Cryptography 1, Homework 2

Mark Vijfvinkel & Aram Verstegen
0863002, s4092368
Radboud University

September 18, 2013

1 Majordomo

There was some debate on how interpret the rotation operation; whether the rotation means simply a left shift (rotation without carry) or a rotation with carry. The former is trivially forged, as only the last $\frac{32}{4} = 8$ bytes of the input are needed to compute the secret; everything before that would fall off on the left side of the 32-bit state. We therefore continue on the assumption that rotation with carry is implied. (For example, if we rotate bits 11110001 to the left by 4 bits we would end up with 00011111 rather than 00010000.)

Majordomo actually uses XOR, rather than addition as described in the assignment. Since both XOR and addition are associative and commutative, there is nothing stopping us from applying the algorithm in reverse to compute the 32-bit state for an unknown secret, based on a valid e-mail address / cookie pair.

The truncation to 32-bits is what makes the operation non-commutative, but because of the order in which the secret and address are concatenated, the information lost through truncation to 32 bits can be recovered from the address to get back to the 32-bit state which h was in after processing the secret.

To subscribe God@heaven.af.mil to the mailing list the following steps need to be taken:

1. The attacker subscribes himself to the mailing list, for example eve@evil.com;
2. The attacker will then receive the legitimate 32-bits cookie as usual;
3. The attacker applies $h()$ in reverse to recover the 32-bit state, let's call it $recovered\_state = h^{-1}(cookie, address)$ which, for each byte in the reversed e-mail address;
   - Rotates the 32-bits number (cookie) to the right by 4 bits;
   - Subtracts (or XORs) the byte value of the current character in the e-mail address;

- Keeps rotating and subtracting/XORing until it has subtracted and rotated the first character in the e-mail address;
- Now the attacker is left with a 32-bits value that is $h()$'s internal state for the secret string - in other words we have $h(secret)$.

4. While this does not give the attacker the secret, we have the 32-bit state from which to compute $h(secret||address)$ using $initial\_state = recovered\_state$ rather than starting from initial state zero. For this we will use the modified function h_2.

   $h(secret||addresses) = h_2(secret||addresses, 0) = h_2(address, recovered\_state)$

5. The attacker now proceeds to send an email subscribing the victim to the mailing list.
6. After waiting for a bit (the average time for Majordomo to generate and send the cookie, which could be timed in step 1 and 2) the attacker replies with the previously calculated cookie in another forged email.
7. Majordomo finds this cookie correct and activates the victim's subscription.

There are several ways to combat this attack. A simple fix would be to use the secret last, so $h(ak)$ instead of $h(ka)$. The attacker cannot compute forwards beyond the e-mail address bytes because the secret is unknown to him and the result he obtains has been truncated to 32-bits so some crucial information is missing. At least this would make the algorithm more strongly resemble a one-way hash function.

But this is still vulnerable to forgery, because the attacker may collect a number of e-mail address / cookie pairs and use a linear solver (or brute-force) to recover the secret by guessing the bits that fell off. A much better way would be a proper keyed MAC algorithm like HMAC based on a cryptographically secure hash function that is still considered secure.

From the Majordomo developers mailing list we found out that the developers were actually quite concerned about the kind of attack this assignment is about. Here is an entertaining comment from one of the threads about the gen_cookie feature:

> ... Second, with the checksum algorithm used, how difficult is it going to be for an attacker to figure out the secret given a single or small handful of samples? I did some quick experiments on my main machine (a 66 MHZ 486), and was able to check about 333 seeds/sec, which means it would take about 5 months to do a brute-force search of a 32-bit seed space; however, would a brute force search actually be necessary? Or is there a more elegant way of determining the seed? I don't know, I'm not a checksum wiz, but I've seen approaches like this fail before, for reasons like this. I wonder

> how long it would be before a "hack_majordomo" script appeared in the underground to figure out a site's secret based on a few samples. I don't _know_ that the algorithm has this problem, but I don't know that it doesn't, either, and I'm not in a position to make that determination. The nice thing about cryptographic checksum algorithms like MD5 and SNEFRU is that a lot of people have spent a lot of energy demonstrating that they _don't_ have such weaknesses.

[1] (Note that SNEFRU was known to be broken as early as 1993. Also, somebody else in the thread proposed a method of hashing which is vulnerable to a hash-length extension attack.)

The script hack_majordomo.py is attached.

2 Carter-Wegman MACs

The tricky part, and the part on which we fumbled a bit, was to express m in base-p. The first thing we did was to find the exponent of the leading term, by finding the biggest power of p that was still smaller than m, this proved to be p^4. But with a little more thought we realised:

$$leading\_exponent = \lfloor \log_p m \rfloor = 4$$

We found the coefficients for these powers of p by dividing m by them, and then subtracted this term from m and repeated these steps until $m = 0$. With a bit of trial and error, we saw we can get these coefficients in 1 step:

$$coefficient_i = m \div p^i \pmod p$$

Which resulted in the polynomial:

$$m = 454 \cdot p^4 + 351091 \cdot p^3 + 251633 \cdot p^2 + 245176 \cdot p + 411981$$

After that it was simply a matter of inserting the parameters r and s.

$$a = 454 \cdot r^5 + 351091 \cdot r^4 + 251633 \cdot r^3 + 245176 \cdot r^2 + 411981 \cdot r + s \pmod p$$

$$a = 686886$$

The script mac.py is attached.

References

[1] Majordomo-workers archive, march 1996. `http://www.greatcircle.com/lists/majordomo-workers/archive/majordomo-workers.199603`. Accessed: 2013-09-18.