A Method to Protect Passwords in Databases for Web Applications

Scott Contini

ABSTRACT

Trying to make it more difficult to hack passwords has a long history [3, 14, 20, 16]. However the research community has not addressed the change of context from traditional Unix mainframe systems to web applications which face new threats (DoS) and have fewer constraints (client-side computation is allowed). In absence of updated guidance, a variety of solutions are scattered all over the web, from amateur to somewhat professional. However, even the best references have issues such as incomplete details, misuse of terminology, assertion of requirements that are not adequately justified, and too many options presented to the developer, opening the door to potential mistakes. The purpose of this research note is to present a solution with complete details and a concise summary of the requirements, and to provide a solution that developers can readily implement with confidence, assuming that the solution is endorsed by the research community. The proposed solution involves client-side processing of a heavy computation in combination with a server-side hash computation. It follows a similar approach to a few other proposals on the web, but is more complete and justified than any that we found.

1. INTRODUCTION

The way many modern web applications protect passwords in databases is similar to the the way operating systems protect passwords in databases. But are the security requirements for these two different scenarios the same? Maybe not.

A good survey on password protection in modern operating systems is [3]. The initial ideas for protecting passwords in Unix operating systems came from Morris and Thompson [14]. Their goal was to design a system that would make it hard for hackers to find even poorly chosen passwords assuming the database was public. This assumption is important because in practice it is hard to keep databases secret. Morris and Thompson used a slow one-way function to encode passwords, the output of which was stored in the database. The purpose of it being slow was so that it would impede hackers from determining passwords via trial and error or via dictionary attacks. They also introduced the concept of salt which prevented an attacker from going after multiple accounts at once, and also staved off precomputed table attacks. While it was good for its time, their solution soon became insufficient due to increases in hardware performance which made dictionary attacks and precomputed tables viable again.

Improved solutions came in 1999 with bcrypt [20] and in 2000 with PKCS #5 V2.0 [13, 11]. bcrypt was actually specifically designed to fill the gap that Unix systems had, namely the problem with the algorithm becoming less secure due to hardware advances outpacing general purpose computing advances. On the other hand, PKCS #5 was originally intended to be a password based key derivation function (PBKDF): that is, a function that turns a password into a cryptographic key, used to encrypt and decrypt documents. Since PBKDFs have similar requirements to functions used to protect passwords in databases, Version 2 of the standard made a passing remark that it could also be applicable to storing encoded passwords in databases. We will discuss this more in Section 1.4.

In 2009 Colin Percival introduced scrypt [16] which is an improvement upon bcrypt. As noted in [18], the main threat against bcrypt in 1999 was ASICs with low gate counts, but today the threat is FPGAs and bcrypt was not designed to protect against that threat.

In the above cases, a function is being provided that developers can put passwords into to get an encoded result. The solutions are improvements upon [14] for protecting passwords in Unix systems. But do the same assumptions for protecting passwords in traditional Unix systems apply to today's web systems? Prior to making hand waving arguments suggesting that it should be easy to translate their ideas into the web context, one should do a little web searching to see the confusion web developers have in implementing such a solution. Crackstation.net [7] gives good examples of the issues that developers struggle with including which side (client or server) does the heavy computation, how to choose the salt, the threat of denial of service, etc....

In fact, the web is scattered with advice on how to properly implement such a solution for a web base system, ranging from amateur to somewhat professional. The problem is

that it is not so easy for a web developer to distinguish between the amateur and professional recommendations since there is no authoritative standard or publication telling developers how they should do it. This gap needs to be filled by the research and/or standards community.

Our research aims to fill this gap by providing such a solution. For developers who just want to know what the final answer is, we direct them to Section 4. The rest of the paper is aimed at a research audience.

1.1 What about SRP and related work?

SRP [24], the Secure Remote Password Protocol, was introduced in 1998 as a way for a client to authenticate to a server without revealing the password to the server. Unlike the operating system based solutions, SRP was designed for web usage. It has a number of desirable security properties, such as not requiring a TLS connection for its security. However, it does not prevent offline guessing attacks in the event of a database leak. This was noted in [20], including the point that SRP can be combined with their solution to deal with the database leak scenario. Still, neither of these papers address a threat that we believe should be addressed: denial of service attacks which force the server to consume heavy resources.

