Assignment 6 – Due Monday, April 28

1. Consider the following authentication scheme, which is a slightly simplified version of standard HTML digest authentication. Using a secure cryptographic hash function H (meeting all security goals given in Table 11.1), if the user with user id u has password p, then the server stores $y = H(u\|p)$ in its user database. Then when a user seeks to authenticate, the server generates a random nonce n which it sends to the client system, which in turn gets a user name u and password p from the user, computes $z = H(H(u\|p)\|n)$ locally and sends the result back to the server. The server can then compute $H(y\|n)$ and see if these values match. Answer the following questions, with explanations (they're not really just yes/no questions!).

   (a) An eavesdropper sees both n and z as they are transmitted. Can the eavesdropper figure out the user's password?

   (b) An attacker breaks into the server, copies the authentication database, so learns the value y. Can she compute the user's password from this?

   (c) If an attacker obtains the server's user database, and consequently learns y, can the attacker log in to the system as user u?

2. Clearly a hash function that has the strong collision resistance property also has weak collision resistance. What about the next step down? Does a hash function that has weak collision resistance also satisfy the one-way property (see Table 11.1 on page 323 for these terms)? To answer this question, consider a hash function $H(x)$ that produces k-bit hash codes, and satisfies all three of these security properties. Now construct a hash function $H'(x)$ that produces $(k+1)$-bit hash codes as follows: If x is exactly k bits long, then output $0\|x$ (a single 0 bit followed by x); otherwise output $1\|H(x)$ (a single 1 bit followed by the H-hash code of x). Is $H'(x)$ weakly collision resistant? Is it one-way? Justify your answers!

3. With a basic understanding of hash functions, you can understand what a "proof of work" problem is, which is integral to systems like Bitcoin. Use Google to find reliable references that describe how a "proof of work" is implemented in Bitcoin, and describe this process in your own words. What cryptographic property of hash functions is most relevant for such a proof of work to provided the necessary properties?

4. ElGamal encryption was described in Section 10.2 of the textbook. In this problem, you are to explore how ElGamal measures up in terms of the formal security models we discussed.

   (a) It is impossible for "textbook RSA" to be IND-CPA secure. Why is that, and does ElGamal have the same problem?

(b) Let (C_1, C_2) be an ElGamal ciphertext, computed as shown in Figure 10.3 (page 293). Write out formulas (in terms of α, k, Y_A and M) for $\alpha \cdot C_1$ and $Y_A \cdot C_2$, where operations are preformed mod q.

(c) What result would be produced if you ran fake ElGamal ciphertext $(\alpha \cdot C_1, Y_A \cdot C_2)$ through the ElGamal decryption function?

(d) Use these observations to create an attack algorithm that wins the CCA game against ElGamal. In addition to describing the algorithms, remember to analyze the advantage of your adversary in this game.

As an aside, the message you should learn from this problem is that textbook RSA cannot be IND-CPA secure, much less IND-CCA secure. ElGamal in fact does turn out to be IND-CPA secure, but it is not IND-CCA secure.

5. This is a programming problem. The standard way to compute multiplicative inverses modulo p is to use the Extended Euclidean Algorithm, as described in Chapter 4. When the modulus p is prime, you can also compute $a^{p-2} \bmod p$, which will give the multiplicative inverse of a.

   (a) Why is $a^{p-2} \bmod p$ the multiplicative inverse of a? The reason is pretty simple, so don't give a long convoluted argument — a single, unambiguous sentence is all you need!

   (b) Implement both of these methods for large (1000+ bit) integers. Note that if you are careful about your choice of programming language, you can use a built-in function for the modular powering so that this second implementation would be a single line!

   Once you are sure your implementations work correctly, time them on a 1024-bit modulus. To do this, you'll need a 1024-bit prime — the best way for you to do this would be to find a way to generate your own large random prime, either by typing up the Miller-Rabin primality testing algorithm, or using any of the mathematics packages that have this functionality built in. If you really can't generate your own large random prime, you can use the one linked from this assignment on the class web page (you'll still have to get it into your program!). Note that your programs should be too fast to accurately time directly without using some advanced profiling tools. The easiest way around this is to iterate your test as many times as necessary so that the overall time is around 30 seconds. If you iterate n times, just divide your final time by n and you've got the time for one modular inverse calculation.

6. This question is based on "Case Study 2" (Analysis of an electronic voting system) that is available in the "Readings" section of the class web page.

   (a) Novices often underestimate how important good randomization is to security. Describe two different aspects in which algorithms/protocols in the studied electronic voting system were either deterministic or used poor randomization, leading to security vulnerabilities. For each one, describe what the software did, what vulnerability this leads to (give an actual attack!), and how to correct the problem.

   (b) The key used for DES, as described in Section 4.4, is hard-coded in the software, which is terrible. Furthermore, while the key looks like it should be a densely-code hexadecimal

value, it is really a string of ASCII characters. If you didn't have access to the software, but wanted to brute force this key, how difficult would it be? To answer this question, ...

  - If the key is a string of ASCII characters, where each character is either a digit (0-9), an uppercase letter in the range A-F, or a lowercase letter 'h', what is the size of the keyspace? DES brute-forcing software written for GPUs can test a little over a billion keys/second — how much time would it take to brute force a key from this keyspace?
  - Repeat the preceding calculations (size of keyspace and time to brute force) if they keys were less restricted: characters can be any uppercase or lowercase letter in addition to any digit.

- The authors suggest using a voter-verified paper audit trail (this does exist on all electronic voting machines that I have used in North Carolina). Lets say that there are 10,000 electronic voting machines in North Carolina (this is a number I just made up, and it's probably way off the mark) — if an adversary decided that they needed to tamper with 10% of the voting machines in order to have a significant chance of affecting the outcome of the election, and an audit of the paper-trail would reveal any malicious activity, how many randomly chosen machines should be audited so that the probability of detecting the fraud is over 95%?