Cracking PwdHash: A Bruteforce Attack on Client-side Password Hashing

David Llewellyn-Jones(✉) and Graham Rymer

University of Cambridge, Computer Laboratory,
William Gates Building, 15 JJ Thomson Avenue,
Cambridge CB3 0FD, UK
{David.Llewellyn-Jones,Graham.Rymer}@cl.cam.ac.uk
https://www.cl.cam.ac.uk

Abstract. PwdHash is a widely-used tool for client-side password hashing. Originally released as a browser extension, it replaces the user's password with a hash that combines both the password and the website's domain. As a result, while the user only remembers a single secret, the passwords received are all unique for each site. We demonstrate how the hashcat password recovery tool can be extended to allow passwords generated using PwdHash to be identified and recovered, revealing the user's master password. A leak from a single website can therefore compromise a user's account on other sites where PwdHash was used. We describe the changes made to hashcat to support our approach, and explore the impact this has on speed of recovery.

Keywords: passwords; password cracking; brute-force attacks; user authentication

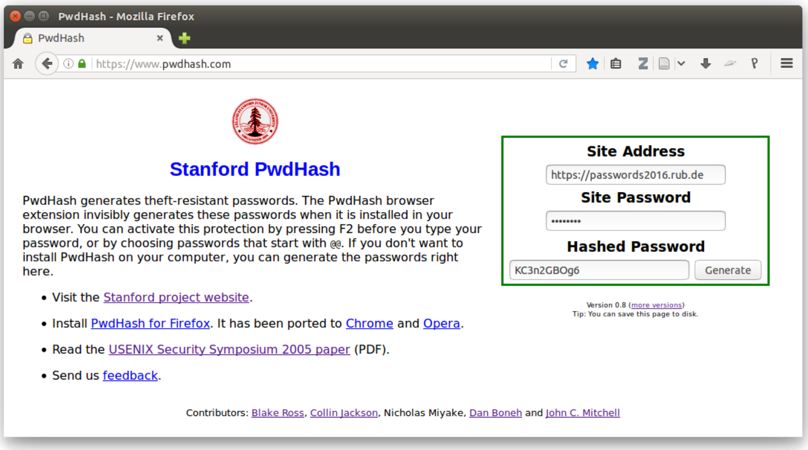
1 Introduction

Despite their many shortcomings, passwords have become the *de-facto* mechanism for user-authentication online. At each site where a user registers, they are asked to create a unique, complex and secret password that must also often fulfil a number of other site-specific constraints [4]. Users are increasingly relying on coping mechanisms such as browser autofill, password managers and password generation algorithms in order to lighten the burden of password management [16]. Ross *et al.* proposed one such mechanism in 2005 called *PwdHash* [9]. Essentially a password generation scheme, PwdHash calculates a hash of the user's master password and the domain-name of the website on the client-side in order to create a strong, site-specific password that the user doesn't need to remember.

Most password managers allow the user to choose their own password for a site. The password manager's role is to store the password and allow the user to retrieve it when logging back in to the site. Most password managers will also integrate with common browsers. This allows them to automatically extract the password for the page the user is currently viewing and autofill any password fields on the page.

For this to work a database of passwords must be stored somewhere; either locally, or in the cloud. If stored locally, the user is restricted to accessing services on the machines where the password database is stored. If remotely, there is a danger the password database may become compromised, exposing all of the user's passwords to an attacker. To mitigate this, it's common for the password database to be encrypted with a master password provided by the user.

Password generation algorithms offer an alternative scheme. They are attractive for users since they don't rely on a database of passwords being stored, giving two obvious benefits. First that there is nothing for an attacker to compromise which provides a heightened sense of security. Second that there is no constraint on where the system can be used. Passwords can be re-generated on any system where the algorithm is available. In the case of PwdHash, the algorithm is available as a web page [8] as shown in Figure 1, a browser plugin[1], or as various smartphone apps[2]. This means that it's usable on any system allowing access to the PwdHash website with Javascript enabled.