1.2 Our Contributions

This note will attempt to make the following research contributions:

- A claim that the security requirements for protecting passwords in modern web applications are different from that of traditional Unix mainframes, and the advice given by the research community has not addressed this change. Notably, modern web applications have to defend against denial of service attacks whereas traditional Unix mainframes did not worry about this.
- A recognition that there are numerous developers trying to craft various solutions to this problem on the web due to the lack of guidance from the research community.
- A recommended solution to the problem is given which partially overlaps with some independent solutions off the web, but additionally fills in gaps that others do not.
- A consolidation of various suggestions from solutions on the web.
- An identification of the exact requirements needed to solve the problem.
- An identification of wrong/inaccurate requirements given by various proposals on the web.
- An identification and correction of the misuse of terminology.

In general, what is needed is for the research community to address the problem that many web developers are trying to solve, and to give a blessing to some solution. This note aims to provide such a solution.

1.3 Requirements for Protecting Passwords

The solution presented in this article is based upon resisting certain types of attacks. Upon listing the attacks, we can derive security requirements from them. Provided that the threats we list correctly represent the real threats to web systems, we can conclude that our solution satisfactorily meets what web developers require for protecting online systems.

We limit ourselves to attacks that the server can defend against. For example, attacks such as phishing and client-side key loggers are outside the scope of what the server can protect against. With that in mind, we claim that the threats in scope to stored password-based web systems are:

- An attacker might get access to the database (examples: SQL injection, insecure backups, snooping admin), potentially allowing him to get access to every body's passwords.
- Attacker might be able to launch denial of service (DoS), possibly distributed (DDoS), attacks on the password verificiation mechanism by forcing the server to consume heavy resources.

The second one deserves some discussion. The threat of a denial of service that exploits the password verification method has been discussed in software development circles [7, 18, 21, 5, 9]. In fact, there was a security patch for the Django framework due to a related denial of service possibility but in this case they blamed it on the fact that they did not limit password size [6]. However, the general threat of a DoS or DDoS is a real one due to the imbalance between client-side request and server-side computation time.

There are a number of ways that DoS can be performed in the TCP/IP protocol suite – see the survey paper [2]. The way many web applications implement authentication opens the door to another way that they could be vulnerable to the consumption of a scarce, limited, or non-renewable resource: namely CPU time. One might argue that standard defenses such as Crackstation.net's recommendation of Captchas [7] can apply here, but it is far more elegant to have a solution that inherently protects against this threat without annoying legitimate users. See also Section 3.3 for another defense option.

From the threats listed above, we derive the following three requirements:

1. Access to the database must be resistant to both online and offline password guessing attacks.
2. The server must not be vulnerable to "pass the hash attacks" [15]. This means that if the attacker gets a hold of the database, that should not allow him to trivially impersonate anybody by simply sending the information stored in the database to the server.
3. Verification of the data provided by the user should be fast.

The problem is that the research community has not addressed these requirements that may seem contradictory at first glance. In fact, none of the publications [3, 14, 20, 16] mention the DoS threat, or consider the possibility that the secure/heavy computation can be done on the client-side. Perhaps the reason for this is timing, since the first DDoS occurred in 1999 [12] and the trend to client-side computation has largely been pushed by html5 which is only now being accepted as the new standard for web markup language.

Going in the direction of client-side computation, we can offer an additional "nice to have" feature for such a solution is that the server never sees the user's plaintext password[1], similar as to [24]. For example, think of the Heartbleed attack which would leak exactly what the server received from the client, or think of a dishonest system administrator who monitors memory. For users who had the same password on multiple systems[2], the compromise of a single system that receives a plaintext password would imply that the attacker could get access to other systems that the user shares that password on.

1.4 A Note about Terminology

There is a lot of confusion about terminology when reading various sources on this topic. Much of this seems to be due to PKCS #5 V2.0 [13].

PKCS #5 was originally written as a method to turn a password into a *cryptographic key for the purpose of encryption or decryption*. For example, if one wants to encrypt a document using a user-entered password, the password typically is short for the purpose of memorisation and usability (humans make errors in typing long passwords which hurts usability), yet keys required for encryption are long. Thus the method in PKCS turns the short password into a long cryptographic key which can be used to encrypt and decrypt a document.

Version 1.5 of the PKCS #5 standard was designed for only this purpose, and there is no mention of using it for other purposes such as password based authentication. It is only Version 2 of PKCS #5 that the door opens to other uses.

