Massachusetts Institute of Technology — Handout 4
6.857: Network and Computer Security — September 13, 2001
Professor Ronald L. Rivest

Problem Set 1

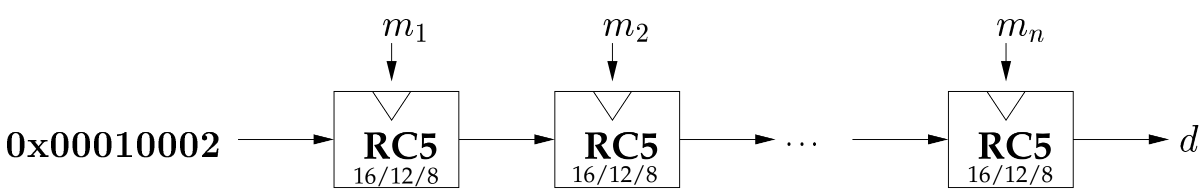
This problem set is due on *Thursday, September 20, 2001* at the beginning of class. Late homeworks will *not* be accepted. You are to work on this problem set in groups of three or four people. Homeworks turned in by individuals, pairs, pentuples, etc. will not be accepted. Be sure that all group members can explain the solutions. Please re-read the homework guidelines in handout 1. If you do not have a group, seek partners on `6.857-students@mit.edu`.

Mark the top of each sheet with your names, 6.857, the problem set number and question, and the date. **Type up your solutions**[1] and be clear. Each problem should begin on a new sheet of paper. That is, you will turn in problems 1 and 2 in separate piles of paper. Points may be deducted if your TA has problems understanding your solution. We strongly recommend you get started on this problem set early as there is some programming involved.

An anonymous TA discovered in his undergraduate classes that by writing problem set solutions in LaTeX, he would receive a good grade even if his solution was wrong. We can't guarantee this feature in 6.857, but writing your solutions in LaTeX will certainly give you our favor. For your convenience, the raw LaTeX version of this problem set is available on the 6.857 Web site.

Problem 1-1. Hash functions

Ben Bitdiddle decided to impress us with a new hash function design that relies on the security of a well-tested encryption algorithm. Ben claims that even though his digest output is small, it will take someone several days to invert his function. The hash function is described by the picture below. The input message m is broken into 8-byte blocks $m = m_1, m_2, ..., m_n$. The last block is padded with zeros on the right as needed. The first block, m_1, is used as the key to encrypt the initial vector `0x00010002` using the RC5-16/12/8 cipher. Each subsequent message block is used as the key to encrypt the output of the previous encryption operation. The message digest, $h(m) = d$, is the result of the final encryption operation. RC5-16/12/8 is a cipher that operates on two 16-bit words at a time, has 12 rounds, and requires an 8-byte key. More information on this cipher can be found at `http://theory.lcs.mit.edu/~rivest/Rivest-rc5.ps`.



For example, the digest of the message "BigHash" is:

$$h(\text{“BigHash”}) = \texttt{0x7044E5E1}$$

[1] We will distribute our favorite solution for each problem to the class as the 'official' solution – this is your chance to become famous!

(a) Describe an attack that inverts this hash function. That is, given an arbitrary value, your attack should find a message that hashes to that value. Evaluate the running time of your attack and the probability of success [if applicable].

(b) Implement your attack and find a message that hashes to `0xDC98 F5EF`. Please turn in a copy of your code as well. The course Web site points to C files that contain code for the hash function and the RC5 library. In order to hash a message, type:

```
>gcc -c hashlib.c
>gcc -o hash hash.c hashlib.o
>./hash BigHash
msg: BigHash --> 7044 E5E1
```

However, you are free to implement your attack in any language.

Problem 1-2. Le BigMAC

Louis Reasoner recently entered the Web security business. He is a consultant for the Back Street Journal, an online e-zine that tracks the events of pop culture icons such as the WannaBoys and Dream Street. The BSJ offers a premiere service for paid subscribers. Customers can purchase items and participate in auctions. After a customer logs in with a username and password, the BSJ Web server sets a cookie-based authenticator. Louis designed the following scheme for the BSJ using the proprietary BigMAC algorithm:

$$\text{Cookie} = (\text{username, expiration time, BigMAC}_k \text{ (username, expiration time)})$$

The values in the cookie are properly marshalled to disambiguate fields. As shown in Figure 1, the BigMAC algorithm takes as input a username, **u**, and an expiration time, **e**. The output is a pair (**l**, **h**). BigMAC uses the AES[2] block cipher in a Feistel-like form to transform its inputs into a MAC. The $\oplus$ denotes an exclusive-OR.