Stanford PwdHash

PwdHash generates theft-resistant passwords. The PwdHash browser extension invisibly generates these passwords when it is installed in your browser. You can activate this protection by pressing F2 before you type your password, or by choosing passwords that start with @@. If you don't want to install PwdHash on your computer, you can generate the passwords right here.

- Visit the Stanford project website.
- Install PwdHash for Firefox. It has been ported to Chrome and Opera.
- Read the USENIX Security Symposium 2005 paper (PDF).
- Send us feedback.

Site Address: https://passwords2016.rub.de

Site Password: ••••••••

Hashed Password: KC3n2GBOg6

Generate

Version 0.8 (more versions)
Tip: You can save this page to disk.

Contributors: Blake Ross, Collin Jackson, Nicholas Miyake, Dan Boneh and John C. Mitchell

Fig. 1. The PwdHash website for generating site- and user- specific passwords.

We'll discuss the algorithm used by PwdHash in more detail in Section 2, but as a rough overview, it can be summarised as follows

$$P_D = H_1(P_M \mid S_1 \mid D) , \tag{1}$$

where H_1 is a cryptograhic hash function, P_M is the user's master password, S_1 is a theoretical fixed salt chosen by the user (but not included in the current implementation), D is the website domain and P_D is the password generated for use with the given domain.

[1] `https://addons.mozilla.org/en-GB/firefox/addon/pwdhash/`
[2] `https://play.google.com/store/apps/details?id=com.mathijsjh.pwdhash`

We consider the susceptibility of this approach, and the algorithm used by PwdHash in particular, to offline brute-force attacks on leaked password databases.

2 The Problem

It is becoming disturbingly common for databases containing password hashes to be leaked online. The *Have I been pwned?* (HIBP) site, which aggregates breach data to allow users to check whether their account has been compromised, contains at present over 1.3 *billion* compromised accounts, including nearly 360 million Myspace and 164 million LinkedIn accounts[3]. Much of the data collected by HIBP comes from the @dumpmon Twitter-bot created by Write, which monitors paste sites for ‘interesting’ content[4]. Statistics released by Jordan show that dumpmon found 46 significant database dumps every day amounting to 1 million unique hashes over the 12 month period running up to May 2015 [17]. These statistics do not include the 500 million accounts from the Yahoo! breach that was confirmed while this paper was under review, and which have not yet been added to the HIBP database.

Any site that takes security seriously will apply a salted hash to a user’s password before storing it [6]. This step is intended to mitigate the risk of a database breach or leak, since there is – in theory – no computationally feasible way to reverse the hash function to reveal the password. In practical terms, increases in computational power are making brute-force attacks feasible, even on commodity hardware. The current state-of-the art software in this area is the *hashcat* password recovery tool. A recent analysis by Ruddick and Yan [10] benchmarked oclHashcat[5] with throughput[6] of approximately 1462 MH/s for SHA1 running on an HD6870 GPU[7]. Benchmarks posted by Gosney[8] – a member of the hashcat team – for the software running on a machine with 8 $\times$ GTX Titan X GPUs show astonishing throughput of 48.87 GH/s. This is fast enough to exhaust the complete keyspace of a 12 character password comprised of the base64 alphabet in a little over 12 hours.

The ability to crack passwords at this rate would be relatively unimportant were it not for the fact that users routinely re-use their passwords across multiple accounts [2, 15]. Consequently a breach at one service allows an attacker to compromise multiple other services. One of the stated aims of PwdHash is to mitigate against this, but one conclusion of our work is that in its present form,

[3] `https://haveibeenpwned.com/`

[4] `https://twitter.com/dumpmon`

[5] Originally hashcat was provided as two versions: *hashcat* and *oclHashcat*; the latter being for use on GPUs. Since the publication of Ruddick and Yan’s paper the software has been combined into a single version called *hashcat*.