It in Version 2.0 of PKCS #5 where the term password based key derivation function (PBKDF) is introduced. It is also in Version 2.0 of the standard where they suggest that it can additionally be used for protecting passwords in databases for the purpose of authenticating users. At this time, they also blur the definition of "key" as seen in the text quoted below:

> A general approach to password-based cryptography, as described by Morris and Thompson for the protection of password tables, is to combine a password with a salt to produce a key. The salt can be viewed as an index into a large set of keys derived from the password, and need not be kept secret.

The above text refers to the "key" as the output of a cryptographic function (a PBKDF). But Morris and Thompson did not use the word "key" in this way. They used it to mean the cryptographic key of the cipher that they were using as a one-way-function. In their case, the password and salt were combined to form the cryptographic key, which is used to encrypt a constant value. The output was referred to "encrypted result" (not "key") and was stored in the database table. In summary, Morris and Thompson used the word "key" consistently with cryptographic literature whereas PKCS #5 does not.

PKCS #5 V2.0 then goes on to suggest that it is straightforward to define password-based encryption and message authentication schemes from the standard, but then takes it one step further:

> It is expected that the password-based key derivation functions may find other applications than just the encryption and message authentication schemes defined here.... Another application is password checking, where the output of the key derivation function is stored (along with the salt and iteration count) for the purposes of subsequent verification of a password.

The last sentence is the invitation for the expanded use. It's worth noting that the NIST Special Publication 800-132 [22] does not allow the use of the PKCS #5 PBKDF2 in this way for FIPS certification.

Encoded passwords used for authentication are not cryptographic keys. So calling the encoded passwords "keys" or using the term "password based key derivation function" to describe the encoding process is an abuse of terminology. This abuse is seen in many places online including [7, 9, 5]. Thomas Pornin has also pointed out this terminology abuse [19].

Having said that, it is indeed true that the requirements for the function that processes passwords to be protected in a database overlap with that of a PBKDF. So therefore PBKDF2 is an acceptable function for protecting passwords in web applications even though the terminology behind it for this context is wrong.

In this document, we will refer to the function performing the heavy computation involving direct input of the password (bcrypt, PBKDF2, scrypt, etc...) as a *password processing function* (PPF) and the output of it as the encoded password.

The PPFs [20, 16, 13] take at least a salt, password, and cost parameters. The cost refers to the amount of effort to attack it, and is related to the number of iterations used in the PPF. PPFs may take other parameters (such as output size), which we will refer to as "misc". We will thus refer to

[1] If done in Javascript, then secure client-side controls such as in [17] would also be needed to enforce this.

[2] We are all told that we should not do it, but most people do due to the complexity of managing multiple passwords, including the complexity of password manager tools.

the PPF computation as

$$PPF(\text{salt}, \text{password}, \text{cost}, \text{misc})$$

2. DRAFTING A FIRST SOLUTION

The problem with using a PPF alone to protect passwords in databases is that it does not seem to allow meeting requirements (1) through (3) in Section 1.3. To see this, note that:

- If the PPF computation is done on the client-side, then the resulting value will match what is in the database. This means that any attacker who gets access to the database can bypass the PPF by simply passing the known hashed value to the server (pass-the-hash).
- If the PPF computation is done on the server-side, then requirements (1) and (2) above are met, but goal (3) is not because it puts the burden on the server to do a heavy computation to verify the authenticity of the user.

Crackstation.net acknowledges these shortcomings and recommends a few options to deal with them. One of them actually has potential (where "key stretching" is meant to mean using a PPF on the user entered password):

> If you are worried about the computational burden, but still want to use key stretching in a web application, consider running the key stretching algorithm in the user's browser with JavaScript. The Stanford JavaScript Crypto Library includes PBKDF2. The iteration count should be set low enough that the system is usable with slower clients like mobile devices, and the system should fall back to server-side computation if the user's browser doesn't support JavaScript. Client-side key stretching does not remove the need for server-side hashing. You must hash the hash generated by the client the same way you would hash a normal password.

The paragraph suggests that one can do the PPF on the client-side as long as one hashes the result on the server-side. Similar ideas have been expressed by [21, 5, 9].

