CS255: Cryptography and Computer Security **Winter 2016**

Project #1

Due: Friday, Feb. 12, 2016

1 Introduction

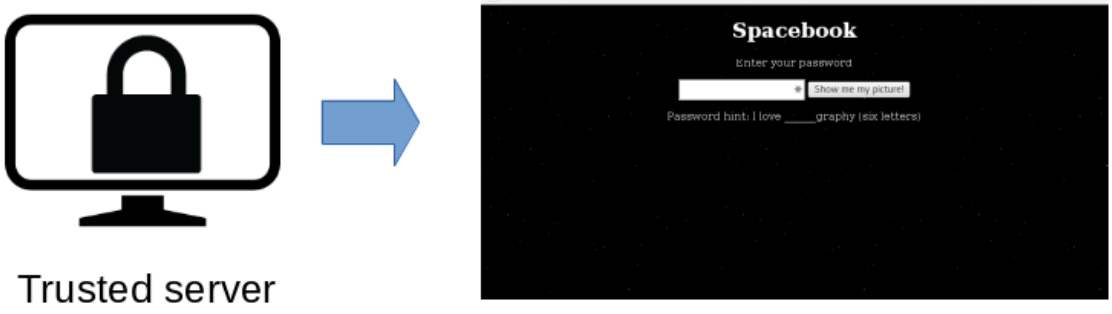
February 1, 2046: Your friends all laughed at you when you left your seven-figure job as a cryptographer to found Spacebook, the social network for astronauts. But now that Elon Musk's Mars colony has grown to over one million inhabitants, space travel has gone mainstream and your social network is the hottest website on the information superhighway. However, you have a problem. In order to enable quick access to your website, you have had to co-locate servers in various data centers throughout the solar system, not all of which you control or trust.

To manage this problem, you serve the Spacebook homepage only from data centers you trust. The homepage is small and rarely changes, so it can be downloaded quickly and cached by the browser. The homepage contains Javascript that downloads encrypted user-specific data, such as photos of the user's friends, from a CDN. This CDN, or content delivery network, consists of servers in untrusted data centers, which is one reason it is important that the user's data be encrypted. This data is then decrypted in the browser using Javascript and displayed for the user.

In this assignment, you will write the code that verifies and decrypts the user's data. Your implementation will use authenticated encryption (AES using Galois/Counter Mode), a SHA-256 Merkle tree, and a new password hashing function called Balloon. We have provided a library that exposes the hashing and decryption functions you will use, as well as a starter homepage and some content on a "CDN."

Here's what content gets loaded when, and from where.

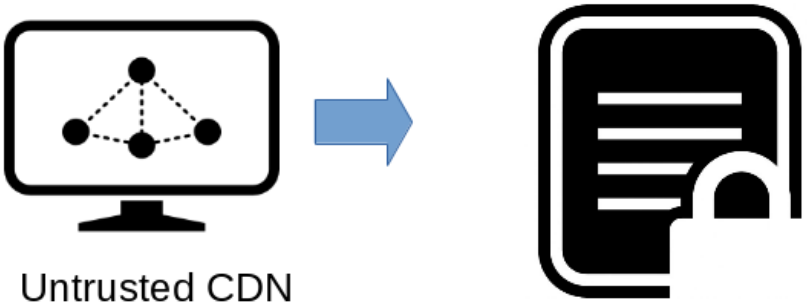
(1) Browser downloads HTML and JS from trusted server



Trusted server

---

(2) JS running in the browser downloads encrypted payload from CDN

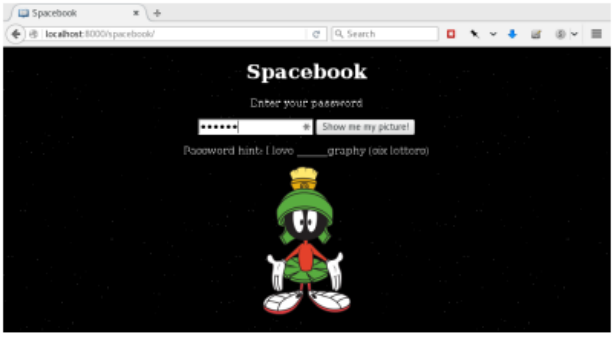


Untrusted CDN

Encrypted payload

---

(3) JS running in the browser verifies Merkle tree, decrypts, and displays payload



2 Running the assignment

This assignment has been tested in Chrome and Firefox. If you open Spacebook's `index.html` file directly in the browser using a `file://` URL, the starter code will not work (due to CORS restrictions on `file://` URLs). Instead, **you should serve the files using a web server**.

