Hardened Stateless Session Cookies
(Transcript of Discussion)

Steven J. Murdoch
University of Cambridge

Pekka Nikander: Maybe I’m missing something, but what’s the advantage of a double hashing here, or first supplying \( a \) and then \( H(\cdot) \), instead of having, for example, two different hash functions?

Reply: If the cookie was another hash of something then the server wouldn’t be able to verify whether that cookie is correct.

Pekka Nikander: Oh, OK.

Reply: So I’m considering, let’s start with the medical principle of do no harm, what is this compared to previous schemes? The thing that makes me suspicious about this is I’m sending the password in a cookie, which is probably not a good idea, but the function includes the salt which is kept secret from the database host, so I claim that it should not be possible to go from the cookie back to the password. This is important because users share passwords over multiple websites, and if the client is a victim of a cross scripting vulnerability, it shouldn’t be possible for the attacker to go back to the password and then use that to attack another site.

Pekka Nikander: So you’re saying your attacker has read access to the data?

Reply: Yes.

Bruce Christianson: So previously it was assumed that the attacker cannot see \( v \) or \( s \), is that right?

Reply: Yes, and that was the scenario that all of the previous schemes were meant to be secure under, the attacker has no access to the database. But then let’s suppose that the attacker does have access to the database, so he has \( v \) and he has \( s \), so with these he can brute-force the passwords, but that was always the case because this is how the password hashing algorithm works. But he shouldn’t be able to get from \( v \) to \( c \) unless he brute-forces the password, so if your password is strong then this scheme should be secure even if the attacker has read access to the database.

Bruce Christianson: And the \( s \) is there now simply in order to prevent cookies from being transported from one server to another?

Reply: The \( s \) is there for the same purpose as normal password salting, which is stop a brute-force table from being built up. The thing I’ve missed out from this (which is in the paper) is actually all this is MAC’d under a static secret, which I am assuming that the attacker will get access to as well: but if the attacker doesn’t have access to that key he can multiply the cookies. Another aspect of that is, there’s an expiry thing in the cookie, and if the attacker gets access to the database and a valid cookie, they can extend that until it compromises the data, but if they don’t have a cookie to start off with, they shouldn’t be able to
generate a new one. I can’t think of a way to be able to get any better security properties than that, if you can, I’d be very interested to hear.

**Feng Hao:** I’m not very sure of these a functions, so it’s just a hash?

**Reply:** Yes, you will get all the same properties if you use a various small hash options. I suggest that it be implemented is based on the `phpass` algorithm, which is, you take the hash of the salt and the password, and then take the hash of that and the password, and a hash of that and the password, and then do that 256 times. This is because you want the attacker to work as hard as possible, if they have access to the database and the salt, and want to get back to the password.

**Feng Hao:** Oh I see, but if the attacker is able to read a cookie and read the database then basically . . .

**Reply:** If they can read the database, they can brute-force the password, so you do have to use good passwords. But they can’t build an offline database because of the salt.

**Matthew Johnson:** And there are two scenarios given. If the attacker has access to a cookie which is valid, then they can extend them a little because it’s a stateless system so all of the state has to be in the cookie, but you can’t really defend against that if they have access to those. But they still can’t get the password without brute-forcing it.

**Reply:** Yes. Maybe we can look at it in three examples. One is the attacker has access to the cookie only, and then I think this is cryptographically secure, one is that the attacker has access to the database, and I think that’s secure. The third option is the attacker can brute-force the password because the password is poor, and then if the attacker has access to the database it’s not secure.

**Michael Roe:** What do you do if your server receives a very large number of cookies which don’t match, as in somebody trying to do an online search for the password?

**Reply:** There needs to be rate-control at the login stage, I don’t know if I need to do a rate control at the cookie verification stage unless the attacker has the salt.

**Michael Roe:** That is exactly the case I was thinking of.

**Reply:** Well if the attacker has a salt then they probably have access to the database anyway, so they can do an offline attack. In the database, with the salt and $v$ you can then do an offline brute-force attack, so I don’t think online brute-force buys you anything extra.

**Matthew Johnson:** It buys you part of the hash algorithm, which is a lot.

**Reply:** Yes.

**Bruce Christianson:** How stateless are you? Does the server even know which clients are logged in?

**Reply:** Yes, I don’t think the server needs to know what clients are logged in. The slight different in state to the Fu et al. scheme is I do require that the client on every request is able to read the user login interface, whereas the Fu scheme doesn’t require that, it just requires that you read a login database on login, but everything else doesn’t need to touch the database at all.