate good passwords and how to correctly store them in a database also came up.

[The piece ends with some code for generating passwords.]

Other Protection Mechanisms

Domains An object is a component of a system needing protection - hardware (e.g. CPUs memory segments, disc drives, printers) or software (e.g. processes, files, DBs or semaphores). A domain is a set of (object, rights) pairs - where a right is a permission to (e.g.) read or write to a file, execute a program. A domain can correspond to a single user or a group.

Protection/ Access Control Matrices This rows of this matrix are domains and the columns are objects. However this matrix is very sparse - so other devices are needed. See the next two items for this.

Access Control Lists The matrix is decomposed by columns - for each object, an access control list gives users and their rights.

Capability Tickets The matrix is decomposed by rows - specifies authorised object and operations for a user.

rw permissions in Unix Unix has rw permissions for each file and directory. (The command ‘ls -lg’ will give you these in your own directories - where ‘g’ stands for group.)

Segments - security Segments can also be provided with rw permissions.