Let's explore this more. Consider the following implementation:

- Database stores { username, salt, hash(s) } where

  $$s = PPF(\text{salt}, \text{password}, \text{cost}, \text{misc})$$

  for each user. Note that neither s nor password is stored.
- For user to authenticate, client gets salt and computes $\varsigma = PPF(\text{salt}, p, \text{cost}, \text{misc})$ where p is the user entered password and sends { ς, username } to server via secure TLS connection.
- Server computes hash(ς) and checks it against the hash(s) in the database for that user. It accepts if and only if there is a match.

For now ignore the issue of the salt coming from the server to the client and consider how this solution satisfies the requirements above:

- Access to the database resistant to password guessing attacks seems to be preserved. The attacker must either compute the PPF computation using a dictionary of words or he must invert the hash. The former is time consuming: this is the purpose of PPF. The latter is impossible for a secure hash[3]: hash functions are designed to be non-reversible.
- The server is not vulnerable to a pass the hash attack. Whatever the server receives, it is going to hash it and then compare the result to the database. In order to pass-the-hash, the attacker would need to know s or a second preimage of hash(s) which are not available from the database. Knowing these values is hard according the the definition of secure hash function.
- Verification of the data on the server-side is quick. The server only needs to compute the hash of an small amount of data[4] and do a database lookup. These operations do not require heavy resources.

Thus, the goals are met.

3. IMPLEMENTATION REQUIREMENTS

To make a full solution, we need to be more specific about the requirements for salt.

The original purpose for the salt [14] was to force an attacker who has access to the database to hack each user's password individually rather than to hack all passwords at once. The salt was to assure that two users in a system do not have the same encoded password even if they had the same password. A better and stronger requirement is that two users should not have the same encoded password even if they are on different systems. Morris and Thompson also noted the salts impede the possibility of preparing precomputed tables in advance that allow looking up passwords for users.

As we move to a web based solution, particularly a client-side solution, we find new requirements for the salt.

3.1 Analyzing Crackstation.net's Solution

Crackstation.net identifies the following issue with the salt coming from the server:

> "The obvious solution is to make the client-side script ask the server for the user's salt. Don't

[3]Assuming the input space is large enough. This can be met by requiring the output of the PPF to be at least 128-bits.
[4]Of course, the server should verify that the data is small before computing the hash in order to avoid the [6] vulnerability.

> do that, because it lets the bad guys check if a username is valid without knowing the password. Since you're hashing and salting (with a good salt) on the server too, it's OK to use the username (or email) concatenated with a site-specific string (e.g. domain name) as the client-side salt."

The point about bad guys learning valid usernames in the system is a real one for web base-systems since it helps outsiders identify targets. In the event of not being able to get the system database, being able to find valid targets is a fallback path for hacking user accounts. This threat is commonly known as account enumeration [1].

What about the server-side salt? The following three claims are made, none of which are justified:

- "Every time a user creates an account or changes their password, the password should be hashed using a new random salt. Never reuse a salt."
- "The salt also needs to be long, so that there are many possible salts. As a rule of thumb, make your salt is [sic] at least as long as the hash function's output."
- "Salt should be generated using a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) As the name suggests, CSPRNGs are designed to be cryptographically secure, meaning they provide a high level of randomness and are completely unpredictable. We don't want our salts to be predictable, so we must use a CSPRNG."

Although these choices feel wise, the approach of throwing everything but the kitchen sink for defense without being clear of what we are defending against is not scientific, and potentially introduces unnecessary complexity. So we therefore would like to analyze the solution to identify more exact requirements, and tailor our recommendation accordingly.

The first two claims from Crackstation.net seem to be for the same purpose: preventing salt reuse. The third claim of unpredictability could potentially have another purpose.

Although [14] implied that different users should have different salts, what is not clear is why the user's salt must change when his password changes. More specifically, to the recommendation of using a client-side salt of username (or email) concatenated with a site-specific string (e.g. domain name), what are the dangers of server-side salt reuse?

Thomas Pornin gives his rationale [19] for the dangers of salt reuse, but much of it is already precluded by the client-side salt choice. The exception is this one claim that seems to be wrong:

> "...because people tend to generate their passwords "in sequence": if you learn that Bob's old password is "SuperSecretPassword37", then Bob's current password is probable "SuperSecretPassword38" or "SuperSecretPassword39"."

Generating passwords in sequence is a problem independent of salt. Regardless of whether or not the salt is reused, if one password is known then the other can be readily guessed (either online or offline attack). Using the same salt does not leak extra information for passwords in sequence.