One easy way to serve the webpage locally is to open a Terminal, change into the `proj1` directory, and run `python --version`. If you see "Python 2.x.x", run `python -m SimpleHTTPServer`. If you see "Python 3.x.x", run `python -m http.server`. Then visit `http://localhost:8000/spacebook` in Chrome or Firefox. If you don't have Python, you can try Mongoose, a small stand-alone web server.

You can also use Stanford's shared computing resources to host your assignment. Just be sure not to make your code available to others on the Internet. We have made it easy to protect your code using WebAuth, Stanford's authentication system. Follow these instructions to go that route.

1. `ssh` *`yourSUNet`*`@corn.stanford.edu`
2. `cd WWW`
3. `wget https://crypto.stanford.edu/~dabo/cs255/hw_and_proj/proj1.zip`
4. `unzip proj1.zip`
5. In a text editor, open `proj1/spacebook/.htaccess` and change `dabo` to your SUNet ID.
6. In Chrome or Firefox, visit `https://www.stanford.edu/~`*`yourSUNet`*`/proj1/spacebook`.
7. You may have to log in using your SUNet ID. This is to prevent other students from viewing your solution.

If you need help setting up the assignment, please post on Piazza.

3 FAQ and summary of requirements

- Your code should use the given Merkle path to compute the root of the Merkle tree containing the encrypted data, and verify that it matches the given Merkle root.
- You can read about Merkle trees on Wikipedia. The Merkle tree we have constructed uses the SHA-256 hash function.
- Your code should use Balloon hashing to derive a key from the user-provided password and the salt given in the starter code.
- The correct password is **crypto**, but you should not embed the plain-text password in your code (e.g., no `if (password === "crypto")`).
- You must import your key and use it to decrypt and display the provided data.
- `balloonHash` outputs 512 bits while `importKey` only takes 256 bits. You should only use the first 256 bits of `balloonHash`'s output.
- Your code should only call `balloonHash` once.

4 Submitting

You will submit your assignment by uploading the proj1 folder to AFS and running the submit script. This should look something like the following.

1. (to upload to corn): `scp -r proj1 yourSUNet@corn.stanford.edu:~`
2. `ssh yourSUNet@corn.stanford.edu`
3. `cd proj1` or `cd WWW/proj1`
4. `/usr/class/cs255/bin/submit`

5 Implementation details

We have provided you with the Spacebook homepage (`index.html`), the page-specific Javascript (`spacebook.js`), a small cryptography library (`lib.js`), the Balloon hashing code (`balloon.js`), and the encrypted payload from the CDN (`data.enc`). You should only need to look at `spacebook.js` and `lib.js`, and should only need to edit `spacebook.js`.

The data you will decrypt is asynchronously loaded once the homepage opens, and is stored in the variable `data`. The variables `salt`, `iv`, `merkle_root`, and `merkle_tree` will also be useful to you. `salt` contains the salt used by Balloon to derive the password, `iv` contains the IV used to encrypt the payload, and `merkle_root` and `merkle_tree` contain nodes of the Merkle tree, described below. We have made these variables global so that they will be accessible from the developer console and hopefully make debugging easier.

You will need to check the integrity of the encrypted data using the Merkle tree, decrypt it, and display it for the user.

5.1 Asynchronous Javascript

The Web Cryptography API on which our cryptography wrapper library is based is asynchronous. Rather than return hashes, keys, or data directly, the functions return promises. Each promise "resolves" to a value, and once the promise is finished, it will call a function of your choosing and pass it that value. For example, if we have the promise `waitAndReturn4`, then assuming it does what its name suggests, writing

```
waitAndReturn4.then(function(num) {console.log(num);});
```

