Homework 2: Symmetric Crypto
Due at 11:59PM on Monday Feb 23, 2015 as a PDF via websubmit.

February 17, 2015

Submission policy. This assignment MUST be submitted as a PDF via websubmit and MUST include the following information:

1. List of collaborators
2. List of references used (online material, course nodes, textbooks, wikipedia, etc.)
3. Number of late days used on this assignment
4. Total number of late days used thus far in the entire semester

If any of this information is missing, at least 20% of the points for the assignment will automatically be deducted from your assignment. See also discussion on plagiarism on the course syllabus.

Administration. This homework will be administered by Aanchal Malhotra.

Exercise 1. PRFs. Recall the security definition for a pseudorandom function (PRF):

An adversary D is given oracle access to an unknown oracle $\mathcal{O}$. The game master flips a coin, and sets $\mathcal{O}$ to be a random function (RF) with probability $\frac{1}{2}$, and a PRF with probability $\frac{1}{2}$. If a PRF is chosen by the game master, then a fresh random bit string is chosen to the key the PRF. Now, the adversary D does not know whether $\mathcal{O}$ is a RF or PRF (or its key). However, by Kerckhoff's law, he does know the description of the PRF. He can query the random oracle (*i.e.,* choose x and obtain $y = \mathcal{O}(x)$), and then he must decide if $\mathcal{O}$ was an RF or PRF.

A PRF is secure if for every adversary D running in a reasonable amount of them, we have that $\Pr[D \text{ says RF} | \mathcal{O} = PRF] \approx \Pr[D \text{ says PRF} | \mathcal{O} = PRF]$

Consider a function f that takes in a key k of length n and maps from n-bit inputs to n-bit outputs. Let $n = 128$.

1. Let $f_k(x) = x + k^2 \mod 2^n$. Prove the f is not a pseudorandom function. (To do this, write down the algorithm that D uses to "win" the security game described above.)
2. Let $f_k(x) = (x + k)^2 - (x - k)^2 \mod 2^n$. Prove the f is not a pseudorandom function.

Exercise 2. MACs & Encryption schemes. Let f be a pseudorandom function (PRF). f takes in a key of length n and an input of length $2n$ and produces an output of length n. Let $n = 128$.

In the question below, the symbol $||$ means concatenation and the symbol $\oplus$ is a bit-wise XOR and the symbol $|m| = n$ means the bitstring m has length n.

1. Prove that the following MAC for messages of length $5n$ is insecure: The shared key is a random bitstring $k \in \{0,1\}^n$. To authenticate a message $m_1||m_2||m_3$ where $|m_1| = n, |m_2| = |m_3| = 2n$, compute the tag $f_k(m_1||f_k(m_2))||m_3$.

2. Prove that the following MAC for messages of length $5n$ is insecure: The shared key is a random bitstring $k \in \{0,1\}^n$. To authenticate a message $m_1||m_2||m_3$ where $|m_1| = n, |m_2| = |m_3| = 2n$, compute the tag $f_k(m_1||f_k(m_2))||f_k(m_3)$.

3. The following is a CPA-secure encryption scheme. The shared key is a random bitstring $k \in \{0,1\}^n$. To encrypt a message m of length n bits, choose a random $2n$-bit string r and output the ciphertext
$$r||(f_k(r) \oplus m)$$
   - Write down the decryption algorithm.
   - We won't ask you to prove that this is secure. But, the scheme should remind you of the one-time pad. Explain why, in no more than 2 sentences.

4. Suppose we slightly modify the encryption scheme above, as follows. The shared key is a random bitstring $k \in \{0,1\}^n$. To "encrypt" a message m of length $2n$ bits, choose a random n-bit string r and output the ciphertext
$$r||(f_k(m) \oplus r)$$
   Prove that this is not an encryption scheme.

5. Now we slightly modify the encryption scheme again. Instead of using a PRF, we use a collision resistant hash function H to "encrypt" our message; H maps $2n$-bit string to n bit strings. The shared key is a random bitstring $k \in \{0,1\}^n$. To "encrypt" a message m of length $2n$, choose a random n-bit string r and output the ciphertext
$$r||(H(m) \oplus r \oplus k)$$
   Prove that this is not a CPA secure encryption scheme.

