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

Handout 4
March 8, 2011
Due: March 18, 2011

Problem Set 3

This problem set is due *online,* at `https://courses.csail.mit.edu/6.857/` on *Sunday, March 20* by **11:59 PM**. Please note that no late submissions will be accepted.

You are to work on this problem set with your assigned group of three or four people, or with your project group. You should have received an email with your group assignment for this problem set. If not, please email `6.857-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 3-1. Reflections on Trusting PRNGs[1]

A reliable random-number generator is important to any cryptosystem. Consider a broken random-number generator that only outputs a limited set of values – an attacker can notice this and then just exhaustively search each possible "random" choice by brute force.

However, chances are that the legitimate users of the system will notice such a lack of randomness quickly. A *malicious* pseudo-random-number generator can be much more devious. In particular, by using cryptography itself and indistinguishability results, it can ensure that numbers appear random to anyone who isn't the malicious party and holds information only the malicious party knows.

Consider the following two scenarios.

1. If Alice is sending a message to Bob signed via DSS, and Randy can predict (but not influence) her random number generator, can he forge one or more messages? How bad is the break?
2. If Alice is generating an RSA keypair and using a "black box" key generator provided by Randy, can he generate keys that appear random but let him compute Alice's private key from her public key? Can he further ensure that, even if another attacker Eve reverse-engineers the key generator and figures out what Randy did, she still cannot compute Alice's private key from the information in the key generator and Alice's public key? If so, how?

Problem 3-2. Discrete Logarithms and Matrices

Let p be a large prime, and let G be the following matrix over $GF(p)$:

$$\begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}$$

Suppose you are given a matrix Y that is known to be of the form

$$Y = G^k$$

[1] This is a reference to Ken Thompson's classic "Reflections on Trusting Trust", which is only tangentially related to this problem, but you should read it anyway if you haven't.

for some positive integer k, where k is less than the multiplicative order of G. (Here *order*(G) is the least t such that $G^t \equiv I$, the 2×2 identity matrix, modulo p.)

Write a Python or SAGE program that takes as input

a prime p, a 2×2 matrix G modulo p, and a 2×2 matrix Y modulo p,

and returns

- the order t of G
- Y's discrete log, $k \equiv \log_G Y \bmod p$, if it exists, otherwise "None"

Your algorithm should run in time proportional to the square root of the answer it returns. (Don't search linearly for t or k, but use one of the cute ideas given in the "Algorithms" section of the Wikipedia page on Discrete Logarithms or in your textbooks. Don't use any built-in SAGE functions for computing orders or discrete logarithms.)

Turn in your code. Explain how it works.

Give the largest d such that your program can solve the above problem in less than one-half hour, modulo p, where p is the first prime after 10^d. (In sage: `p = next_prime(10^d)`)

Problem 3-3. BrokenSSL

On port 6857 of 6857.scripts.mit.edu can be found an HTTPS server. For some incomprehensible reason, however, its RSA private key was generated by an OpenSSL implementation modified as follows:

```
--- openssl-0.9.8k/crypto/rsa/rsa_gen.c.orig    2011-03-09 03:51:07.540568227 -0500
+++ openssl-0.9.8k/crypto/rsa/rsa_gen.c 2011-03-09 04:06:45.240318197 -0500
@@ -101,7 +101,7 @@
        r3 = BN_CTX_get(ctx);
        if (r3 == NULL) goto err;

-       bitsp=(bits+1)/2;
+       bitsp=(bits+1)/10;
        bitsq=bits-bitsp;

        /* We need the RSA components non-NULL */
```

In other words, instead of the length of p and q being roughly equal, one is a tenth the size of the other. This should make the key fairly easy to break.

a. What is the public modulus $N = pq$ and public exponent e?
   There are a couple of ways to answer this. You can visit https://6857.scripts.mit.edu:6857/ in your browser and look at the certificate information. You can also use the `openssl s_client` command (run `man s_client` for more info).

b. What is the private key (p, q, d)?
   (You do not need to show your work.)

