Midterm Quiz

The Midterm Quiz is a take-home examination. It is due at 4pm this **Thursday, November 2** at the end of class. Late exams will **not** be accepted.

Policy: The quiz is open book, but *no collaboration is allowed!* You may not communicate with any person about any aspect of the exam until after the hand-in deadline, even if you have already handed in your exam. You may use your notes from this course, class handouts from this year and any textbooks, but *do not use any other sources!*

Instructions: There are five problems, each worth 20 points. Hand in all the problems. Each problem is to be done on a separate sheet (or sheets) of paper. Mark the top of each sheet with your name, You have plenty of time to do the exam, so there is no excuse for sloppiness.

TA support: Rosario will be available for questions about the quiz Monday morning 9:30-12noon in his office NE43-344. Also during the weekend you can send him email at `rosario@theory.lcs.mit.edu` if you have questions.

Bugs, etc.: Eventual bugs in the quiz or any hint the TA may feel inclined to give will be reported to you by email. Check your email regularly during the examination period.

Problem Q-1. Hash functions from symmetric block ciphers

In this problem we will consider two proposals to construct hash functions from symmetric block encryption schemes as DES and RC5.

Let E denote a symmetric block encryption scheme. Let $E_k(M)$ denote the encryption of the 1–block message M under key k. Let $M = M_0 \circ M_1 \circ M_2 \circ \ldots \circ M_n$ denote a message of $n+1$ blocks.

The first proposed hash function h_1 works as follows: let $H_0 = M_0$ and then define

$$H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1} \quad \text{for} \quad i = 1, \ldots, n.$$

The value of the hash function is defined as

$$h_1(M) = H_n$$

The second proposed hash function h_2 is similar. Again $H_0 = M_0$ and then

$$H_i = E_{H_{i-1}}(M_i) \oplus M_i \quad \text{for} \quad i = 1, \ldots, n.$$

The value of the hash function is defined as

$$h_2(M) = H_n$$

For both proposals, show how to find collisions if the encryption scheme E is chosen to be DES.

Problem Q-2. Group signatures

In class we described the concept of *key escrow* for public key encryption schemes. A user chooser her public key E and matching secret key $D = E^{-1}$. She makes E public and shares the secret key D among a set of trustees, so that if needed they can decrypt her messages by putting together their shares.

A related concept is the concept of **group signatures**. Suppose that a corporation has a public key E used to sign official documents. So a document M will be signed by publishing $\sigma = D(M)$ and such signature can be publicly verified by everybody against the public key E by checking

$$E(\sigma) = M$$

However in order for documents to be signed on behalf of the corporation *all* n of the corporation's vice-presidents (VP's) have to agree. One possible way to achieve this is to give a public key to each VP and have each of them sign the document. Only documents with all n signatures will be accepted as "signed." This is however cumbersome since it involves appending and verifying n signatures for each document.

Group signatures solve the problem by sharing the secret key D among all the VP's. Let D_i be the share of the secret key held by VP i. At signing time each VP produces a *partial signature* $\sigma_i = D_i(M)$ and then these signatures are combined in order to produce $D(M)$.

Example: Assume the encryption scheme is RSA and the public key is (N, e) and the secret key is d. Then a signature of a message M is $\sigma = M^d \bmod N$. It is verified by checking that $\sigma^e = M \bmod N$. One way to create a group RSA signature is to have a trusted party generate the keys (N, e, d), make (N, e) public and give privately to VP i a value d_i such that

$$d = d_1 + d_2 + \ldots d_n \bmod \phi(N)$$

Then the trusted party self–destructs. Suppose now that the VP's want to sign a message M. Each VP produces $\sigma_i = M^{d_i} \bmod N$ and the signature is simply set to

$$\sigma = \prod_{i=1}^{n} \sigma_i = \prod_{i=1}^{n} M^{d_i} \bmod N = M^{\sum_{i=1}^{n} d_i \bmod \phi(N)} = M^d$$

which is the correct corporation signature. Notice that it is necessary for the trusted party to self–destruct otherwise the information contained in it would allow someone to sign messages for the corporation without the help of the VP's.

In this problem you will be asked to come up with a group signature scheme for the following signature algorithm. The algorithm is a modification of ElGamal devised by the teaching staff in order to make this problem solvable. So there is no guarantee that such scheme is secure.

The signature scheme you have to use for the problem is as follows. Let p be a large prime, g a generator of Z_p^*, x the secret key and $y = g^x$ the public key.

When the signer wants to sign a message M she chooses a random number k (modulo $p-1$) and computes

$$a = g^k \bmod p$$

then she computes

$$b = xa + k - M \bmod (p-1)$$

The signature is the pair (a, b). In order to verify the signature one has to check that

$$g^{M+b} = y^a * a \bmod p$$

Notice that as in the ElGamal scheme the value k has to remain secret, otherwise the secret x would be easily computable from a signature.

