Programming Assignment 1 Winter 2014

CS 255: Intro to Cryptography

Prof. Dan Boneh Due **Jan. 27 11:59pm**

1 Introduction

In this assignment, you will be writing code that implements efficient S/KEY authentication in Javascript, as discussed in the lecture. We will give you starter code with a naive example and some test code, but the implementation will be largely up to you.

2 S/KEY

You can review S/KEY by watching the video from the first day of class. Discussion of S/KEY starts at 1:09:27, and the videos are available at `https://mvideox.stanford.edu/Graduate/Course/Details/86` . Wikipedia also has a good discussion at `https://en.wikipedia.org/wiki/S/KEY` .

For this assignment, you must find an algorithm that can implement S/KEY for hash chains of length n using client-side storage space $O(\log n)$ and amortized $O(\log n)$ time per authentication[1]. Try solving this on your own, but you may come to office hours if you'd like to discuss the problem. If you get stuck you may also try reading Markus Jakobsson's paper referenced below.

Hint: suppose the current hash chain has length k. The hash value at the end of the chain (i.e. at position k) is the next one-time password to use for authentication. Try storing the hash values of $\lceil \log_2 k \rceil + 1$ positions (called pebbles) along the chain so that one of the pebbles is at position k. Once this one-time password is used for authentication show how to update the position of the pebbles so that one of the pebbles will be at position $k - 1$ and therefore holds the next one-time password to use. The update should be done by computing at most an $O(\log n)$ number of hashes (amortized). Next, update the pebbles so that one of them is at position $k - 2$, and so on. To get you started, when the length of the chain is $k = 2^{20}$ the 21 pebbles should be placed at positions $k + 1 - 2^i$ along the chain for $i = 0, 1, \ldots, 20$.

For a challenge (no extra credit), you may simultaneously like to attempt *guaranteed* logarithmic time per authentication (i.e. no amortization). For this, you may consult the paper "Fractal Hash Sequence Representation and Traversal" by Markus Jakobsson.

3 Requirements

The starter code provides a `naive_chain` implementation of S/KEY. This naïve implementation has the correct functionality, but authentication is very slow.

[1] Amortized $O(\log n)$ time means that the *average* time per authentication is $O(\log n)$. Some (few) authentications may require $O(n)$ time, but the average time over all n authentications should be $O(\log n)$.

Your task is to program an alternative `pebble_chain` that:

- implements S/KEY using the functions `initialize()` and, `advance()`,
- can save its state to and from a string using `save()` and `load()`, and
- uses logarithmic storage space and (amortized) logarithmic time per call to `advance()`.

You should also respond to the following questions at the top of `skey.js`:

- Briefly describe your implementation and its design choices. (e.g. What algorithm did you use? How did you structure your code? Did you do something interesting in `save`/`load`? If it's not obvious, justify the space/time used by your implementation.)
- If you were designing an authentication mechanism for a hot new startup that wants to protect its users, how would you decide whether/where to use S/KEY?
- (Will not affect your grade:) How long did you spend on this project?
- (Optional:) Do you have any comments or suggestions for improving the assignment?

All deliverables are due in `skey.js`, although you may include additional files. Do not modify `lib/sjcl.js`, and make sure that the version of `test.js` from the starter code runs successfully on your submission.

Here are descriptions of the functions. See section 4.1 for information about `hash()` and `bitArray`.

3.1 `pebble_chain(num_iterations)`

- Returns: Javascript object

This is essentially a constructor. It returns a Javascript object that has at least four callable functions: `initialize()`, `advance()`, `save()`, and `load()`.

You should not need to modify the main structure of `pebble_chain`, but you may add more functions or restructure it as long as the four required functions work correctly.

3.2 `pebble_chain.initialize(num_iterations, seed)`

- `num_iterations`: integer (must be a positive power of 2)
- `seed`: string or `bitArray`
- Returns: `bitArray`

This takes in a given value `seed` and hashes it once to produce the start of the chain (which will eventually be the last non-`null` value that `advance()` will output).