An HTTPS connection consists of HTTP layered inside TLS (still colloquially called SSL, although properly that refers to an older version of the protocol). TLS has the following rough structure when used with RSA for key exchange:

- The client sends a ClientHello message, which consists of a list of supported sets of authentication, encryption, and key exchange algorithms, and a random number.

- The server chooses the best of these sets that it supports, and replies with a ServerHello message consisting of the choice and a random number. It also sends a Certificate message.
- The client replies with a ClientKeyExchange message, which consists of a random number called the "pre-master secret" encrypted to the server's public key.
- Here it can also reply with a client certificate in case the server requested client authentication.
- Both parties combine the client random number, the server random number, and the pre-master secret into a "master secret", from which is derived all the keys for the chosen encryption and authentication algorithms.
- This ends the handshake, and all future data is encrypted and integrity-protected with the negotiated algorithms and the generated keys.

See RFC 5246 for the authoritative, gory details, but this should be more or less enough for this problem. One reasonable implementation of TLS is the Python package TLS Lite from `http://trevp.net/tlslite/`. Unlike many other libraries, this is pure Python, as opposed to just a way to access OpenSSL from Python or some other high-level language; this makes it easier to understand how it works.

c. Connect to our server with TLS Lite, without a client certificate. What are the contents of the web page at /ping?

```
>>> from socket import *
>>> from tlslite.TLSConnection import *
>>> s = socket.socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
>>> s.connect(("6857.scripts.mit.edu", 6857))
>>> settings = HandshakeSettings()
>>> settings.minKeySize = 512
>>> t = TLSConnection(s)
>>> t.handshakeClientCert(settings=settings)
>>> t.write("GET /ping HTTP/1.1\r\n")
>>> t.write("Host: 6857.scripts.mit.edu\r\n")
>>> t.write("\r\n")
>>> t.read()
>>> t.read()
...
```

(If you're having trouble with this step please e-mail 6857-tas or come to office hours — this isn't intended to be the difficult part of the problem set.)

Because this key setup procedure can potentially be computationally expensive, the ServerHello message also includes a session ID. The client can request to use the same algorithms and master secret as an existing connection by specifying its session ID in the ClientHello message. (By default a new session is created for each connection.)

The ability to resume a session, however, makes attacking a server easier, since the attacker doesn't have to hijack an existing connection. While this isn't a huge vulnerability, since after all if an attacker couldn't hijack an existing connection we wouldn't really need TLS that much, we'll take advantage of it to make the attack simpler.

A client is accessing our server every minute using a client certificate, and retrieving a web page. At http://6857.scripts.mit.edu/ps3 you can see the traffic logs of this client. Since you have broken the server's private key, you should be able to recover the pre-master secret.

d. Identify the client random number, server random number, encrypted pre-master secret, and session ID from one of these leaked communications.

The pre-master secret is always 48 bytes long. Before encryption, it was padded with the PKCS1 v1.5 scheme, which prepends a random string to it. (The full padding is 0x0002, a series of random bytes, 0x00, and the message.)

e. Recover the 48-byte pre-master secret.

f. Resume the session you have attacked. What are the contents of the web page at /magic-words to an authenticated user?

You can resume a session from an existing TLSConnection to a new one by retrieving the *session* property from the first and passing it to *handshakeClientCert* for the second. Therefore, if you fake up a Session object, you can "resume" a session you haven't used before:

```
...
>>> sess = Session()
>>> from array import array
>>> sess._calcMasterSecret((3, 1),
      array('B', [...]),  # premaster secret
      array('B', [...]),  # client random
      array('B', [...]))  # server random
>>> sess.sessionID = array('B', [...])
>>> sess.cipherSuite = [...]
>>> sess._setResumable(True)
>>> s = socket.socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
>>> s.connect(("6857.scripts.mit.edu", 6857))
>>> t = TLSConnection(s)
>>> t.handshakeClientCert(session=sess, settings=settings)
...
```