Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ron Rivest

Handout 11
April 22, 2013
Due: May 6, 2013

Problem Set 5

This problem set is due on *Monday, May 6* at **11:59 PM**. Please note that no late submissions will be accepted. Please submit your problem set, in PDF format, *by email* to `6857-staff@mit.edu`. Submit only one problem set per group (with all of your group members' names on it).

You are to work on this problem set with a group of three or four people. You may choose your own groups for this problem set. If you do not have a group, please email `6857-tas@mit.edu`. Be sure that all group members can explain the solutions. See Handout 1 (*Course Information*) for our policy on collaboration.

Homework must be submitted electronically! Each problem answer must appear on a separate page. Mark the top of each page with your group member names, the course number (6.857), the problem set number and question, and the date. We have provided templates for LaTeX and Microsoft Word on the course website (see the *Resources* page).

Grading: All problems are worth 10 points.

With the authors' permission, we will distribute our favorite solution to each problem as the "official" solution—this is your chance to become famous! If you do not wish for your homework to be used as an official solution, or if you wish that it only be used anonymously, please note this in your profile on the homework submission website.

Problem 4-1. Unlinkable Serial Transactions

In this problem, we will explore *unlinkable serial transactions*, as described by Stubblebine, Syverson, and Goldschlag; the paper is available on the course website. The general model is that:

- There is a server providing some service to multiple clients.
- Clients must "subscribe" initially (in the real world, this may involve payment, age verification, or some other check). After subscribing, clients may use the service as often as they wish.
- The server wishes to verify that every time it receives a request, the client has previously subscribed.
- To protect the privacy of the clients, *the server should not be able to trivially correlate different requests as being from the same client.* Thus, simply making each client provide a username and password does not work, since the server could simply look at all requests coming from the same username. [1]

Many existing services require the first three points, and solve the problem by having users create accounts with usernames and passwords. However, this does not meet the final criterion, since the server could easily look for all transactions associated with one account. (Ideas along the lines of creating multiple "one-time accounts" encounter difficulties with ensuring that the owner of each account is properly subscribed, without tying them together.)

The proposal uses *blind signatures*, a technique for creating digital signatures where the signer does not know the message. The idea works as follows:

- The user signs up for a subscription at which time she submits payment and a message of the form

$$blind(hash(N_1))$$

(e.g. $(r^e hash(N_1)) \pmod n$ where server's $PK = (n, e)$), where N_1 is her initial nonce.

[1] Here we won't consider traffic/timing analysis, or looking at the IP address of the incoming request, or the like; we'll just consider the interaction at the cryptographic/protocol level.

- The server accepts payment and gives the user

$$blind(sign(hash(N_1))$$

(e.g. $r \cdot (hash(N_1))^d \pmod{n}$ where server's $SK = (n, d)$, from which the user can obtain

$$sign(hash(N_1))$$

(e.g. by dividing by r to obtain $(hash(N_1))^d \pmod{n}$). This is the server's hash on her first nonce (which the server hasn't seen).

•Later on, the user can request service by sending a message of the form:

$$\text{"request"}, sign(hash(N_i)), N_i, blind(hash(N_{i+1}))$$

- Upon receiving such a request, the server checks that the nonce N_i is previously unused, but that the signature is valid. If so, it provides service, and also returns

$$blind(sign(hash(N_{i+1}))$$

so the user now has a token for her next request.

For this problem, we will use RSA signatures for the signature algorithm, and SHA-256 for the hash function. The server has a public key (n, e) and a secret key d. It signs messages m by generating $\sigma_m = m^d \pmod{n}$.

We have set up a server that follows the above protocol. The service it provides is simply returning different plaintext ASCII strings. We will skip the "subscription" step, so all the requests will be service requests; thus, there's no need to include the string "request" as part of the request.

On the course website you will find a file `USTclient.py` that includes skeleton code for communicating with the server. (It uses HTTP, so you can access it in other ways if you wish, but using this code is almost certainly easiest.) This code includes the server host:port and its public verification key.

You will also need an initial (nonce,signature) pair $(N_i, sig(hash(N_i)))$ to get started (the one that normally would be provided with subscription). Each student can get ONE such nonce/signature pair by going to `https://6857.scripts.mit.edu:444/2013ps5/index.py` with their MIT certificate. (Remember that the server will only accept each nonce *once*, so be careful with your code!)

Find eight of the strings returned by the server. (Once you find some of the strings, do not share them with other groups!) Submit the strings you found, and any code you used.

Remember that in order to be successfully anonymous, both the nonces and the blinding must be *random*!

Problem 4-2. Digital Signatures.

You are going to design a new type of signature scheme called TwoSign in which the signer of a message could be one of two people, but the verifier of the signature cannot learn who the signer is.