Exercise 3. Secure Channels. A "secure channel" allows Alice and Bob to communicate with confidentially, integrity, and authenticity. In this question we will consider secure channels that can be constructed using symmetric keys.

The scheme. The standard construction for this is the "encrypt-then-MAC" approach, which proceeds as follows:

- Key Generation: Choose a random key k_1 and another random key k_2.
- To send a message m, compute $c = \mathsf{Enc}_{k1}(m)$ and then compute $t = \mathsf{MAC}_{k2}(c)$. Send $y = c||t$. We will call this algorithm $\mathsf{Send}_{k1,k2}(m)$.
- To receive a message y, first parse it as $c||t$. Then, output "fail" if $\mathsf{Ver}_{k2}(c,t) = 0$; otherwise, output $m' = \mathsf{Dec}_{k1}(c)$. We will call this algorithm $\mathsf{Receive}_{k1,k2}(y)$.

The security definition. Here is a simplified version of the security definition for secure channels. The security game is as follows:

- The adversary is given access to a Send oracle; he can ask the oracle to compute $\mathsf{Send}_{k1,k2}(m)$ on any message m he chooses.
- The adversary must output $y*$ such that $\mathsf{Receive}_{k1,k2}(y*)$ is not "fail". $y*$ must not be the answer to any queries he made to the Send oracle.

We have a secure channel if no adversary can win this game with more than negligible probability.

Why is this the right scheme? For many years, there was a debate as to a whether a secure channel should encrypt-then-authenticate (as described above) or encrypt and authenticate, or authenticate then encrypt. We won't ask you to prove that the encrypt-then-authenticate scheme satisfies the definition of security for secure channels; Instead, we discuss why the other two approaches lost the debate.

1. The "encrypt-and-authenticate" scheme modifies the Send algorithm as follows. First we compute $c = \mathsf{Enc}_{k1}(m)$ and then we compute $t = \mathsf{MAC}_{k2}(m)$, and send $y = c||t$. Write down the $\mathsf{Receive}$ algorithm for this scheme.
2. Consider the following implementation of the "encrypt-and-authenticate" scheme. Our MAC will be HMAC. Our encryption algorithm Enc will be the following strange scheme:
   - Encode every bit '0' in the message m as '00'. Encode every '1' in the message m as: '01' with probability $\frac{1}{3}$, '11' with probability $\frac{1}{3}$ and '10' with probability $\frac{1}{3}$. (Thus, the encoding of '001' could be '000010'; the encoding of '111' could be '111101'.)
   - Apply a CPA-secure stream cipher to the encoded message.

   One can show that this strange encryption scheme is CPA-secure.

   Answer the following questions:

   (a) Write down the decryption algorithm for the encryption scheme described above.

   (b) Write down the $\mathsf{Receive}$ algorithm for an encrypt-and-authenticate scheme that uses our strange encryption algorithm as Enc and HMAC as MAC.

   (c) Present an attack that proves that this scheme does not satisfy our definition of secure channels.

(d) Now present an attack that proves this scheme does not satisfy the definition of CCA security.

3. The "authenticate-then-encrypt" scheme modifies the Send algorithm as follows. First we compute $t = \mathsf{MAC}_{k2}(m)$, and then we compute $y = \mathsf{Enc}_{k1}(m||t)$ and send y. Write down the Receive algorithm for this scheme.

4. Suppose we build an "authenticate-then-encrypt" scheme using our strange encryption algorithm from above, a normal secure MAC like HMAC. Present an attack that proves that this scheme does not satisfy our definition of secure channels.

5. Explain why the two attacks you discovered suggest that we should not use encrypt-and-authenticate or authenticate-and-encrypt for secure channels.

6. Our definition of secure channel is missing an important component – robustness to replay attacks. That is, an attacker the eavesdrops on the communication from Sender to Receiver can resend an old message was sent from Sender to Receiver, and the Receiver will accept it as valid.

   (a) Give an sample scenario where this might be a problem.

   (b) Explain how you would modify the encrypt-then-authenticate scheme to address this.

Exercise 4. Password cracking. Suppose you are in charge of security for a major web site, and you are considering what would happen if an attacker stole your database of usernames and passwords. You have already implemented a basic defense: instead of storing the plaintext passwords, you store their SHA-256 hashes [1].

