Additional Security by Hashing Passwords in Browsers

By Luc Gommans

Draft, version 0.2.

Abstract

This paper proposes hashing passwords in the browser, as a security measure additional to hashing server-side. This 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

People often reuse passwords, or even if they use a unique password, remember the password rather than using a password manager. This has disadvantages in the event of password compromises. It is best practice to use special password hashing algorithms to prevent attackers from being able to obtain the original password. Still, there have been various events in the past where plaintext passwords were compromised due to transport layer security (TLS) being the only protection. It is often argued that the server needs to be trustworthy anyway: if the server is not trusted, it would simply serve you a web page which does not hash your password.

This paper proposes to hash passwords client-side despite the aforementioned required trust of the server. This will protect against:

- vulnerabilities such as Heartbleed[1] and Cloudbleed[2] where server memory was accidentally leaked;
- protect against passive attackers which attack websites without TLS encryption, or websites that use early TLS termination[3]; and
- weak or no password hashing on the server.

One attack it does not protect against is pass-the-hash. For this reason it is still recommended to hash passwords server-side, regardless of whether passwords are hashed client-side.

In section 4 it is described how the proposed solution can also partially protect against an untrustworthy server, for example in the even of phishing, and man-in-the-middle attacks with active adversaries (as opposed to mere passive interception).

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.

A salt needs to be unique, but is not secret nor does it matter if it can be reversed. A fast hashing algorithm can be used for this purpose. For version 1, the KDF algorithm is an HMAC of the service identifier and the username, in that order.

```
KDF = HMAC(service-identifier, username)
```

HMAC is used as defined in RFC 2104[4]. The underlying hashing algorithm to be used is SHA-256. The inputs are encoded as UTF-8.

The hashing algorithm to be used for password hashing is PBKDF2, as defined in RFC 2898[5], with an iteration count of 30 000 and derived key length of 32

[1] en.wikipedia.org/wiki/Heartbleed

[2] en.wikipedia.org/wiki/Cloudbleed

[3] Early TLS Termination is a technique where TLS traffic is decrypted by a TLS termination proxy before it reaches its destination, usually at the edge of a private network, after which it is assumed to be secure. A well known example of abuse of this scenario is the NSA's tapping of Google's private network (more info).

[4] tools.ietf.org/html/rfc2104 HMAC

bytes (64 hexadecimal characters). The iteration count is based on it taking just below 200 milliseconds on a reasonably modern mobile phone: quick enough to not be very noticeable on the vast majority of devices and, given that it is not an action that needs to be performed often, acceptable on slower devices.

2.1 Upgrading the algorithm

As computers get faster and cryptography advances, the algorithm will have to be replaced at some point in the future. Unfortunately no well-reviewed password hashing algorithm supports seamless upgrades, where the iteration count can be upgraded without requiring user input. Rather than proposing a completely new password hashing algorithm, we use an existing one (pbkdf2) and provide an upgrade scheme.

To upgrade to a new version, an option can be specified which causes the browser to send both the new and the old hash to the server, in that order. They are separated by a dollar symbol. When upgrading from version 1 to version 2, this results in the following string:

```
hashed$v2$2b00042f7481c7b056c4b$hashed$v1$b60ca610c6aa34ccb0f7
```

The option to be specified is defined in section 3.

3 HTML API

The HTML API for this feature consists of four new attributes for the `input` element with type `password`.

- `hash` is required and contains the version to be used. Currently only version 1 is valid, denoted as `v1`.
- `username-field` is required and should point to the `name` of the username field in the same form.
- `service` is required and should be set to the service for which the client is encrypting a hash. It is recommended to use your service's domain name here, e.g. `example.com`. Elsewhere in the document, this parameter is referred to as the service identifier.
- `upgrade-from` can be used to upgrade an older version. Legal values are all legal values for the `hash` attribute. When specified, the browser must send both the new and the old version, in that order.

If an unknown version is specified, the browser must issue a warning and should fall back to the nearest known version, preferring the higher value in case of a

[5] tools.ietf.org/html/rfc2898#section-5.1 PBKDF2

tie. If a required attribute is missing or if `username-field` does not point to an existing field, the browser must issue a warning and should send the string `error-hashing!` concatenated with a random alphanumeric string of eight characters.

The reason for sending a random string is to make certain that the server cannot possibly process the data correctly. If we would omit the field, languages such as PHP would issue a NOTICE-level warning (often not shown) and continue with an empty string. If we would fill the field with only a static string, users might all register and be able to sign in with *any* password, since they all turn into the same, static string.

The reason for not providing defaults for the required fields is because specifying the `hash` field indicates that the developer intends to better secure the page. If a mistake is made which would (partially) compromise this security, it is considered better to fail rather than to silently weaken the security.

This results in the following example HTML form. Note that this is the same for both registering and logging in.

```
<form method=post action="/login">
Username: <input name=MyUsername>
Password: <input type=password name=MyPassword hash=v1
 service=example.org username=MyUsername>
<input type=submit>
</form>
```

On the server side, the `MyUsername` field is unaltered. The `MyPassword` field contains a string in the following format:

```
hashed$version$hash
```

where `hashed` is literal; `version` is the value from the `hash` attribute; and `hash` is 64 hexadecimal characters. For example:

```
hashed$v1$202d6132b2d2a469607f093dfef14e0f4798956dacbffe615453e00
8d190c861
```

Browsers which do not support these attributes will not react to the specification of any of the parameters and send the password in plain text instead. This can be detected because the value submitted to the server does not begin with `hashed$v1$`.

Browsers must detect this. If an unhashed password is submitted, it should hash the password as the browser should have done and continue as normal.

Example implementations are provided in the source code repository at github.com/lgommans/browserhashing.

4 Additional effects

Allowing the website to supply the service identifier, rather than deriving it from the domain or other semi-static field, 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 ask the ledger to be notified when a duplicate is discovered, allowing them to spot impostors before the first victim was even able to enter their password.

Browsers could also issue visible warnings to users, notifying them that this website on domain X attempts to obtain their password for this other website from domain Y, and asking whether they want to continue. This does provide UX challenges and is a future step which is out of scope for this paper.

Finally, once many websites adopted this scheme, browsers could issue a warning similar to the one given on HTTP pages with a password field if a website does not use it. This forces websites to use it if they do not want to be marked as insecure, which means phishing websites must also do it. Combined with the aforementioned duplicate service identifier detection, this would be an effective measure against phishing.

Conclusion

TODO: Brief overview of proposed solution and why it will help.

Acknowledgements

Many thanks for early feedback from Khaled Nassar, Paul Cornelissen, [someone, todo] and Bart Erven.