Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ronald L. Rivest

Handout 6
April 11, 2016
Due: April 25, 2016

Problem Set 5

This problem set is due on *Monday, April 25, 2016* at **11:59 PM**. Please note our late submission penalty policy in the course information handout. Please submit your problem set, in PDF format, on Gradescope. *Each problem should be in a separate PDF.* Have **one and only one group member** submit the finished problem writeups. Please title each PDF with the Kerberos of your group members as well as the problem set number and problem number (i.e. *kerberos1_kerberos2_kerberos3_pset5_problem1.pdf*).

You are to work on this problem set with groups of your choosing of size three or four. If you need help finding a group, try posting on Piazza or email `6.857-tas@mit.edu`. You don't have to tell us your group members, just make sure you indicate them on Gradescope. Be sure that all group members can explain the solutions. See Handout 1 (*Course Information*) for our policy on collaboration.

Homework must be submitted electronically! Each problem answer must be provided as a separate pdf. Mark the top of each page with your group member names, the course number (6.857), the problem set number and question, and the date. We have provided templates for LaTeX and Microsoft Word on the course website (see the *Resources* page).

Grading: All problems are worth 10 points.

With the authors' permission, we may distribute our favorite solution to each problem as the "official" solution—this is your chance to become famous! If you do not wish for your homework to be used as an official solution, or if you wish that it only be used anonymously, please note this in your profile on your homework submission.

Our department is collecting statistics on how much time students are spending on psets, etc. For each problem, please give your estimate of the number of person-hours your team spent on that problem.

Problem 5-1. Zero-Knowledge Proofs-Of-Knowledge (ZKPOK)

In class, we have seen Shnorr's ZKPOK for Discrete Log (DL) problem. In this problem, we will explore a similar ZKPOK for RSA problem.

Let $N = pq$ and e such that $gcd(e, \phi(N)) = 1$. Let $m \in \mathbb{Z}_N^*$ and $c = m^e \pmod N$. Consider the following protocol for proving knowledge of m given (N, e, c):

1. P picks $k \in \mathbb{Z}_N^*$ randomly and sends $t = k^e \pmod N$ to V.
2. V picks integer $r \in [1, B]$ randomly and sends r to P, where B is some fixed value.
3. P computes $w = m^r \cdot k \pmod N$ and sends w to V.
4. V accepts if and only if $w^e = c^r \cdot t \pmod N$.

Argue that the protocol above is an honest-verifier ZKPOK. Remember to argue completeness, soundness, honest-verifier zero-knowledge and to demonstrate an extractor for proofs-of-knowledge.

Problem 5-2. Aggregate signatures Consider the following setting:

We have a standard "gap group" setup. Let G_1 and G_2 be cyclic groups such that $|G_1| = |G_2| = p$ where p is prime. Let g_1 be a generator of G_1. Let $e : G_1^2 \to G_2$ be a function such that:

- Bilinearity condition: $e(P^a, Q^b) = e(P, Q)^{ab}$ for all P, Q in G_1 and a, b in Z_p^*.
- Non-degeneracy condition: Let g_1 be a generator of G_1. Then $e(g_1, g_1)$ is generator of G_2.

An aggregate signature scheme is a digital signature that supports aggregation : Given n signatures on n *distinct* messages m_i from n distinct users u_i ($1 \leq i \leq n$), it is possible to aggregate all these signatures into a single short signature. This single signature and the n original messages (plus access to their public keys) suffices to convince the verifier that user i indeed signed message m_i.