In our case, the server-side salt is never exposed to users that do not hack the system. So we start by assuming that the database has been exposed to a hacker and ask what advantages does he have by knowing that salts are reused? To exploit this, we would also need to assume that the attacker can discover future databases, so that he can mount offline attacks against targeted users. As a consequence, the attacker can do a heavy precomputation early so he can quickly lookup future versions of users' passwords later. In other words, a one-time computation which stores a large set of encoded passwords for a list of targeted users allows the attacker to do future password searches very inexpensively via a simple table lookup.

But does Crackstation.net's requirements above prevent such an attack? Hardly. In fact, there is a slight modification to the above attack which only makes it mildly more difficult to carry out. Rather than the one-time precomputation which computes a set of encoded passwords for a targeted set of users, the hacker instead does a precomputation that involves only the PPF for those users. In other words, the values of ς for various password possibilities. The results are stored in a set of files indexed by the username. Then, the next time the attacker discovers the database, he will know the real server-side salts – how they were generated or how big they are are irrelevant to him. He simply hashes the various ς values from the large files with the corresponding real server-side salts to see if he discovers the targeted users' encoded passwords. These subsequent computations are very fast since they bypass recomputing the PPF.

The conclusion bifurcates according to whether or not the reader believes that protecting against future accesses to the database is a realistic threat or not. If the reader does not believe it, then server-side salt adds no value at all because an attacker can't mount a new offline attack without getting the new database. If the reader does believe it, and we believe they should, then we just showed that as long as the PPF is computed on the client-side only, server-side salting prior to the final hash computation offers almost no value.

3.2 Proposed Solution for Salt

At this point one might modify Crackstation.net's solution in order to put more of a burden on the server-side, but that is exactly what we want to avoid according to our requirements given in Section 1.3. The only way to deal with this problem properly is to bring variability into the client-side PPF function. But how to do so in such a way that does not leak information about valid users in the system to those who do not discover the database?

The key point here is that we only have to prevent that leakage to those who do not discover the database: those who have discovered it already know the users.

We propose the following solution. Let σ be a system level secret that changes every year. The value of σ will be used

to create pseudo-salts for users that do not exist in the database.

At any time a user changes his password or a new user is created in the system, a randomly generated value υ is stored in the corresponding row of the database for that user. The salt for that user is taken to be hash(username $\|$ domainname $\|$ υ), but this is not stored for a reason explained below.

When a user tries to authenticate, the server first looks up the user in the database. If the user is found, then the system uses the corresponding υ in the database to compute the salt and return it to the client. If the user is not found, then the system computes the pseudo-salt using the current system-level σ value, and sends it to the client.

By this design, a salt (which might be a pseudo-salt) is always returned to the client so we avoid the problem of absence of a salt revealing whether the username is valid or not. We also always compute a salt in order to prevent timing attacks from leaking whether a user was valid or not.

Obviously, if a new randomly generated value was sent to the client when a user was not present in the database, then two consecutive calls for that username would reveal whether the username was real or not. By fixing υ for a year (or similar long period) we aim to obstruct hackers from using this attack. Still, if a hacker was persistent enough to try this attack, what can he learn?

- If the attacker tries the same username twice more than a year apart and gets the same salt, then he learns that the username is indeed in the database and that user has not changed his password over that time period.
- If the attacker tries the same username twice within one year and gets the same salt, then he learns that either the user is in the database and the password has not changed in that time frame, or else the username is not in the database and the current σ has not changed.
- If the attacker tries the same username twice more than a year apart and gets different salts, then he learns that either the username does not exist in the database (σ has changed in this time period), or it is in the database but the password has changed in that time, or the user was newly created within that time frame.
- If the attacker tries the same username twice within one year and gets different salts, then he learns that either the username is not in the database and the current σ has changed, or else it is in the database but the password has changed in that time, or the user was newly created within that time frame.

While the first bullet point directly gives the attacker information that he is after, the remaining three cannot be used so directly. Perhaps by querying many users and analyzing the results the attacker can derive more valuable information, but from a practical point of view, the attacker is greatly inhibited.

3.2.1 Requirements for σ and υ

For our solution, the uniqueness of salts is already a consequence of having the username and domainname being input into the hash function. But what else do we require for σ and υ?