[6] Throughput measured in MH/s = million hashes per second.

[7] 1120 stream processors clocked at 900 MHz.

[8] `https://gist.github.com/epixoip/63c2ad11baf7bbd57544`.

PwdHash does not succeed in this aim. Moreover the case of PwdHash is arguably more serious than for other passwords, since PwdHash encourages users to input the same master password for every site. PwdHash also allows users to essentially circumvent defences against weak passwords applied by a service. A PwdHash password will appear strong (comprised of a random sequence of alphanumerics and punctuation), even though it may be derived from a very weak password in practice. The service will think the password is strong because it only sees it after it was hashed by PwdHash, not before. Consequently the ability to reverse PwdHash passwords is a serious problem for the scheme.

We will demonstrate that the strength of PwdHash in its current form depends on the complexity of the user's master password to the same extent that a standard password does.

3 A Brute-Force PwdHash Attack

Our approach bootstraps the existing and well-developed brute-force cracking techniques used to reverse salted hashes leaked from website databases. To describe how we altered the hashcat password recovery tool to support this, it's helpful to formalise the PwdHash passwords that might be found in such leaks.

Referring back to Equation 1, we can see that the value P_D received by the website will be stored with an extra layer of hashing as follows.

$$H = H_2(S_2 \mid P_D) = H_2(S_2 \mid H_1(P_M \mid S_1 \mid D)) , \tag{2}$$

where H_2 is the hash function used by the site to protect their users' passwords and S_2 is a salt value chosen at random by the website and stored alongside the hash value H. Table 1 provides a set of example values.

Table 1. Example values in the hash creation process.

Stage	**Example value**
P_M	`correcthorsebatterystaple`
D	`linkedin.com`
S_1	None
H_1	HMAC-MD5
$P_D = H_1(P_M \mid S_1 \mid D)$	`NPxWlbriQoYdlQFE1y7lkwAAAA`
S_2	`tWLZc587`
H_2	SHA1
$H = H_2(S_2 \mid P_D)$	`4ddfdae77906baca73b6ecd129bd513905e6230f`

Given a leaked hash H with associated salt S_2, the standard hashcat approach generates a list of potential passwords $P_1, \ldots, P_n$ (where n is a Very Large Number) which are then hashed and compared against the leaked values until a match is found: $H_2(S_2 \mid P_i) = H$. Once such a match is found hashcat outputs P_i as the password corresponding to that hash.

If the value found is for a PwdHash-generated password, it will be the site-specific password (the value `NPxWlbriQoYdlQFE1y7lkwAAAA` from the examples in Table 1) that will be output rather than the master password. This value is of no use to an attacker, since it can't be used on any other site where the user has an account, due to the domain input D. Moreover the PwdHash value P_D will be harder to crack than a usual password, since it doesn't appear to be easily derivable from a combination of dictionary words, making it harder to guess. We avoid this problem by targeting the original master password, rather than the relatively harder-to-guess site-specific password.

To do this our approach adds an extra 'mangle' stage prior to the application of the site's hash, which applies the PwdHash algorithm. This replicates the use of PwdHash on the client-side by the user, combining the PwdHash hash and the site's hash into a single process. The input values P_i represent the possible master password values.

As a result, the process for cracking PwdHash hashes becomes equivalent to the process for cracking standard leaked hashes. There is a small overhead for performing the additional hash, but as we discover from our benchmarking (Section 4), the choice of HMAC-MD5 for the PwdHash algorithm means this overhead is low. Users of PwdHash are as vulnerable to brute-forcing attack as those using a single password for every site they visit. The security of their master password is therefore entirely reliant on the complexity of the password they chose: PwdHash provides no additional security.

4 The Details

In this section we will describe the changes we made to hashcat in order to support cracking of PwdHash passwords, the results of our use of the tool and benchmarking of its performance.

