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

Handout 2
February 11, 2012
Due: February 25, 2012

Problem Set 1

This problem set is due on *Monday, February 25* 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 your assigned group of three or four people. Please see the course website for a listing of groups for this problem set. If you have not been assigned a group, 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 1-1. Security Policy for Wikipedia

Write a security policy for an implementation of Wikipedia. Make sure to define relevant roles, functions, and policies. Do not include guidelines or policies that are not meant to be enforced. Your policy should be of the sort that would be usable for a re-implementation of Wikipedia by the implementor. If you can't find relevant material on Wikipedia as currently implemented, invent new material as appropriate. Try to be as complete as you can, but emphasize the Wikipedia-specific aspects—in particular, what security goals are the most relevant for Wikipedia? What roles does (or, rather, *should*) Wikipedia have, and what should each principal be allowed to do?

(This problem is a bit open-ended, but should give you excellent practice in writing a security policy. Also, you may actually care about such security policies for designing systems used by large numbers of people—or if you contribute to Wikipedia yourself! We have included sample solutions from similar questions in previous years on the course website.)

Problem 1-2. Overused Pad

It is well known that re-using a "one-time pad" can be insecure. This problem explores this issue, with some variations.

In this problem all characters are represented as 8-bit bytes with the usual US-ASCII encoding (e.g."A" is encoded as 0x41). The bitwise exclusive-or of two bytes x and y is denoted $x \oplus y$.

Let $M = (m_1, m_2, \ldots, m_n)$ be a message, consisting of a sequence of n message bytes, to be encrypted. Let $P = (p_1, p_2, \ldots, p_n)$ denote a pad, consisting of a coresponding sequence of (randomly chosen) "pad bytes" (key bytes).

In the usual one-time pad, the sequence $C = (c_1, c_2, \ldots, c_n)$ of ciphertext bytes is obtained by xor-ing each message byte with the corresponding pad byte:

$$c_i = m_i \oplus p_i, \text{ for } i = 1 \ldots n$$

which we can also write as

$$C = M \oplus P$$

When we talk about more than one message, we will denote the messages as $M_1, M_2, \ldots, M_k$ and the bytes of message M_j as m_{ji}, namely $M_j = (m_{j1}, \ldots, m_{jn})$; we'll also use similar notation for the corresponding ciphertexts.

(a) Here are two 8-character English words encrypted with the same "one-time pad". What are the words?

['99', 'f0', '11', '31', '3f', 'f6', '9d', '52']

['89', 'fa', '1f', '34', '38', 'eb', '8c', '59']

Describe how you figured out the words.

(b) Alice has decided to "fix" this problem with the one-time pad, by arranging the encryption operation so that you can't "cancel" the pad bytes by xor-ing the ciphertext bytes.

She chooses a function f mapping pairs (p, m) of a pad byte and a message byte to an output byte that is invertible for any fixed pad byte p. Thus, there exists another ("decryption") function g mapping pairs (p, c) of pad byte and ciphertext byte to an output byte such that

$$g(p, f(p, m)) = m$$

for all bytes p and m.

The functions f and g are each represented in an implementation as fixed tables of size $(256)^2 = 65536$ entries. These tables are effectively randomly chosen, subject to the above condition.

It is assumed that the adversary (in this case you!) knows what these tables are (but the pad is kept secret). We posted along with this problem set a file `fg.txt` containing the f and g tables. Each table consists of 256 by 256 entries. Each row corresponds to one pad byte and each column corresponds to: one plaintext byte for f or one ciphertext byte for g. For example, for the f table, the value in the first row and the third column corresponds to $f(0, 2)$, namely the ciphertext corresponding to a pad byte of 0x00 and a plaintext byte of 0x02.

Given a pad $P = (p_1, \ldots, p_n)$ and a message $M = (m_1, \ldots, m_n)$, Alice encrypts the message byte-wise to obtain ciphertext $C = (c_1, c_2, \ldots, c_n)$ where

$$c_i = f(p_i, m_i) \ .$$

Decryption is done as expected, using the g table:

$$m_i = g(p_i, c_i) \ .$$

Alice is confident that she can now re-use her pad, since there is no apparent way that one can use algebra to "cancel" the effect of the pad on the message to obtain the ciphertext. That is, xor-ing ciphertexts doesn't seem to do anything useful for an adversary. So, she feels that she can now re-use a pad freely.

You are given the file `tenciphs.txt`, containing ten ciphertexts $C_1, C_2, \ldots, C_{10}$ produced by Alice, using the *same* pad P. You know that these messages contain valid ASCII text—the only characters Alice uses are those in the range 0x00–0x7F, inclusive.

To facilitate this task, we provided Python code (`encdec.py`) to load the f and g tables as well as to encrypt/decrypt using them.

Submit the messages and the pad, along with a careful explanation of how you found them, and any code you used to help find the messages. The most important part is the explanation.

(c) **Optional and not for credit:** We separately provide four ciphertexts (in the file `fourciphs.txt`) encrypted with a different pad. Are you now able to decrypt the values? (It might not be possible...we haven't tried!)

Problem 1-3. In a Pineapple Under the Sea

The hash function SHA-3 (Keccak) is designed as a *sponge function*. At a high level, such algorithms have a fixed-size internal state of size $r + c$ bits, and can process an arbitrary amount of input by taking r input bits at a time and combining them with its internal state. After processing the input, the algorithm then outputs r bits at a time in a similar manner. Thus, the internal state can be viewed as "absorbing" the input one block at a time, and then the output is "squeezed" out of the internal state one block at a time.

This general design principle is reminiscent of `/dev/random` on Unix-like operating systems. This code maintains an entropy pool (its "internal state"), and updates the pool with new "random-looking" input from the environment of the machine (such as interrupt timing information). When a program requests random bits, the code returns a function of the bits in the entropy pool, as well as updating the pool.

The following references might be useful in this problem:

- A description of sponge functions from the SHA-3 designers: `http://sponge.noekeon.org/`
- A description of the idea used by `/dev/random`, within its source (mostly the "Theory of Operation" section): `http://stuff.mit.edu/afs/sipb/contrib/linux/drivers/char/random.c`
- *An Analysis of the Linux Random Number Generator*, by Gutterman, Pinkas, and Reinman.
  `http://eprint.iacr.org/2006/086.pdf`
  Note that you are not responsible for the content of this paper; rather, it is intended as a reference (largely for the authors' descriptions of the operation of `/dev/random`).
- *Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices*, by Heninger, Durumeric, Wustrow, and Halderman.
  `https://factorable.net/weakkeys12.extended.pdf`
  Again, you are not responsible for the content of this paper, but section 5.1 in particular gives a description of the various entropy sources used by `/dev/random`.

(a) Compare and contrast the security and functionality goals of SHA-3 and `/dev/random`. What assumptions do each make about their input? What security properties are desired from each?

(b) Ben Bitdiddle proposes changing the implementation of `/dev/random` to use SHA-3 directly:

- The entropy pool will have $r + c$ bits.
- New random bits will come from the same sources as before, except they will accumulate r at a time, at which point they will be mixed with the entropy pool using the SHA-3 "absorbing" method.
- Whenever a total of r bits are extracted, the "pot will be stirred" in the SHA-3 manner.

Is this a reasonable idea? What security properties of SHA-3 are important in trying to answer this question? Are there additional properties that one would want to try to prove that SHA-3 has in order for this to be reasonable?