Then, it prepares an authentication chain of length `num_iterations`. It returns the initial value, which is the result of hashing `seed` successively (`1 + num_iterations` times).

This function should ”walk the chain” only once, i.e. it should only make about `num_iterations` calls to `hash()`.

3.3 `pebble_chain.advance()`

- Returns: `bitArray` or `null`

The first time, this returns the value that hashes to the output from `initialize()`. After that, each output is the value that hashes to the previous output. After this function has been called `num_iterations` times, the output should be `null` every time.

The function must take amortized logarithmic time in the number of iterations. To explain this, suppose that an implementation of `advance()` calls `hash()` a total of $h[i]$ times on iteration i. Then the amortized running time is defined as:

$$\frac{1}{\texttt{num\_iterations}} \sum_{i=1}^{\texttt{num\_iterations}} h[i]$$

For the naïve implementation, $h[i] = \texttt{num\_iterations} - i$ from which it follows that the amortized running time is $(\texttt{num\_iterations} + 1)/2$, which increases linearly with the length of the chain. For your implementation, the amortized running time must be no more than $O(\log_2(\texttt{num\_iterations}))$.

3.4 `pebble_chain.save()`

- Returns: string

Serializes the state of the chain into a string. It should be possible to shut down the program and load the stat using `load` using just the data saved by this string.

Additionally, the string should be reasonably short (linear in `log_num_iterations`).

3.5 `pebble_chain.load(str_data)`

- `str_data`: string
- (No return value.)

Loads a chain from the serialized string. Note that is an *alternative* to `initialize()`, i.e. you can assume that only one of `initialize()` or `load()` will be called.

This process should be very fast – it should not compute any hashes, and it *definitely* should not compute the chain from scratch.

3.6 Example Usage

To give you an idea of how the functions work, here is how a typical use would look:

```
var chain = pebble_chain();
var initial = chain.initialize(log_num_iterations, some_bit_array);

var next = chain.advance();
next = chain.advance();
// ...

var saved_string = chain.save()
var new_chain = pebble_chain();
new_chain.load(saved_string)

next = chain.advance();
next = chain.advance();
//...
```

4 Running and Writing the Code

The easiest way to run the project on the commandline is using `node.js`. Node.js is available on the `corn` clusters. You may also visit `http://nodejs.org` and install node.js on your computer locally, and you can ask on Piazza if you're having trouble installing it.

On a computer with `node.js` installed, run the following from the project directory:

```
node test.js
```

This will run a basic set of tests on `naive_chain` and `pebble_chain` with `log_num_iterations == 4`. The `pebble_chain` tests will crash or fail until your implementation is correct. You'll probably want to start by copying code from `naive_chain` to `pebble_chain` and adjusting it to your needs.

Note that we may run more exacting tests than those in `test.js`. If you took a reasonable approach, your submission should be able to pass these easily. As a rule of thumb, your submitted code for `pebble_chain` should be able to traverse the entire chain for `log_num_iterations == 16` in at most a few seconds.

4.1 The `hash` function and `bitArray`

The starter code is plain Javascript, but it uses a custom `hash()` function provided at the top of the file. Its output is a `bitArray`. You should only make the following assumptions about it:

- The output of `hash()` is a `bitArray`.
- A `bitArray` can be passed in to `hash()` again.
- A `bitArray` can be printed using the `hex()` convenience function.
- Two `bitArrays` can be compared using the `is_equal()` convenience function.

In actuality, the hash function taken from the Stanford Javascript Crypto Library (SJCL), and you may read up on the documentation for `bitArray` at `http://bitwiseshiftleft.github.io/sjcl/doc/symbols/sjcl.bitArray.html` . However, your code should continue to work even if we replace `hash()`, `hex()`, and `is_equal()` with a new set of compatible functions.

5 Questions?

If you have general questions, please post on them on Piazza at `https://piazza.com/class/hou9vrkx5ls5rw` so that all students can benefit from the answer. If you have a problem with your specific implementation, come to office hours or ask a private Piazza question.