4.1 New hashcat Process

Hashcat uses a number of heuristic algorithms to choose the best possible passwords, P_i, to test, as shown in Table 2.

Table 2. Hashcat attack modes.

Attack mode	**Description**
Straight	Simple dictionary attack; test each entry from a dictionary in turn.
Combination	Combine pairs of words taken from two dictionaries.
Brute-Force	Try all keyspace combinations constrained by a user-supplied mask.

All of these algorithms are applied on the GPU in order to offer massive parallelism, and the GPU kernel code is highly optimised. Each attack type has its own implementation in order to maximise the efficiency of the process. This presents a challenge for the integration of the PwdHash algorithm. The standard approach capitalises on the way the input space is partitioned, with each GPU pipeline tasked with cracking a subset of the passwords, allowing a single GPU to concentrate on a subset of passwords all having similar characteristics. For example, in the case of the brute-force attack, only the first four bytes of each allocated subset of inputs will change. Consequently, a certain amount of pre-calculation can be performed that can be re-used across all of the hashes. Similarly, for the combination attack, a single GPU pipeline will be allocated inputs with the first of the two words the same and only the second word changing.

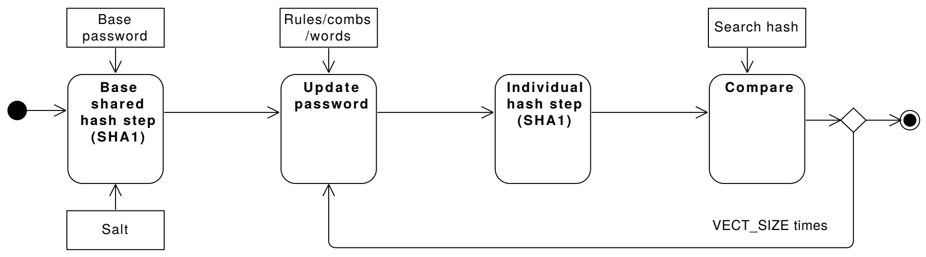


Fig. 2. The standard hashcat GPU process pipeline.

An overview of this process on the GPU can be seen in Figure 2. The pre-calculation step takes in the base password and salt, and performs the portion of the calculation shared across all of the passwords in the allocated subset. The GPU then enters a loop that updates the portion of the password that changes (*e.g.* overwriting the first four bytes with new values) and then completes the remainder of the hash steps required for this individual password. Finally the resulting hash is compared to establish whether it's one of the passwords to be cracked.

Applying PwdHash to these subsets will result in intermediate inputs which no longer share these patterns; every byte of the input password is liable to change for each cycle of the loop. As a result, we must move a portion of the pre-calculation back into the inner loop, as shown in Figure 3.

Here we can also see the crucial *mangle* step, which takes the password to be checked and passes it through the PwdHash algorithm before applying the final hash. The OpenCL code used to perform this stage can be seen in Appendix A and has also been made available on GitHub[9]. The PwdHash algorithm can be broken down into five parts: HMAC-MD5, base64-encode, truncate, character substitution and rotation. First the domain is passed through an HMAC-MD5 algorithm using the users's password as a key. The result is base64-encoded and then truncated to the same length as the password entered by the user. At this

[9] `https://github.com/llewelld/hashcat`

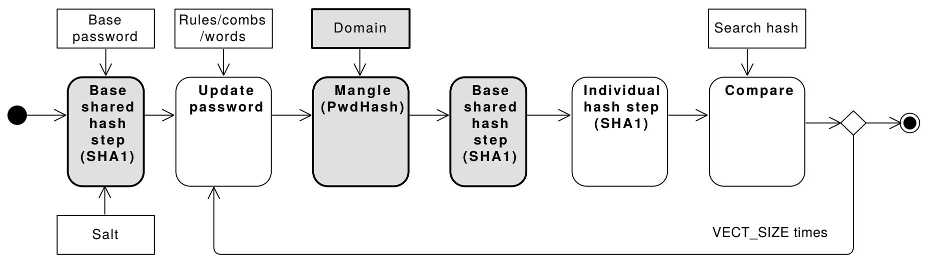


Fig. 3. The hashcat GPU process pipeline with PwdHash check integrated (the shaded activities indicate where alterations were made to the standard process).