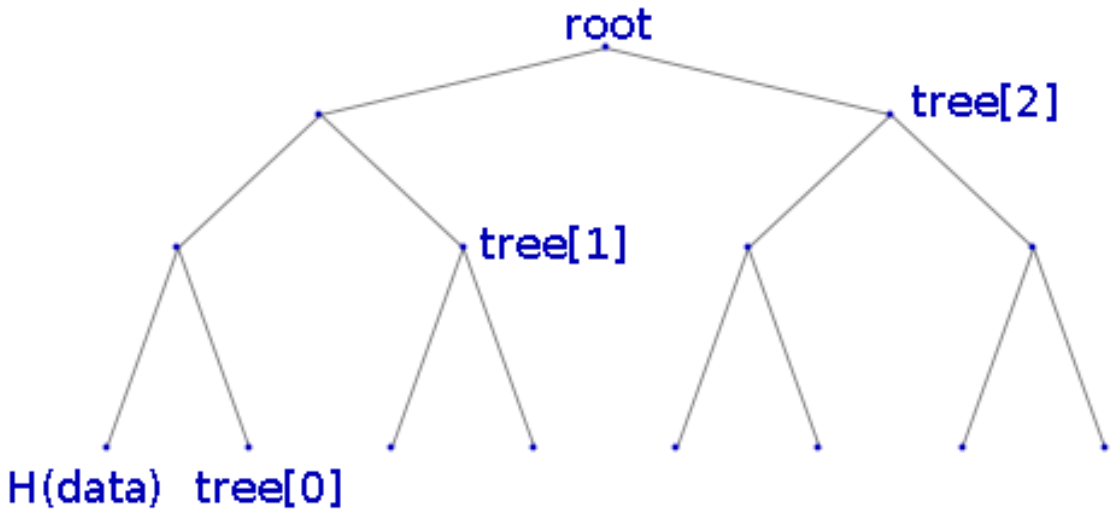
will eventually print "4" to the console.

5.2 `spacebook.js`

The Javascript starter code contains a `passwordEntered()` function that is called when the user enters a password. It also contains a `displayImage()` function which takes as input an ArrayBuffer containing an image and displays it on the page. Once you have decrypted the user's data, you should pass the resulting ArrayBuffer to `displayImage()` to display it.

5.3 Merkle tree

The Merkle tree was generated by hashing the latest version of every user's data together. Using the Merkle tree, we not only guarantee the integrity of our encrypted data (which authenticated encryption does for us anyway), but also ensure that we have the latest version of our data. Otherwise, a malicious or malfunctioning CDN could send us an old version of our data. For simplicity, you may assume that the number of encrypted content items is fixed at eight. The Merkle tree looks like the following image, with "tree" and "root" correspoding to the variables `merkle_tree` and `merkle_root`.



The tree was generated by taking the SHA-256 hash of each user's encrypted data, then concatenating each pair of hashes and hashing that, and repeating this process until reaching the root of the tree.

5.4 Key generation

The key was generated using the password "crypto". You will need to derive a key from the password using Balloon hashing, then import the key to a format the library recognizes and can use for decryption. The password entered by the user is stored in the variable `password` inside the `passwordEntered()` function.

5.5 Error handling

It's up to you exactly how to handle errors that might arise when this code is executed, but if the Merkle tree integrity check fails you should not attempt to decrypt the data.

6 API description

6.1 `lib.balloonHash(password, salt)`

This function computes the Balloon password hashing function on the given password and salt, using reasonable time and space cost parameters. It is very slow! It will output progress updates

in the development console as it runs.

The function returns a promise that will resolve to a 512-bit hash, contained in an ArrayBuffer.

6.2 `lib.importKey(raw_key)`

The hash returned by balloonHash is a raw 512-bit key, and must be passed through the importKey function to convert it to a CryptoKey object, which can be used by the decrypt function.

The function takes a 256-bit raw key in an ArrayBuffer and returns a promise that will resolve to a CryptoKey object containing the 256-bit key.

6.3 `lib.decrypt(key, data, iv)`

This function decrypts the given data using the given key and IV.

The function returns a promise that will resolve to the decrypted data.

6.4 `lib.sha256Hash(data)`

This function takes in an ArrayBuffer and hashes it using the SHA-256 algorithm.

The function returns a promise that will resolve to the SHA-256 of the data, stored as an ArrayBuffer.

6.5 `lib.hexToArrayBuffer(hex)`

Given a string containing hex data (e.g. "abcd25ff"), this function will return an ArrayBuffer containing the same data.

6.6 `lib.arrayBufferToHex(buf)`

Given an ArrayBuffer, this function will return the bytes in the buffer represented as a hexadecimal string.