Concretely, consider that each user U has a public key PK_U and a secret key SK_U. In particular, Alice has a public/secret keys pair (PK_A,SK_A), and Bob has the pair (PK_B, SK_B). Alice can use her public/secret keys and Bob's public key ($\mathsf{PK}_A, \mathsf{SK}_A, \mathsf{PK}_B$) can create a signature that can be verified as having been created by **one** of the two parties, but the verifier cannnot tell which one (i.e., cannot tell if it was Alice or Bob that signed the message). In other words, the scheme consists of three routines TwoKeyGen, TwoSign and TwoVerify such that

- $\mathsf{TwoKeyGen}(1^\lambda) = (\mathsf{PK}, \mathsf{SK})$
- TwoSign(M, SK_A, PK_A, PK_B) $= \sigma$
- $\mathsf{TwoVerify}(M, \sigma, \mathsf{PK}_A, \mathsf{PK}_B)$ = True/False

with the properties that

- Correctness: $\mathsf{TwoVerify}(M, \mathsf{TwoSign}(M, \mathsf{SK}_A, \mathsf{PK}_A, \mathsf{PK}_B), \mathsf{PK}_A, \mathsf{PK}_B) = \text{True}$
- Public keys can be input in any order to the verification:

$$\mathsf{TwoVerify}(M, \sigma, \mathsf{PK}_A, \mathsf{PK}_B) = \mathsf{TwoVerify}(M, \sigma, \mathsf{PK}_B, \mathsf{PK}_A)$$

- Unforgeability: no one can forge a signature for a new message that verifies with the public key of party A and party B without having the secret key of either of these two parties.
- Signer anonymity: no adversary can guess with a chance better than $1/2 + \mathsf{negl}$ whether A or B signed the message.

Note that A does not know SK_B and B does not know SK_A. Moreover, A does not need B's cooperation to produce such a signature, only B's public key PK_B. So, you can produce a message that was signed by either yourself or by President Obama (by only using his public key), and no one can tell which of you actually signed it! (Of course, the signature does not convince them that it was Obama either :)). There, it is not reasonable to have the parties agree on a shared key.

Devise such a signature scheme, describe each of its three protocols (TwoKeyGen, TwoSign, TwoVerify), and show it satisfies the three properties above.

Hint: Assume you have an unforgeable signature scheme with the original Diffie-Hellman notion of a signature (this is not referring to Diffie-Hellman key exchange!), where the public-key+verification algorithm maps the message space to the ciphertext space **one-to-one**, and the secret-key+signing algorithm provides the inverse mapping; then $\mathsf{Verify}(\mathsf{PK}_U, \mathsf{Sign}(\mathsf{SK}_U, M)) = M$ where $\mathsf{PK}_U/\mathsf{SK}_U$ are the keys of a user U.

Start by noting that while a signature scheme in the D/H model may be unforgeable for any *given* message, it can't be existentially unforgeable, since x is the signature for the random-looking message $\mathsf{Verify}(\mathsf{PK}_U, x)$. Extend this observation to design your routines.

Hint: Think about two-out-of-two secret-sharing schemes.

Hint: If convenient, you may assume that all PK/SK pairs have the *same* message and ciphertext space, which is the set of *all* bit-strings of length λ. (This is not quite true for RSA, and it is an annoying detail...) Recall that one typically signs a *hash* of the message instead of the message itself; you may also assume that you have a one-way hash function thatmaps arbitrary strings into bit strings of length λ.

Problem 4-3. EvilKeyGen again!

On the quiz, you defined the hiding/Nifty property of EvilKeyGen. In this problem, let's actually construct this evil key generation algorithm!

Suppose the Adversary can surreptitiously replace KeyGen in a public-key encryption scheme software library with EvilKeyGen. A user calling KeyGen is in fact calling EvilKeyGen (without knowing). EvilKeyGen also receives as input the security parameter 1^λ and returns a public-key/secret-key pair $(\mathsf{PK}, \mathsf{SK})$ just as KeyGen would. The EvilKeyGen code may have embedded within it code and cryptographic parameters constructed by the Adversary. Using these, the Adversary could subsequently extract a SK from a PK of a user who used EvilKeyGen, and thus learn the secret key of the user.

A proper EvilKeyGen satisfies the following three properties:

1. **Evil.** Whenever a user generates $(\mathsf{PK}, \mathsf{SK})$ from EvilKeyGen, the Adversary can efficiently determine SK given only PK—that is, he can easily extract the SK from any PK generated by EvilKeyGen. (!)
2. **Hiding/Nifty.** EvilKeyGen is considered to be "*Nifty*" if any user who only *executes* the code and examines the output, but does not *examine* the code, cannot tell if he/she is using KeyGen or EvilKeyGen.

3. **Exclusivity.** Only the Adversary can exercise the Evil property (1) —even if the user examines the code, and sees that it has been hacked, he is unable to extract the SK's from the PK's for PK's produced by EvilKeyGen (say by other users using this hacked software library); the Adversary is the *only* party who can take advantage of the "evil trapdoor" of (1).

Devise such an EvilKeyGen (by providing the algorithmic description of EvilKeyGen) for RSA and argue why it respects each of the three properties above. You can assume that in the standard RSA KeyGen, p and q are randomly chosen safe primes of length λ, and that e is a randomly chosen odd integer of length λ-2.

Hint: To achieve Exclusivity, the adversary should be using his own cryptography. How could the adversary send himself a message via (n, e) (while not breaking the Nifty property)?