(a) Give a group signature scheme for the signature algorithm described above. As in the example you can assume the existence of a trusted party.

(b) For the signature scheme described above a trusted party is not needed. Modify your solution to part **(a)** so that it does not use a trusted party for the generation of the shares.

Problem Q-3. Mutual Identification

Consider a password scheme that tries to achieve **mutual identification**, i.e. the user authenticating himself to the system and vice versa.

A possible solution for this is the following protocol:

The user sends a first user password to the system.

If the first user password is correct, the system sends a system password to the user.

If the system password is correct the user sends to the system a second user password.

If the second user password is correct the system allows the user to access the system.

Comment on the possible advantages and disadvantages of this scheme.

Problem Q-4. Authenticating executables

Suppose that we want to make sure that when we load an executable from disk to run it, the code has not been infected by a virus or modified in some subtle way. One way of doing this would be to authenticate the code with a MAC computed with a secret key stored in the CPU. The computer would load the code, check the MAC and if they don't match it knows that something has been modified.

(a) Identify the possible weaknesses of the above approach. What alternatives could be used?

(b) The same system could be used for copy protection of software. When the user goes to the vendor he brings the serial ID of his CPU. The vendor now gives the software to the user with attached the same MAC as above. The software will not run on another CPU since the MAC will not match.

Do you see any problems or difficulties with the above approach?

Problem Q-5. UNIXAC

UNIXAC (pronounced "unix-hack") is a new variant of UNIX, invented just for this exam, which has novel access control features. We'll first descibe UNIXAC, and then ask you about its properties.

UNIXAC has a normal UNIX-like hierarchical file system, with typical UNIX naming conventions and linking (where a file name can be declared to be an alias for another file or directory). There are, however, no "protection bits" for each file or directory. The following four permission types are supported. Note that a directory is just a file, too; the permission types are interpreted for directories specially.

- **Read permission** denoted **r**. For a directory, read permission allows one to list the names of the files in the directory.
- **Delete permission** denoted **d**. For a directory, delete permission allows one to delete one or more entries from the directory.
- **Append permission** denoted **a**. For a directory, append permission allows one to add new files to this directory.
- **Execute permission** denoted `x`. As in UNIX, execute permission for a directory allows one to search the directory.

(For a file, UNIX write permission is modelled as UNIXAC delete+append permission.)

Each file `f` in UNIXAC is stored with the names of four associated directories for access-control purposes. These names are stored as fully specified path-names from the root of the file system. The four directories are called the associated r-node, d-node, a-node, and x-node for the given file `f`, and are also denoted `f:r`, `f:d`, `f:a`, and `f:x`. (These pointers are stored as names at the file itself (actually the inode for `f`, in UNIX terminology) not in any of the directories that point to `f`.) Any of these names may instead be **null**. However, `f:r` is always *interpreted* to be non-null by inheritance from parents: if `/f1/f2/.../fn:r` is **null** (or if the named directory doesn't exist) then it is interpreted to mean `/f1/f2/.../f(n-1):r`, and so on, defaulting to a name to the directory `/root` if all such pointers were **null**. The relevant four names are determined using these inheritance rules when the file is opened, using the name supplied to open the file with. (The inheritance rules are why the four names are stored symbolically.)

In addition, each file knows who its owner (i.e. creator) is, and only the owner may change these names, via commands of the form `chr f dr` to set `f:r` to be directory `dr`. Commands `chd, cha, chx` can set the other names. When a file is created, its four names are initially set to **null**.

In UNIXAC, user `joe` has permission `p` (where `p` is one of `r`, `d`, `a`, or `x`) on file `f` if and only if directory `f:p` (or one of its subdirectories, recursively), contains an ordinary file named `joe`. (The contents of file `joe` are not important, merely its existence is important.) A recursive directory listing of the form `ls -R f:r` should list all of the users who have permission to read file `f`.

(a) What is the interpretation of `f:r:d` and `f:r:a`?

(b) Argue that UNIXAC can provide the normal features of access control lists.

(c) Argue that the permission structure of UNIXAC is sufficiently expressive to model the UNIX `rwx` permission structure. That is, for any given setting of the `rwx` permission bits in UNIX, there is a corresponding setting in UNIXAC. (Or, if you find a reason why this is not true, give your reason.) (Do not consider the SUID and SGID bits here, but do consider group permissions. UNIXAC/2 is scheduled to have SUID and SGID bits, or something better.)

(d) What does it mean when `f:r:a` = `f:r`?

(e) Sketch how you might be able to model capabilities within UNIXAC. Can you handle revocation as well? How about limiting propagation?

(f) (Optional) Please note any unusual features, advantages, or disadvantages or UNIXAC.