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

Handout 3
February 25, 2012
Due: March 11, 2012

Problem Set 2

This problem set is due on *Monday, March 11* 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 `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 2-1. Encoding Schemes

In this problem, we will explore how to pad or encode a text before hashing it. Let $[k]$ denote the set of symbols $\{0, 1, \ldots, k-1\}$. Thus, $[2]$ is the binary alphabet, $[256]$ is the set of bytes, $[10]$ is the set of decimal digits, etc. Let $[k]^*$ denote the set of strings consisting of any finite number of symbols from $[k]$, including zero. For example, all the strings (4), (255, 1), (4, 8, 9), and () are in $[256]^*$.

Suppose we wish to hash arbitrary strings from $[r]^*$. Typically $r = 2$ (for a bit representation) or $r = 256$ (for a byte representation) in applications.

We say that a string x is a *prefix* of a string y in the ordinary sense of the word: for example the string "form" is a prefix of the string "formal", and 100 is a prefix of 10000.

A typical hash function might only accept strings in $[B]^*$ for some $B > r$. For example, MD5 is a simple hash function that works on strings in $[B]^*$, where $B = 2^{512}$ and B denotes the size of a block: 512 bits. Any message to be hashed should consist of a sequence of blocks, each of length 512 bits, so the message length in bits needs to be a multiple of 512.

When $B > r$, not every string x in $[r]^*$ is an element of $[B]^*$. For example, the binary string 1011 in $[2]^*$ has length 4, and so is not suitable as input to the simple MD5 algorithm.

Padding a message means appending bits at the end of the message until a desired length is obtained. Encoding is more general and it can include padding as well as transforming message symbols. Let e be an encoding function

$$e(x) = y$$

that transforms any input message $x \in [r]^*$ into an input $y \in [B]^*$. Then, y can be operated on by the hash function. The use of e in this way transforms a hash function with domain $[B]^*$ to a hash function with domain $[r]^*$. That is, we start with a simple hash function h' with domain $[B]^*$ and transform it into a more useful hash function h with domain $[r]^*$ via

$$h(x) = h'(e(x)) \ .$$

For cryptographic reasons, the function e should satisfy the following properties:

–It should be *invertible*: given $y = e(x)$, it is the case that x is uniquely determined. That is, the function e is collision-free (otherwise h is trivially not collision-free).

–It should be *prefix-free*. That is, for any distinct strings x and x' in $[r]^*$, if we let $y = e(x)$ and $y' = e(x')$, then it is the case that neither of y or y' is a prefix of the other. (This property is needed for preventing length-extension attacks, and for demonstrating that h is indifferentiable from a random oracle – the exact definition of indifferentiable is omitted here).

For the following problems, assume that $r = 2$ and $B = 2^{512}$. Namely, the messages are represented as strings of bits, but the hash function expects blocks of 512 bits.

(a) Consider the encoding function e_1 that performs the following padding of an input message x: e_1 appends enough 0's to the end of x until its length in bits is a multiple of 512; if the input's length was already a multiple of 512, it does not append anything. Show that e_1 does *not* meet *either* of the given criteria.

(b) Consider the encoding function e_2 that appends a 1 bit to the end of its input x, and then appends enough 0's afterwards until the result y has a length in bits that is a multiple of 512. If x's length in bits was already a multiple of 512, it still performs this padding until the next multiple of 512. Show e_2 is invertible but not prefix-free.

(Note that e_2 is the encoding function actually used in MD5.)

(c) Consider the encoding function e_3 with range $[B+1]^*$ defined by applying e_2 to obtain a sequence of 512-bit blocks, and then appending a new symbol EOF to the end of the sequence. Show that e_3 is both invertible and prefix-free. Here the symbol EOF $\in [B+1]$ is represented by the value B, which is a symbol not in $[B] = 0, 1, \ldots, B-1$; we have expanded the alphabet by one symbol.

(However, we have "cheated" by introducing this new symbol EOF!)

(d) Read the web site

`http://infoweekly.blogspot.com/2010/06/prefix-free-codes.html`

to see a very cute and efficient way of encoding strings in $[B+1]^*$ as strings in $[B]^*$. (At the cost of lengthening the string by perhaps a couple of symbols. Just for the record, there is a easy way to encode a string that has an EOF by doubling the length, but that is probably too expensive for us.) Call the encoding given in this web site, f:

$$f : [B+1]^* \to [B]^* \ .$$

Argue that the encoding function:

$$e(x) = f(e_3(x))$$

maps strings from $[r]^*$ to $[B]^*$ in a manner that is both invertible and prefix-free. (You may use the claimed properties of the f described there.)