The AES block cipher takes a secret server key, **k**, and a 128-bit plaintext input to produce a ciphertext block. The key can be 128, 192, or 256 bits. The BSJ randomly generates the key **k** from a Lava Lamp[3]. AES_k has the signature $\{0,1\}^{128} \rightarrow \{0,1\}^{128}$.

The usernames and expiration times all consist of 128 or fewer bits. When a username or expiration time is less than 128 bits, it is padded on the right with null bytes (0x00). The key is kept secret on the BSJ Web server; the same key is used in all AES encryptions.

That is,

$$\begin{aligned}
\text{BigMAC}_k(u,e) &= (u \oplus \text{AES}_k(e), (\text{AES}_k(u \oplus \text{AES}_k(e))) \oplus e) \\
&= (u \oplus f, g \oplus e) \\
&= (l, h)
\end{aligned}$$

[2] `http://csrc.nist.gov/encryption/aes/`

[3] `http://www.maa.org/mathland/mathtrek_5_7_01.html`

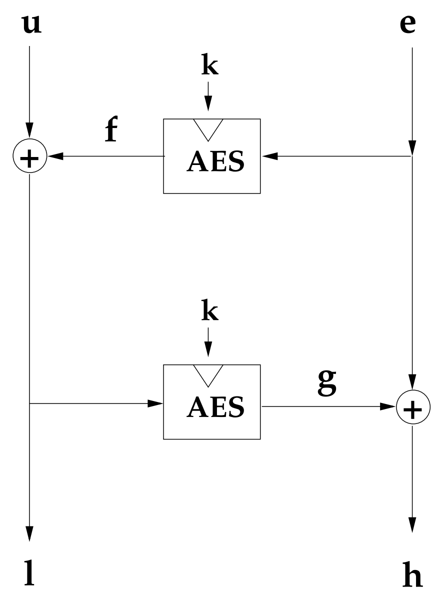


Figure 1: Louis Reasoner's BigMAC scheme. Variables e, f, and g are not output by the BigMAC, but are merely labelled for your convenience.

The Web site will allow you to create new accounts and set passwords for any username not already created. The Web site allows both printable and non-printable characters in usernames. The account login process can be modeled as a function α which takes as input a username and password to produce a cookie-based authenticator:

$$\alpha(u, password) \rightarrow \begin{cases} (u, e, (l, h)), & \text{if the password is correct for the username } u \\ & \text{where } e \text{ is a timestamp}^4 \\ & \text{and } (l, h) = \text{BigMAC}_k(u, e) \\ \text{INVALID}, & \text{otherwise} \end{cases}$$

After logging in and receiving a cookie-based authenticator, one can model the verification routine on the BSJ Web server as:

$$\omega(\text{cookie}) \rightarrow \begin{cases} \text{true}, & \text{if cookie can be unmarshalled to } (u, e, (l, h)) \\ & \text{and if } e > \text{the current time} \\ & \text{and BigMAC}_k(u, e) = (l, h) \\ \text{false}, & \text{otherwise} \end{cases}$$

You've been called in by the BSJ to give a second opinion.

(a) Explain using the notation above how an interrogative adversary can make an existential forgery for this user authentication scheme. That is, implement an adaptive chosen message attack. State your assumptions.

(b) Louis decided that users ought to have control over their expiration times. After all, some paranoid users may want short expiration times[5]. Explain how an interrogative adversary can

[4] You can assume the timestamp is in seconds since UNIX epoch time approximately 2 hours from the time of login. This detail is not important for the break.

[5] For instance, X10.com allows users to specify expiration times for an advertising opt-out CGI script.

make a selective forgery for this user authentication scheme if the account login process is instead:

$$\alpha(u, password, e) \rightarrow \begin{cases} (u, e, (l, h)), & \text{if the password is correct for the username } u \\ & \text{where } (l, h) = \text{BigMAC}_k(u, e) \\ \text{INVALID}, & \text{otherwise} \end{cases}$$

For forging an authenticator for a user u, the better solutions will never have to query α with u.

Note that a forgery does not include the exact replay of an entire cookie previously returned by α. That is, if you ever query for $(u, e, (l, h)) = \alpha(u, \text{password})$ or $(u, e, (l, h)) = \alpha(u, \text{password}, e)$, you cannot simply replay $(u, e, (l, h))$ as a forgery.

You can query the BSJ Web server a reasonable number of times. An existential forgery is a forgery of an authenticator for some username, not necessarily of your choosing or having any meaning. A selective forgery is a forgery of an authenticator for any user of your choosing.