University of Maryland
CMSC414 — Computer and Network Security
Professor Jonathan Katz

Problem Set 3

Due *before* class on Nov. 11

1. (**10 points**) Assume a website which requires a user to input a username/password in order to log in. You may also assume that the initial connection between the user and the website is secure (i.e., either it was encrypted or else there were no eavesdroppers at the time the user logged in).

   (a) Assume that once the user logs in, the website stores a cookie on the user's computer which contains the user's username and password; if the user accesses the website again, the site first checks for a pre-existing cookie and — if it finds one — checks the username/password contained therein. Show how an adversary can log in as the user if this cookie is ever compromised.

   (b) Assume now that the website instead computes a hash of the user's name and password (i.e., $h = H(\text{username}, \text{pw})$), and stores this value h in a cookie on the user's computer. When the user visits the site again, the site checks for a pre-existing cookie; if one exists, the server obtains h from the cookie and accepts the user iff $h \stackrel{?}{=} H(\text{username}, \text{pw})$ (of course, the server stores the user's name and password). *Assuming the hash function H is secure*, is this method now secure in case an adversary obtains the cookie?

2. (**90 points**) Implement a basic public-key infrastructure (PKI). The model will contain three types of entities: *certificate authorities* (CAs), *servers*, and *clients*. The requirements should hopefully be clear from the following description of the various functionalities that should be supported:

   - To run your CA program, you should invoke the command "java CA ⟨CAname⟩ ⟨port1⟩". The CA should then: (1) generate a random public-/secret-key pair for a signature scheme and (2) wait for a connection from a server on port1. During the course of a connection, the CA should: (1) receive a name and a public key from a server; (2) sign the received name and public key using its own secret key; (3) send the resulting signature along with CAname to the server; (4) close this connection and be ready to accept additional connections.
   - A server program should be invoked via the command "java server ⟨serverName⟩ ⟨serverPort⟩". Upon being invoked, the server should generate its own random public-/secret-key pair for a signature scheme and then wait for keyboard input (as well as listen for a connection on serverPort). The server then has three possible actions: *obtaining a certificate*, *creating a certificate*, or *showing a certificate*. I explain each of these in turn:

Obtaining a certificate. Upon receiving keyboard input "obtainCert ⟨machineName⟩ ⟨port1⟩", it should send serverName and its public key to the entity listening at port1 on machineName (note that this entity can be either a CA or a server); it expects to receive in return a signed certificate from that entity on its public key (in addition to obtaining the name of the other entity).

Creating a certificate. The server should be ready to accept connections on serverPort from either other servers or clients (note that the server does not know, *a priori*, whether the connecting entity is a server or a client, so this should be determined based on flags in the messages sent when the communication is established). Connections from clients are discussed below. When server s_2 connects to server s_1, s_1 expects to receive a name and a public key from s_2. Then, s_1 will certify the public key of s_2 almost exactly like a CA does, except that in addition to sending the signature and serverName, s_1 should *also* send a certificate chain which connects s_1 to a CA. This will result in s_2 having a certificate chain to that same CA. *Note:* If s_1 has chains to multiple CAs (or multiple chains to the same CA), it may choose one arbitrarily.

Showing a certificate. If a client connects to the server, the server should show a valid certificate to the client. This is discussed when we describe the functionality of a client.

- To run a client, you should invoke the command "java client" (note that the client itself does not have a name, nor will it have its own public key). Upon being invoked, the client should automatically "learn" the public keys of any CAs (one way to do this is to have CAs output their public keys to files with a particular extension, and for clients to read from those files). It should then output to the screen the names of any CAs whose public keys it has learned. Finally, the client program should accept keyboard input of the following form: "verify ⟨CAname⟩ ⟨machineName⟩ ⟨serverPort⟩", which should cause the following steps to occur: (1) the client connects to a server running on serverPort and machineName; (2) the client sends CAname to that server; (3) the server should check whether it has a certificate chain to the CA named "CAname". If it does, it should send the resulting certificate chain back to the client, who will attempt to verify it using the (known) public key of the CA by that name. If the server does not have such a certificate, it should simply respond with some error code.
- In addition to implementing everything as described above, you should give ample documentation describing your implementation. In particular you should describe what signature scheme you decided to use, how you stored certificates, and how servers find a certificate chain from themselves to a CA (if one exists).