Homework 3: Passwords

This homework is due Thursday, February 18 at 10 p.m.. You will have a budget of five late days (24-hour periods) over the course of the semester that you can use to turn assignments in late without penalty and without needing to ask for an extension. You may use a maximum of two late days per assignment. Once your late days are used up, extensions will only be granted in extraordinary circumstances.

We encourage you to discuss the problems and your general approach with other students in the class. However, the answers you turn in must be your own original work, and you must adhere to the Code of Academic Integrity. Solutions should be submitted electronically via Canvas with the template at the end of this document.

Concisely answer the following questions. (Limit yourself to at most 80 words per subquestion.)

1. Public-key encryption. Professor Heninger has decided to use public-key cryptography with this class to report the grades from your final exams. She instructs each student to create a 2048-bit RSA key pair and post the public key \((e,N)\) publicly on Piazza. She will encrypt each student’s score \(s\) by computing \(c := s^e \mod N\), and she’ll post \(c\) to the website alongside the student’s name. The test will be out of 100 points and you cannot earn fractional points.

   (a) Assume that Professor Heninger safely receives the public key for each student. Nevertheless, how can you easily learn the score of all of your classmates?

   (b) What additional steps should the CIS331 staff take to prevent the attack found in (a)?

2. Password cracking. Since you were both clever enough to discover the flaw in Professor Heninger’s scheme and responsible enough to disclose your findings before the end of the class, you have been put in charge of security for designing a new system. Since you’re well aware of the risks of adversaries backed with lots of computational power, 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.

   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.
(a) 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?

(b) 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?

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 \( (\text{hash}, \text{password}) \) pairs sorted by hash. With this table, he would be able to take a hash and find the corresponding password very quickly.

(c) How many bytes would the table occupy?

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. Table 1 shows a simple rainbow table with \( m = 3 \) chains (columns): \( c_1, c_2, c_3 \). Each chain stores \( k = 3 \) passwords (rows): \( p_0, p_1, p_2 \) (rows).

Chains are constructed using a family of reduction functions \( R_1, R_2, \ldots, R_{k-1} \) that deterministically map each hash value to one of the \( N \) possible passwords. Each \( R_i \) should be a different pseudorandom function. 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 \( i \)th reduction function to arrive at the next password, \( p_i = R_i(h_i) \). Thus, a chain of length 3 starting with the password \( \text{hax0r123} \) would consist of \( \{ \text{hax0r123}, R_1(H(\text{hax0r123})), R_2(H(R_1(H(\text{hax0r123})))) \} \).

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. In Table 1, we only need to store \( p_0 \) and \( p_2 \) in each chain.

<table>
<thead>
<tr>
<th>( p_i )</th>
<th>( c_i )</th>
<th>( c_1 )</th>
<th>( c_2 )</th>
<th>( c_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_0 )</td>
<td>( h\text{x0r123} )</td>
<td>A3VP\text{w57s}</td>
<td>WEBCuG\text{p}n</td>
<td></td>
</tr>
<tr>
<td>( h_1 := H(p_0) )</td>
<td>0x08030...</td>
<td>0xe655...</td>
<td>0x3fb8...</td>
<td></td>
</tr>
<tr>
<td>( p_1 = R_1(h_1) )</td>
<td>xkjTCScS</td>
<td>Kr24FT6m</td>
<td>qnrnnEn6</td>
<td></td>
</tr>
<tr>
<td>( h_2 := H(p_1) )</td>
<td>0xa10c0...</td>
<td>0xdc0d...</td>
<td>0x233a...</td>
<td></td>
</tr>
<tr>
<td>( p_2 = R_2(h_2) )</td>
<td>uUVdGYvN</td>
<td>RHkDFAXc</td>
<td>fMVvMyq</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: An example rainbow table
After building the table, we can use it to quickly find a password \( p_* \) that hashes to a particular value \( h_* \). First we apply \( R_{k-1} \) to \( h_* \) and compare it to the endpoints in the table. If this value is present as an endpoint, we can reconstruct the chain from the corresponding starting point and obtain the original value. If this value is not present as an endpoint, we move backward one step in the chain and check whether \( R_{k-1}(H(R_{k-2}(h_*))) \) is an endpoint of any chain. In our example rainbow table in Table 1, we only need to compute \( R_2(h_*) \), \( R_2(H(R_1(h_*))) \) and check if these values match with any endpoints of the table. In general, the number of hash operations we need to perform is \( k(k-1)/2 \).

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.

(d) 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 \( N \).

(e) If \( k = 5000 \), how many bytes will the attacker’s table occupy to represent the same passwords as in (c)?

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

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

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.)

(h) How does this design partially defend against rainbow table attacks?

(i) Briefly, how could you adjust the design to provide even stronger protection?
**Submission Template**

Submit by uploading a txt file to Canvas. Use the template below to organize your submission. Follow the headings precisely for grading purposes. You may use LaTeX-style math syntax if you wish.

# Problem 1
1a.

1b.

# Problem 2
2a.

2b.

2c.

2d.

2e.

2f.

2g.

2h.

2i.