point the result will be a string of uppercase/lowercase letters, numbers and punctuation (`+` and `/`). The authors identified that not all sites would accept the password in this form (*e.g.* due to constraints on the inclusion of punctuation, or different character types). At this point the actual implementation differs from the approach described in the paper [9]. Rather than creating a list of special cases as suggested in the paper, the algorithm instead allows the user to control the type of characters included in the output by altering the input master password. This is done by replacing the last four characters of the result to suit certain constraints. If the master password includes a capital letter and there's no capital letter already in the result, the fourth-to-last character is replaced by a capital letter. The same process is followed for the last three letters in relation to lowercase letters, numbers and punctuation respectively. Then, in the case where the master password does not include any punctuation, any punctuation characters are replaced by capital letters. The final step is to rotate the result so that the special characters don't all appear at the end of the hash.

The replacement characters and number of characters to rotate are determined using the output of the original HMAC-MD5 calculation. As an aside, entries are removed from the HMAC-MD5 value as they're needed, but rather peculiarly, if all of the entries are exhausted a value of zero will then be used each time. Since HMAC-MD5 generates a 16-byte value (22 characters in base64), any password over 22 characters will therefore have the letter 'A' or a zero byte repeatedly added to the end and no rotation applied. We believe this was unintentional, and while for longer passwords this increases the chance of collisions thereby improving the chance of an online attack succeeding, it may marginally reduce the effectiveness of the offline attack we describe here.

4.2 Benchmarks

In order to understand the performance impact of introducing the PwdHash step to the hashcat process, we ran benchmarks, the results of which are shown in Figure 4. Table 3 shows the raw values and details the extra overhead of running the PwdHash algorithm in comparison to the standard hashcat process. All tests were performed on an Amazon EC2 g2.2xlarge instance with eight Intel Xeon E5-

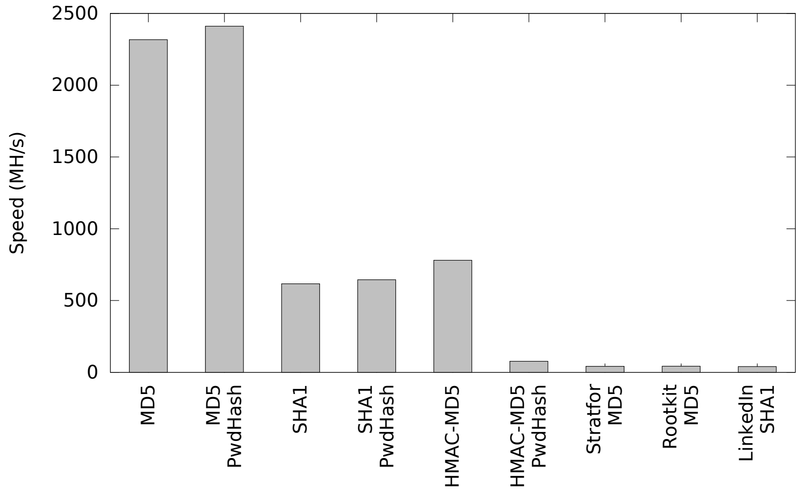


Fig. 4. Benchmarks running on an NVIDIA K520. Taller bars represent faster hashing.

Table 3. Benchmarks running on an NVIDIA K520. Overhead is the hashing speed with the PwdHash process as a factor of the standard hashing speed.