Part A:

Your threat model assumes that the attacker can carry out 4 million SHA-256 hashes per second. His goal is to recover as many plaintext passwords as possible from the information in the stolen database. Valid passwords for your site may contain only characters a–z, A–Z, and 0–9, and are exactly 8 characters long. For the purposes of this homework, assume that each user selects a random password.

1. Given the hash of a single password, how many hours would it take for the attacker to crack a single password by brute force, on average?

2. How large a botnet would he need to crack individual hashes at an average rate of one per hour, assuming each bot can compute 4 million hashes per second?

Part B:

Based on your answer to part (a), the attacker would probably want to adopt more sophisticated techniques. You consider whether he could compute the SHA-256 hash of every valid password and create a table of (*hash*, *password*) pairs sorted by hash. With this table, he would be able to take a hash and find the corresponding password very quickly.

1. How many bytes would the table occupy?

Part C:

It appears that the attacker probably won't have enough disk space to store the exhaustive table from part (b). You consider another possibility: he could use a *rainbow table*, a space-efficient data structure for storing precomputed hash values.

A rainbow table is computed with respect to a specific set of N passwords and a hash function H (in this case, SHA-256). We construct a table by computing m *chains*, each of fixed length k and representing k passwords and their hashes. The table is constructed in such a way that only the first and last passwords in each chain need to be stored: the last password (or *endpoint*) is sufficient to recognize whether a hash value is likely to be part of the chain, and the first password is sufficient to reconstruct the rest of the chain. When long chains are used, this arrangement saves an enormous amount of space at the cost of some additional computation.

Chains are constructed using a family of *reduction functions* $R_1, R_2, \ldots, R_k$ that deterministically but pseudorandomly map every possible hash value to one of the N passwords. (We can think of each R_i as a PRF keyed with a key that the attacker chose uniformly and independently at random; that is, the key is known to the attacker.) Each chain begins with a different password p_0. To extend the chain by one step, we compute $h_i := H(p_{i-1})$ then apply the ith reduction function to arrive at the next password, $p_i = R_i(h_i)$. Thus, a chain of length 3 starting with the password `hax0r123` would consist of

$$(\ \texttt{hax0r123},\ R_1(H(\texttt{hax0r123})),\ R_2(H(R_1(H(\texttt{hax0r123}))))\ )$$

[1] You shouldn't actually use raw SHA-256 for this task, in actual practice you should use a library designed specifically for password hashing that uses a function such as scrypt or bcrypt (see `http://yorickpeterse.com/articles/use-bcrypt-fool/`). Today, GPU-based hashing is so fast that an attacker can often just compute hashes on the fly. See the link for more details.

After building the table, we can use it to quickly find a password p_* that hashes to a particular value h_*. The first step is to locate a chain containing h_* in the table; this requires, at most, about $k^2/2$ hash operations. Since h_* could fall in any of $k-1$ positions in a chain, we compute the password that would *end up* in the final chain position for each case. If we start by assuming h_* is right before the end of the chain and work backwards, the possible endpoints will be $R_k(h_*)$, $R_k(H(R_{k-1}(h_*)))$, ... We then check if any of these values is the endpoint of a chain in the table. If we find a matching endpoint, we proceed to the second step, reconstructing this chain based on its initial value. This chain is very likely to contain a password that hashes to h, though collisions in the reduction functions cause occasional false positives.

[You can read more about rainbow tables here `http://en.wikipedia.org/wiki/Rainbow_table`]

1. For simplicity, make the optimistic assumption that the attacker's rainbow table contains no collisions and each valid password is represented exactly once. Assuming each password occupies 8 bytes, give an equation for the number of bytes in the table in terms of the chain length k and the size of the password set N.

2. If $k = 5000$, how many bytes will the attacker's table occupy to represent the same passwords as in (c)?

3. Roughly how long would it take to construct the table if the attacker can add 2 million chain elements per second?

4. Compare these size and time estimates to your results from (a), (b), and (c).

Part D:

You consider making the following change to the site: instead of storing SHA-256(*password*) it will store SHA-256(*server_secret* || *password*), where *server_secret* is a randomly generated 32-bit secret stored on the server. (The same secret is used for all passwords.)

1. How does this design partially defend against rainbow table attacks?

2. Briefly, how could you adjust the design to provide even stronger protection? (Your answer should be no more than 3 sentences long.)