Note that the paper referred to from the web site gives a nice example of how f works.

Problem 2-2. One-Wayness and Collision Resistance

In class, we saw a definition of hash functions that allowed *arbitrary-length* binary input (that is, h maps $\{0,1\}^*$ to $\{0,1\}^d$ for some fixed d). Here we will consider hash functions that take fixed-length input as well, namely, h maps $\{0,1\}^\ell$ to $\{0,1\}^d$, where ℓ is some fixed value.

(a) Give an example of a hash function (possibly with fixed input length and not necessarily compressing) that is collision-resistant but not one-way.

(b) Assume that h is a hash function that maps $\{0,1\}^{n+1}$ to $\{0,1\}^n$ and assume that for every hash value $y \in \{0,1\}^n$, there are exactly two values $x_1, x_2 \in \{0,1\}^{n+1}$ such that $h(x_1) = h(x_2) = y$. Prove that if h is collision-resistant, then h is one-way. (That is, if there is an efficient algorithm that can break one-wayness with nonnegligible probability, then we can construct an efficient algorithm that can generate a collision with nonnegligible probability.)

Problem 2-3. Buffer Overflow Attack

In this problem, you will mount a buffer overflow attack on the authentication system of an application. This authentication system guards access to important classified documents. Access to these documents is only granted to individuals knowing a secret password, and you are not one of those individuals. Nevertheless, your goal is to get successfully authenticated.

You will be given a virtual machine image 6857VM.zip on the course website. This VM makes sure that all of you have the same testing conditions and can reproduce the buffer overflow attack.

To run the VM, you should download VMware Player (`http://www.vmware.com/products/player/`) if you have a Linux or Windows system. For Mac users, MIT has a site license for VMware Fusion and you can download it from here `http://ist.mit.edu/vmware-fusion/4x/mac/download`.

Now open 6857VM. Your account name is `student` and your password is `6857`.

Here are some basic Unix navigation commands `http://www.tjhsst.edu/~dhyatt/superap/unixcmd.html`. You may find handy `ls` and `cd`. If you want to use an editor, both `vi` and `emacs` are already installed. If you need root privilege to install something else, let us know. There should be no need for this, but if something facilitates your work, feel free to ask us.

In your home folder, you will find the authentication program called **login**. You can run it by typing

```
$ ./login
Type your password:
```

If you knew the secret password, you could type it and login successfully. However, you do not have this password.

In the source folder `src`, you can see the source code of `login.cc`, the authentication system. It contains the hash of the secret password `expectedhash`, it gets a password as input `pw`, and it checks that the hash of the input password matches the expected hash (the hash of the secret password). Of course, trying to invert the hash is not a good idea.

Instead, you should mount a buffer overflow attack on the login system by providing a certain input to this program. Since this input can be binary data, it might be easier for you to write it in a file and then provide this file as input to the `./login` program by running `./login < attackinput`, where `attackinput` is the name of the file.

Feel free to change `src/login.cc` and play with it in any way that helps you. To compile the source code `login.cc`, use `make`. But you should not change the executable `login` (the one outside of the `src` folder) because we will check your response against our unchanged `login` code so it will not be of help to you.

(a) Mount a buffer overflow attack and login successfully. Explain your method for doing so. Also, turn in a file `attackinput` such that when we run `./login < attackinput`, we obtain the correct authentication message. (Do not turn in the resulting VM image!)

(b) The `g++` compiler automatically provides protection against buffer overflows. We had to turn off this protection using the `-fno-stack-protector` flag (as you can see in the `Makefile`). The measure `g++` takes is to introduce a canary random variable immediately after the return address and then check, upon return, if this random value is unchanged. Explain how this measure can help with buffer overflow attacks and how it would (or not) thwart your attack.

Problem 2-4. Real-World Security

The security consulting firm Mandiant recently released a report detailing a series of coordinated and persistent electronic espionage attacks against a number of targets including prominent US businesses. The attacks appear to have been carried out by a skilled and well-funded group (designated "APT1") located in China over a period of years, which Mandiant suspects is supported by the Chinese government.

`http://intelreport.mandiant.com/Mandiant_APT1_Report.pdf`

Imagine that you are the chief security officer at a major US business. Choose two types of attacks employed by APT1 as described in this report and summarize them here. For each of these attacks, give two steps you would take in response. Justify your answer (both in terms of real-world security effectiveness and cost, including efficiency cost). Try to keep your overall answer between one-half and one typewritten page.