EECS 398 December 17, 2009
Intro to Computer Security Final Exam

Final Exam

You must work by yourself and abide by the College of Engineering Honor Code. You may consult your notes; you are also permitted to use books or Internet references, though this should not be necessary to answer the questions. For each response, you must **precisely cite any sources** that you used, other than your own notes.

We believe that the exam will take about 3 hours. Please report the total time you spent in your submission. The time reported will not affect your grade.

You must email your answers to "swolchok@eecs.umich.edu" by **Monday, December 21 at 5pm** with the subject line "EECS398 Final." Attach your solutions in a `.txt` or `.pdf` file named "*your uniqname*`_final`" with the appropriate file extension. Please contact us right away if you do not receive a confirmation email within 60 minutes. **Late exams will not be accepted.**

Answer all six of the numbered questions below. Show your work.

1. **Hashes and passwords.** 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.

   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 consist of the characters a–z, A–Z, and 0–9 and are exactly 8 characters in length. For the purposes of this exam, assume that each user selects a different, random password.

   (a) On average, how long would it take for the attacker to crack a single password by brute force? How large a botnet would he need to crack one password per hour, assuming each bot can compute 4 million hashes per second?

   (b) Based on your answer to part (a), the attacker would need 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.

   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. 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 each hash value to one of the N passwords. 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. Thus, a chain of length 3 starting with the password `hax0r` would consist of (`hax0r`, $R_1(H($`hax0r`$))$, $R_2(H(R_1(H($`hax0r`$))))$).

After building the table, we can use it to quickly find a password that hashes to a particular value h. The first step is to locate h in the table; this requires, at most, about $k^2/2$ operations. Since h could fall in any of k positions in a chain, we compute the password that would end up in the final chain position for each case, $R_k(h)$, $R_k(H(R_{k-1}(h)))$, $\ldots$, and 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 the complete 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.

(c) For simplicity, make the optimistic assumption that the attacker's rainbow table contains no collisions and each valid password is represented exactly once. Estimate the size of the table in terms of the chain length k. If $k = 5000$, how many bytes will the attacker's table occupy? Roughly estimate how long it takes to construct the table if the attacker can add 2 million chain elements per second. Compare these size and run-time estimates to your results from (a) and (b).

(d) Based on your analysis, you consider making the following change to the web 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.)

How well does this design defend against a rainbow table attack? Can you adjust the design to provide even stronger protection?

2. **Secure programming.** StackGuard is a mechanism for defending C programs against stack-based buffer overflows. It uses a *canary*, a value stored in the function's stack frame immediately before the return address. Before the function returns, it verifies that the canary value hasn't changed; if it has, the program halts with a security error.

(a) In an alternate design, suppose that a canary is placed directly after each buffer, and, as in StackGuard, it is checked just before the function returns. How effective do you think this method would be at detecting buffer overflows?

(b) Now suppose that a canary is placed directly after each *array* and that it is checked after every array access. Would this detect a buffer overflow? If so, why do you think this is not suitable for use in practice? If not, describe an attack that could change a number beyond the buffer without affecting the canary.

(c) What are the security drawbacks to choosing the canary value at compile time instead of at run time? Why do some implementations use 0 for the canary anyway?

(d) StackGuard cannot protect against overflows into function pointers. Describe two other kinds of bugs that can corrupt the stack and allow the adversary to take control when the function returns, and for which StackGuard provides no protection.

3. **Authorization**

   For each of the credentials below, describe: (i) the access right granted, (ii) the binding authority that makes the credential valid, and (iii) the authorized user of the credential.

   (a) http://www.google.com/
   (b) /usr/bin/true
   (c) A self-signed public key certificate generated by you
   (d) A dollar bill
   (e) A $50 Zingerman's gift card
   (f) Your uniqname and password
   (g) A "forever" first-class postage stamp
   (h) A VeriSign Public Primary Certification Authority private key

4. **Privacy and anonymity**

   A service lets users browse the Internet anonymously through a network of proxy servers. To avoid having their identities compromised, users must install special software that modifies the way their web browsers behave when they are using the service.

   In the absence of each change or action listed below, describe an attack that would reveal information about the user's identity. Assume that the user regularly visits web sites under adversarial control, but that the adversary is physically separated from the user.

