Additional Security by Hashing Passwords in Browsers

Luc Gommans, [insert your name here, in no particular order yet]

Abstract

This paper discusses a proposal to hash passwords in the browser, as a security measure additional to hashing server-side. This layer will prevent leaking passwords in a variety of scenarios which are discussed in section 1. Passwords should receive additional protection client-side as long as there are users that re-use passwords on multiple websites, or users that remember their passwords since that limits their agility in changing passwords upon compromise. Algorithms that should be used by browsers will be discussed in section 2. Section 3 discusses the HTML interface that should be provided to developers. In section 4 we identify a consequence of the system proposed in section 2 and exploit it to further improve security. We conclude with an overview of the proposal.

1 Threat model

This system will protects against:

- Heartbleed - Cloudflare-like shit - passive attackers, such as one after TLS termination (see NSA before Google encrypted their back-end) - active attackers that don't want to be too high-profile or don't have the skills. (me when I was 14 years old) - weak password hashing on the server

It does not protect against anyone who can change the code or has the means and motive to MITM the page and change its contents.

2 Choice of algorithms

Passwords should be hashed using a slow algorithm which accepts a salt as input to make the output unique. However, the server will not have a copy of the plaintext of the password, making it impossible to use a standard technique.

To generate the salt, we need to use a key derivation function with globally unique inputs. A combination of the username, password and website's domain are good candidates as inputs, but a website's domain is not necessarily stable. There might be multiple, for example one with `www.` included and one without, and it might change over time. Because of this, a service identifier needs to be supplied by the developers. This will be discussed further in section 3. The risks and benefits of service identifiers will be discussed in section 4.

so, a kdf, which provides the salt input to a normal password hashing algorithms. I think pbkdf2, bcrypt, scrypt and argon2 are all good candidates. Parameters should be chosen so they can work on mobile devices without a hitch (under 200ms of computational lag, I would say). For the KDF, any old (password)

hashing algorithm can be used as long as we protect against hash length extension attacks for merkle-dmgard constructions. Not sure that's even relevant here, but why the hell not.

We will probably want to discuss versioning here, since work parameters (e.g. bcrypt's) should be upgraded every few years. Not quite sure yet how to do this, and where to lay the responsibility. Browsers will be part of the responsibility of course, but we should avoid complicated upgrade schemes where the browser needs to submit the old and the new version for a long time until everyone has been upgraded.

3 HTML API

Should be easy to use and easy to upgrade. Must not be possible to misuse by developers.

Overall, I have something like this in mind:

```
<input name=MyUserName>
<input type=password name=password hash=v1 service=example.org
 username=MyUserName>
<input type=sumit>
```

The browser would then use the service, username field value, and the password value to derive a salt. The `hash=v1` part uniquely identifies an algorithm and work factor, since we want the output to be static. The version and format are sent to the server somehow, e.g. `hashed$v1$2b00042f7481c7b056c4b`.

If the output would be dynamic, e.g. the work factor is included in the data sent to the server and this factor is variable, it would require changes on the registration and login code. Now it only requires `if ($_POST[password][0:9] == 'hashed$v1$')` to distinguish between an old-style and a new-style login.

4 Unintended effects

Allowing the website to supply the service identifier, rather than deriving it from the domain or TLS certificate, allows attackers to supply the service identifier of their intended targets. For example, a phishing website which tries to obtain logins for Reddit would supply the same identifier as Reddit does.

Rather than being a risk, this can be used to improve security: duplicate service identifiers can be logged to a public ledger. Reddit could have a bot monitor their service identifier to spot impostors before the first victim was even able to enter their password.

Additionally, browsers might issue visible warnings to users, along the lines of "hey, this website uses the same identifier as [original domain], are you sure about this?". This does provide UX challenges and is a future step which we will leave out of scope in this paper.

Conclusion

Brief overview of proposed solution and why it will help.