Hash type	**Speed (MH/s)**	**Overhead (factor)**
MD5	2316.7	1.0
MD5 PwdHash	2410.7	0.961
SHA1	616.9	1.0
SHA1 PwdHash	645.1	0.956
HMAC-MD5	780.3	1.0
HMAC-MD5 PwdHash	76.9527	10.140
Stratfor.com MD5	42.0085	n/a
Rootkit.com MD5	42.1938	n/a
Linkedin.com SHA1	39.7622	n/a

2670 vCPUs[10] clocked at 2.60GHz and an NVIDIA GRID K520[11] utilising two GPUs with 1536 CUDA cores each (3072 total), 4GB of graphics memory and 8 GB main RAM. The benchmarks represent 'ideal' speeds, so we also include results generated by applying the tool in a more realistic way to previously leaked hash data. These results will be discussed presently in Section 4.3.

The results show a varied picture. For vanilla MD5 and SHA1 hashing the extra code actually resulted in an unexpected marginal *increase* in speed. For HMAC-MD5 we saw a significant slow down of over ten times. This was also unexpected since the mangle step is essentially doubling the work (two HMAC-MD5 runs rather than one), highlighting the importance of proper optimisation of code for GPU execution, reflecting the results of Ruddick and Yan [10]. We've

[10] `http://ark.intel.com/products/64595/`

[11] `http://www.nvidia.com/object/cloud-gaming-gpu-boards.html`

currently performed minimal optimisation of our code; for example, the Pwd-Hash algorithm is applied independently of the site hash, and we could reuse the HMAC-MD5 code of the hash for the PwdHash step. This is left for future work.

4.3 Results

We applied the amended hashcat to the hashes disclosed in three high-profile leaks: Rootkit.com (58 677 unique unsalted MD5 hashes), Stratfor.com (822 657 unique unsalted MD5 hashes), and LinkedIn.com (2 936 840 unique unsalted SHA1 hashes). For our tests we configured hashcat to use the plaintext passwords disclosed in the infamous Rockyou.com leak (14 344 391 words) as a wordlist, in addition to the host Linux distribution's built-in wordlist (479 829 words)[12]. We applied some basic rules to mutate the base words provided by these lists, including reversing each word, appending digits, toggling case, "leetification" *etc.*. A sample of the passwords we were able to crack appears in Table 4. We were able to crack only one hash from the Stratfor.com leak, and three from the Rootkit.com leak, perhaps a reflection on the small proportion of account holders likely to be using PwdHash. We then focused our attention on the massive Linkedin.com leak, where we suspected we may encounter many professionals operating in the security space who could be using PwdHash. Although our source of Linkedin.com hashes contained 6 458 020 unique entries, approximately 45% of these had the first five characters zeroed-out. The typical analysis is that those are simple passwords which the original cracker had already cracked before swapping the remainder on the underground, perhaps for the purpose of distributing the cracking effort. This hypothesis is supported by the appearance of the "remainder" hash for low-hanging fruit such as "linkedin" amongst the zeroed-out entries. We only attacked the hashes which had not been zeroed out, believing them to be derived from more complex passwords, *i.e.* harder to crack.

We were able to immediately crack 75 PwdHash hashes in this list to reveal the master passwords with less than 30 seconds of cracking effort. We believe these hashes have not been cracked previously. None of the resolved passwords would be considered particularly strong if used conventionally (many appearing in common word lists), which suggests that PwdHash is imbuing a false sense of security in users, *i.e.* some PwdHash users believe they no longer have to consider password complexity. This is not really a problem with the PwdHash approach (they don't claim to solve this problem), but it does appear to be a problem with the way some people are using it. We observed that the password "linkedin" appears both amongst the regular MD5 hashes as well as the PwdHash obfuscated hashes, a poor choice in both instances.

The characteristics of the three leaked hash databases we tested, along with a summary of our results, can be seen in Table 5.

[12] `/usr/share/dict/words`

Table 4. A sample of the PwdHash master passwords discovered, along with some notes about their possible meaning.