The protocol is as follows:

- Let U be a set of n users and $H : \{0,1\}^* \to G_1$ be a public hash function from the set of all messages to G_1. Let $x_i \in Z_p^*$ be the private key of user i and $v_i = g_1^{x_i}$ be the corresponding public key. There is a public directory holding all of the (i, v_i) pairs. (Note that we are using multiplicative notation for G_1 and G_2.)
- User u_i in U signs message m_i in $\{0,1\}^*$ to generate signature $\sigma_i = H(m_i)^{x_i}$. The messages m_i must be all distinct. The aggregate signature is $\sigma = \sigma_1\sigma_2 \cdots \sigma_n$.
- The verifier checks that $e(g_1, \sigma) = \Pi_{i=1}^{n} e(v_i, H(m_i))$

(a) Argue that for a function e satisfying the conditions mentioned above it is true that $e(PQ, R) = e(P, R)e(Q, R)$ for all P, Q, R in G_1.

(b) Argue that the proposed verification step works as intended.

(c) What is a simple trick you could do to ensure all messages are unique?

Problem 5-3. Colliding Commits

We saw the importance of code signing in the Apple vs. FBI case. One method of authenticating programs (as an alternative to signing the compiled binary) is to verify the signature of each git commit on the path to the currently checked-out code, and then build the binary from source.

Assume a commit struct has the following layout:

```
Commit {
  string parent_hash, // The previous commit's hash
  string code_diff,
  string author,
}
```

(a) In his first implementation, Bob uses the following hash function for signing and verifying a commit object:

$$H_0(c) = \mathsf{SHA256}(c.\mathsf{parent\_hash} + c.\mathsf{code\_diff} + c.\mathsf{author})$$

where + denotes string concatenation

Unfortunately, there are collisions in Bob's hash function! Find an example of two commits that have the same H_0 evaluation, where the second commit has an empty code_diff. Submit them to the /H0/collision server endpoint. Make sure to set the author of the first commit to be your team name (a comma-separated list of team usernames).

(b) One of the 6.857 students responsibly disclosed Bob's vulnerability, causing Bob to come up with a new hash function for commits:

$$H_1(c) = \mathsf{SHA256}(c.\mathsf{parent\_hash} + \text{“|”} + c.\mathsf{code\_diff} + \text{“|”} + c.\mathsf{author})$$

In placing a pipe character between each value, Bob feels that he has thwarted all possible collisions. Find any two commits that collide and submit them to the /H1/collision server endpoint. Make sure to set the author of the first commit to be your team name (a comma-separated list of team usernames).

(c) Infuriated by his imperfect hash function, Bob sets out to find true collision resistance and comes up with a final hash function:

$$H_2(c) = \mathsf{SHA256}(\mathsf{SHA256}(c.\mathsf{parent\_hash}) + \mathsf{SHA256}(c.\mathsf{code\_diff}) + \mathsf{SHA256}(c.\mathsf{author}))$$

where + denotes concatenation of the hexadecimal-encoded strings of SHA256 output

Argue briefly why H_2 is collision resistant, assuming SHA256 can be modeled as a random oracle.

(d) In this part, we will use Schnorr signatures to sign and verify commits (using H_2 as a hash function). Note that in this problem we treat SHA256 output as hexadecimal-encoded (including the inner calls in H_2).

The Schnorr signature scheme should look similar to ElGamal encryption. Refer to `https://en.wikipedia.org/wiki/Schnorr_signature` and `https://en.wikipedia.org/wiki/Schnorr_group` for a briefing on these primitives. We will work in the group $\mathcal{Z}^*_{p=qr+1}$. The signed message (referred to as M on Wikipedia) should be $H_2(c)$ (which will be a hex encoded output), while H should be computed as SHA256 whose output is interpreted as a big-endian integer (not hexadecimal encoded). r should be converted to hexadecimal (with no leading 0x or trailing L) before appending to $H_2(c)$. Most of these encoded details are already handled for you in the provided schnorr.py.

Generate a Schnorr signing key over the provided Schnorr group ($\mathcal{Z}^*_p$) and sign a commit using H_2. Submit your verification key, signature, and commit to the /H2/signature server endpoint. Make sure to set the author of the commit to be your team name (a comma-separated list of team usernames).

The server APIs and Schnorr parameters are set up for you in the provided schnorr.py.