Resource: Storing User Passwords Securely

**This is an update of a technical post I wrote for a tech company (Rangle). It is completely optional, but it discusses proper ways that companies should store user passwords in 2018:**

Cybersecurity got ever more attention in 2017 as all industries realized how important it is to have strong protection of their computer systems. At the top of the list of security issues are passwords. Many people believe it is the weakest link in internet security ([some believe it is permanently broken](https://www.wired.com/2012/11/ff-mat-honan-password-hacker/)).

In 2016, Yahoo admitted to having [over 1 billion account credentials stolen](http://www.businessinsider.com/yahoo-data-breach-billion-accounts-2016-12), and LinkedIn announced [117 million accounts had been compromised](https://techcrunch.com/2016/05/18/117-million-linkedin-emails-and-passwords-from-a-2012-hack-just-got-posted-online/) in previous years due to weak password management on their part.

From established multinational firms to early stage startups, everybody has an obligation to store user credentials securely. Here at Rangle, we wanted to show our recommendations for storing user passwords and improving security of your applications in 2017. This tutorial will cover the user experience from the front end, all the way to the database. Let’s get started.

Step 1:

It all begins when you ask your user to create an account by setting a username and password. This is where user experience will need to play nicely with proper security measures. United States National Institute for Standards and Technology ([NIST](https://www.nist.gov/)) is currently finalizing their recommendations for password management. Some of these include:

1. Your minimum password length should be at least 8 characters, and the maximum as large are 64 characters for the user. The longer the password that the user sets, the better. This will also play nicely if said user is using password managers like [1Password](https://1password.com/)since their generated password will be long and fit this criteria. There is nothing worse than the password limit being too small for a password generated by 1Password.
2. Do accept both ASCII and UNICODE characters and don't set rules about which characters should and shouldn’t be included (P@ssword1 is NOT a good password). Instead, encourage people to set long passwords with high entropy (upper case letters, lower case letters, digits, special characters).
3. Don’t allow password hints.
4. Avoid security questions. Although this is good for an extra layer of security, the information can be easily discovered by an attacker in this day and age. It’s just an extra implementation step; a step that a user has to take while offering very little added security. Yahoo had this information stolen-- which was saved in plain text, allowing the attacker to easily see this information about each user.
5. Use 2FA (2 factor authentication) if you want an extra layer of security in your application, but avoid using SMS as this can be easily hacked to have the attackers phone receive the confirmation code.

Once the user has registered, there are two things to keep in mind:

1. Don’t let the user know the password was incorrect when logging in. Instead, mention that the username and password combination is incorrect. You want to minimize information that an attacker can get, and the less they know (such as that the user exists), the better.
2. The only time passwords should be reset by an administrator is when they suspect that an account has been compromised. Otherwise, passwords should only be reset by a user when he or she has forgotten their credentials.

Step 2:

Now that the user has entered a good password, they will submit the form and trust you to take care of their credentials. In the second step, you are transmitting the form (account sign up) data in a POST body to your server. You do it this way for a few reasons:

POST requests are never cached  
POST request will not remain in the browser history  
No restriction on data length

Most importantly, use HTTPS so that this data will be encrypted and cannot be attacked by someone who is observing your network traffic. HTTPS is the regular HTTP protocol with SSL/TLS encryption which means that only the server can read what you send it, and only you can read what the server sends back. To use HTTPS, simply purchase an SSL certificate and follow the steps in this [article](https://www.keycdn.com/blog/http-to-https/).

Step 3:

Now that we have received the username and password to our server it is time to do a few things. Before we get started though, always remember to never store passwords in plaintext. Instead, you will want to use some of these popular npm packages to hash the password: [argon2](https://www.npmjs.com/package/argon2), [scrypt](https://www.npmjs.com/package/scrypt" \t "_blank), or [bcrypt](https://www.npmjs.com/package/bcrypt" \t "_blank). We prefer bcrypt for three reasons:  1. bcrypt is 15 years old and has been vetted by the crypto community. Although argon2 [won last year’s password hashing competition](https://news.ycombinator.com/item?id=10493721), it is still fairly new and we would like to see it have a longer lifespan in the crypto community.  2. scrypt is 7 years old and also a good choice, but bcrypt is a bit easier to implement in a node.js server with simpler documentation as you will see below in the example.  3. The bcrypt npm package is better over other bcrypt implementations available on npm since it is native, highly popular, and vetted by the community without trying to reinvent the wheel.

Now you might be asking yourself, why not just use a hashing function like SHA256, add a salt (randomly generated bytes to place in front of the password) for each user, and store those in a database? The problem with computing power increasing is that attackers can now use GPUs to try out passwords at over 100 million per second and see if they get a hit. That’s why you want to use hash functions that were specifically designed to be slow. Although it is fast enough so the user won’t notice (about 100ms), it is long enough to make it infeasible for an attacker to try out a long list of passwords. bcrypt allows you to add a saltRound (10 is the recommended value) which iterates 2^10, or 1024 times over the password in a process called [key stretching](https://en.wikipedia.org/wiki/Key_stretching). Finally, bcrypt implementation is also safe from [timing attacks](http://security.stackexchange.com/questions/94577/string-comparison-timing-attack-in-plain-english). Here is a simple overview of how to use bcrypt for password management. (you can check out their [repo](https://github.com/kelektiv/node.bcrypt.js) for more detail):

1. $ npm install bcrypt
2. /\*
3. \* You can copy and run the code below to play around with bcrypt
4. \* However this is for demonstration purposes only. Use these concepts
5. \* to adapt to your own project needs.
6. \*/
8. import bcrypt from'bcrypt'
9. const saltRounds = 10 // increase this if you want more iterations
10. const userPassword = 'supersecretpassword'
11. const randomPassword = 'fakepassword'
13. const storeUserPassword = (password, salt) =>
14. bcrypt.hash(password, salt).then(storeHashInDatabase)
16. const storeHashInDatabase = (hash) => {
17. // Store the hash in your password DB
18. return hash // For now we are returning the hash for testing at the bottom
19. }
21. // Returns true if user password is correct, returns false otherwise
22. const checkUserPassword = (enteredPassword, storedPasswordHash) =>
23. bcrypt.compare(enteredPassword, storedPasswordHash)

26. // This is for demonstration purposes only.
27. storeUserPassword(userPassword, saltRounds)
28. .then(hash =>
29. // change param userPassword to randomPassword to get false
30. checkUserPassword(userPassword, hash)
31. )
32. .then(console.log)
33. .catch(console.error)

That’s it! You can now rest assured that you have stored the user password securely. The interesting thing with security is that it is always a cat and mouse game. Security advice changes with the time as new exploits are discovered and computing power increases. New attacks will be discovered but so will new solutions. Remember, the best security for your application means constantly keeping up with vulnerabilities, software updates, and new tools. For now, you can breathe easy…