   Changes that apply while the anonymous browsing service is enabled:

   (a) Disable the HTTP Referer header
   (b) Disable Java and Flash
   (c) Set the HTTP User-Agent header to a generic value
   (d) Resize browser windows to multiples of 35 pixels

   Actions that occur each time the user enables or disables the anonymous browsing service:

   (e) Close all open sites
   (f) Clear the history list
   (g) Clear the cookie store
   (h) Clear the browser cache

5. **Web security.** Consider the fictitious social networking site HoneyPlace, located at honeyplace.com. HoneyPlace has millions of users, not all of whom are particularly security-conscious. To protect them, all pages on the site use HTTPS.

   (a) HoneyPlace's homepage has a "Delete account" link which leads to the following web page:

```
<p>Are you sure you really want to delete your account?</p>
<form action="/deleteuser" method="post">
  <input type="hidden" name="user" value="{{username}}"></input>
  <input type="submit" value="Yes, please delete my account"></input>
</form>
```

   (The web server replaces {{username}} with the name of the currently logged-in user.)

   The implementation of /deleteuser is given by the following pseudocode:

```
if account_exists(request.query_parameters['user']):
    delete_account(request.query_parameters['user'])
    return '<p>Thanks for trying HoneyPlace!</p>'
else:
    return '<p>Sorry, an error occurred.</p>'
```

   What are the major security flaws in this design?

   (b) Suppose that /deleteuser is modified as follows:

```
if user_is_logged_in(request.query_parameters['user'],
                     request.cookies['login_cookie']):
    delete_account(request.query_parameters['user'])
    return '<p>Thanks for trying HoneyPlace!</p>'
else:
    return '<p>Sorry, an error occurred.</p>'
```

   (Assume login_cookie is unique to the user, hard to forge, and has a reasonable expiration time.)

   Does this design correct the flaws you identified in part (a)? Why or why not?

   (c) Suppose that /deleteuser is modified again:

```
# user_logged_in_from_ip checks whether the user is logged in and
# whether the login_cookie was assigned to the given IP address.
if user_logged_in_from_ip(request.query_parameters['user'],
                          request.cookies['login_cookie'],
                          request.requesting_ip_address):
    delete_account(request.query_parameters['user'])
    return '<p>Thanks for trying HoneyPlace!</p>'
else:
    return '<p>Sorry, an error occurred.</p>'
```

   Does this design correct the flaws you identified in part (a)? Why or why not?

   (d) If you said that the design in part (c) remains vulnerable, what is the simplest change that would fix the problems? If you said it corrects the flaws, can you suggest an even better solution?

6. **Ethics and law**

   Consider the following scenario: A worm is infecting systems by exploiting a bug in a popular server program. It is spreading rapidly, and systems where it is deleted quickly become reinfected. A security researcher decides to launch a counterattack in the form of a defensive worm. Whenever a break-in attempt comes from a remote host, the defensive worm detects it, heads off the break-in, and exploits the same bug to spread to the attacking host. On that host, it deletes the original worm. It then waits until that system is attacked, and the cycle repeats.

   (a) Many people would claim that launching a counterattack in this scenario is ethically unacceptable. Give at least three arguments that support this view.

   (b) Are there circumstances or conditions under which a counterattack would be ethically justified? Explain your reasoning.

   In 1988, a graduate student accidentally unleashed one of the first Internet worms. It deleted no files but caused widespread service interruptions. Cleaning it up and implementing defenses were costly. The student was convicted of violating the Computer Fraud and Abuse Act. Despite demands that he be sent to prison for the maximum time possible (to make an example of him), the judge sentenced him to pay a fine and perform community service. Today that student is a professor at MIT.

   (c) What factors do you think caused the judge to hand down this sentence? Do you believe the outcome was fair?

   (d) Would you expect a different outcome if a similar case happened today? Suppose you were the judge; what information would you take into account in making your decision?

Extra Credit

When Professor Halderman was 12 years old, he chose a password for AOL Instant Messenger. It was very insecure: a common 4-letter noun starting with the letter **g**. It wasn't "geek." What was it?

(Hint: Several people have guessed correctly on the first try based solely on the information given here.)