- We do not want an attacker to be able to determine σ from his queries for salts. Therefore it should be at least 128-bits.
- If an attacker finds out a past value of σ, we do not want him to be able to guess future values of σ. This is to cover the case that he gets access to the database at some point in time but not again in the future. For this reason, we require it to be generated by a CSPRNG.
- We do not want the attacker to know when a value of υ is being used instead of the system secret σ. Therefore, the values of υ must have the same security requirements as σ (at least 128-bits and generated from a CSPRNG).

One last point that needs to be made is that some PPFs limit the size of the salt. For example, bcrypt allows only 128-bit salt [20]. It is okay to truncate the to meet this restriction.

3.3 Server-side resource requirements

The requirement of using a hash to compute a salt means that in total, the server needs to compute 2 hashes to complete an authentication request (whether it is successful or not). There is no requirement to store any state on the server side in this process.

An interesting solution to DoS is presented in [4]. From our viewpoint, [4] is ideally applied as a defense against SSL/TLS DoS threats, so it is not necessarily incompatible with our solution. Nevertheless, it is interesting to do an 'apples-to-oranges' comparison between the two.

In the face of a DoS attack, [4] always computes a single hash computation to verify whether or not the client is legitimate (i.e. the puzzle has been solved). If the hash check verifies, it also has to check that the client nonce has not been used before (database lookup). The server stores values of client nonces already used only for clients that passed the puzzle test.

Our solution (Section 4) requires one more hash computation and potentially one more database lookup, but has the benefit that it does not store any state information. It also has the benefit that it is not protected by IP [10].

4. PUTTING IT ALL TOGETHER

If you're a developer, you might only care about the final recommendation. This section is it. The terminology in Section 1.4 defines PPF. All communication happens through a secure TLS session.

- Database stores { username, hash(s), υ } where

$$s = PPF(\text{salt}, \text{password}, \text{cost}, \text{misc})$$

for each user.

  - Each user has a different υ which is generated by a CSPRNG and is at least 128-bits.
  - The user's υ changes whenever his password changes.
  - The value s originally comes from the client via TLS when the user sets his password. It is transient on the server.
  - hash is a cryptographic hash function such as SHA256.
  - The salt is taken to be hash(username $\|$ domainname $\|$ υ). It is not stored in the database but instead is recomputed when required. If the salt could be longer than what is allowed for the PPF (such as in bcrypt [20]), then instead use as many bits as possible from the hash.
  - cost should be set to about 1 second computation time on the slowest device that is supported.
  - The output length of the PPF (which might be part of the misc parameter) should be at least 128-bits.

- Database stores system level secret σ that changes approximately once per year. σ is generated by a CSPRNG and is at least 128-bits.

- For user to authenticate, the following operations happen.

  - Client gets username and password from user.
  - Client sends a request with username to the server to get salt, cost, misc.
  - Server only accepts usernames within the allowed size limit. If it is not, the server rejects it and does not continue.
  - Server looks up username in database. If user exists, then it computes salt as hash(username $\|$ domainname $\|$ υ) for that user's υ value. If user does not exist, then it computes salt as hash(username $\|$ domainname $\|$ σ) for the system secret σ value. The salt, cost, and misc are sent to the client.
  - The client computes

    $$\varsigma = PPF(\text{salt}, p, \text{cost}, \text{misc})$$

    where p is the user entered password and sends { ς, username } to server via secure TLS connection.

- Server verifies that ς is the expected length, and if not, it rejects it and does not continue.

- Server computes hash(ς) regardless of whether or not the user exists in database. Server accepts user if and only if user exists[5] and its computation of the hash matches the value in the database.

  It accepts if and only if there is a match.

[5]To avoid storing state information, this should be done by a new database lookup.

This idea is similar in spirit from ideas proposed by Crackstation.net [7], Foy Stip and David Wachtfogel [21], and hbCyber and Thomas Pornin[9]. However we provide a complete solution based upon analysis with all of the devilish details worked out, and we have also shown that solutions such as [7] are not backed up by analysis.

In [9] Thomas Pornin remarks:

> Client-side hashing works, conceptually. The problem, though, is that of power. In a Web context, client uses Javascript, and Javascript is feeble for computing intensive tasks. The client already uses a system which may have a relatively small CPU (it could be a cheap smartphone, for instance), but Javascript adds its own overhead, which is huge (because it is interpreted, hard to JIT, and lacks decent integer types).

While this is indeed a legitimate concern for mobile apps [8], it is likely that it will not be in the near future due to the W3 crypto initiative which is making cryptographic functionality available through the browser [23], thus not needed to be implemented in Javascript.

In the event of a database leak, what matters most is the hacker's computation cost compared to the PPF speed of the weakest device supported by the web application. Our design assumes that a legitimate user on a slow device will not be bothered by a 1 second delay for the login PPF computation. It is important for system designers to understand that there is a tradeoff between security and support for constrained devices. In systems that protect high valued assests (such as online banking), security requirements should take precedent over a wide range of usability (i.e. low powered devices). In other systems (such as online bulletin board systems), the wide range of usability may be more important. Generally, the amount of effort expended by the attacker will be correlated to the value of the assets he can potentially obtain.

Acknowledgements

This research was conducted in the course of employment at Covata. The author would like to thank Blair Strang and David Yeung for their valuable reviews and feedback on various versions of the article.

5. REFERENCES

[1] Testing for user enumeration and guessable user accounts (owasp-at-002). `https://www.owasp.org/index.php/Testing_for_User_Enumeration_and_Guessable_User_Account_(OWASP-AT-002)`, October 2012.

[2] M. Abliz. Internet denial of service attacks and defense mechanisms. Technical Report TR-11-178, Univeristy of Pittsburgh, March 2011.

[3] S. Alexander. Password protection for modern operating systems. *j-LOGIN*, 29(3), June 2004.

[4] T. Aura, P. Nikander, and J. Leiwo. Dos-resistant authentication with client puzzles. In *Revised Papers from the 8th International Workshop on Security*

Protocols, pages 170–177, London, UK, UK, 2001. Springer-Verlag.

[5] Authors. https://news.ycombinator.com/item?id=7626587, April 2014.

[6] J. Bennett. Security releases issued. https://www.djangoproject.com/weblog/2013/sep/15/security/, September 2013.

[7] Crackstation.net. Salted password hashing: doing it right. https://crackstation.net/hashing-security.htm, February 2014.

[8] D. Crawford. Why mobile web apps are slow. http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/, July 2013.

[9] hbCyber. Stackexchange post on secure authentication: partial client-side key stretching. http://security.stackexchange.com/questions/43023/secure-authentication-partial-client-side-key-stretching-please-review-crit, September 2013.

[10] A. Juels and J. Brainard. Cryptographic countermeasures against connection depletion attacks, Mar. 27 2007. US Patent 7,197,639.

[11] B. Kaliski. PKCS #5: Password-based cryptography specification version 2.0, September 2000. RFC 2898.

[12] G. C. Kessler. Defenses against distributed denial of service attacks. http://www.garykessler.net/library/ddos.html, November 2000.

[13] R. Laboratories. PKCS #5: Password-based cryptography standard. http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-5-password-based-cryptography-standard.htm.

[14] R. Morris and K. Thompson. Password security: A case history. *COMMUNICATIONS OF THE ACM*, 22:594–597, 1979.

[15] NSA. Reducing the effectiveness of pass-the-hash. http://www.nsa.gov/ia/_files/app/Reducing_the_Effectiveness_of_Pass-the-Hash.pdf, November 2013.

[16] C. Percival. Stronger key derivation via sequential memory-hard functions. *BSDCan*, 2009.

[17] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, April 2014.

[18] T. Pornin. Stackexchange post on bcrypt vs PBKDF2. http://security.stackexchange.com/questions/4781/do-any-security-experts-recommend-bcrypt-for-password-storage/6415/, August 2011.

[19] T. Pornin. Stackexchange post. http://security.stackexchange.com/questions/211/how-to-securely-hash-passwords/31846/, June 2013.

[20] N. Provos. A future-adaptable password scheme. In *In Proceedings of the 1999 USENIX, Freenix track (the on-line version*, page 99, 1999.

[21] F. Stip. Client side password hashing post on stackexchange. http://security.stackexchange.com/questions/23006/client-side-password-hashing/23012, October 2012.

[22] M. S. Turan, E. Barker, W. Burr, and L. Chen. NIST special publication 800-132: Recommendation for password-based key derivation. http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf, December 2010.

[23] w3c. Web cryptography api. http://www.w3.org/TR/WebCryptoAPI/, December 2043.

[24] T. Wu. The secure remote password protocol. In *In Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, 1998.