Domain	**Leaked hash**	**Password**
Stratfor	`e9c0873319ec03157f3fbc81566ddaa5`	frogdog
Rootkit	`2261bac1dfe3edeac939552c0ca88f35`	zugang
Rootkit	`43679e624737a28e9093e33934c7440d`	ub2357
Rootkit	`dd70307400e1c910c714c66cda138434`	erpland
LinkedIn	`508c2195f51a6e70ce33c2919531909736426c6a`	5tgb6yhn
LinkedIn	`ed92efc65521fe5074d65897da554d0a629f9dc7`	Superman1938
LinkedIn	`5a9e7cc189fa6cf1dac2489c5b81c28a3eca8b72`	Fru1tc4k3
LinkedIn	`ba1c6d86860c1b0fa552cdb9602fdc9440d912d4`	meideprac01
LinkedIn	`fd08064094c29979ce0e1c751b090adaab1f7c34`	jose0849
LinkedIn	`5264d95e1dd41fcc1b60841dd3d9a37689e217f7`	linkedin

Password	**Notes**
frogdog	Slang name for a French Bulldog, also an alien species in the Star Wars universe.
zugang	German word for “access”.
ub2357	A prime number with some interesting properties.
erpland	Second studio album by British psychedelic rock band Ozric Tentacles.
5tgb6yhn	Common keyboard pattern.
Superman1938	The first Superman comic was printed in 1938.
Fru1tc4k3	1337 speak.
meideprac01	“carpe diem” spelt backwards.
jose0849	Possibly a DoB.
linkedin	!!!

Table 5. Summary of the leaked hash databases tested and results produced.

Domain	Stratfor.com	Rootkit.com	LinkedIn.com
Year of leak	2011	2011	2012
Hashes	822 657	58 677	2936840/ 6458020
Type	Unsalted MD5	Unsalted MD5	Unsalted SHA1
Attribution	“Anonymous”	“Anonymous”	Unknown
Previous best effort	93.73%	97.17%	96.61%
Cracking speed	42008.5 kH/s (11.40ms/H)	42193.8 kH/s (11.35ms/H)	39762.2 kH/s (14.88ms/H)
Cracking time	27s	27s	28s
Passwords recovered	1	3	75

Public data from `https://hashes.org/public.php`

5 Mitigation

Our results highlight weaknesses in the current PwdHash implementation, but they also allow us to identify areas for improvement for mitigating these weaknesses. First and foremost, it's clear that the strength of site-specific passwords generated by client-side hashing will be dependent on the strength of the master password used. Client-side hashing does not – in this form – turn a weak password into multiple strong passwords.

Having accepted this, there are several ways we would choose to improve the implementation. The current choice of HMAC-MD5 adds only a limited overhead to the cracking process, and in fact what's needed for this application is a key-derivation function rather than a hashing function. Suitable algorithms that intentionally increase computation and memory requirements can benefit from the natural asymmetry between the task of the user (performing a single hash occasionally) and the task of the attacker (performing a very large number of hashes over a short period of time). PBKDF2 [5] or Argon2 [1] (aimed at resisting GPU attacks) would therefore make more appropriate choices.

The use of a user-specific client-side salt (referred to as a *global password* in Ross *et al.* [9]) that's stored in the browser would also mitigate the attack, since it could dramatically increase the keyspace an attacker would need to search.

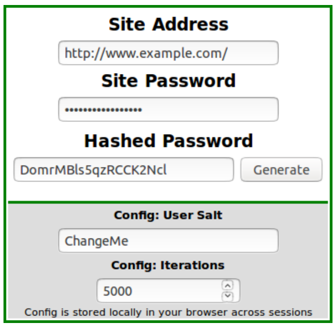
Both improvements are suggested in the original PwdHash paper, but are not implemented in the released version. One possible reason for this is that both impact the usability of the system. The current version uses no stored secrets, allowing the user to access the PwdHash page on any Javascript-enabled browser and immediately retrieve the password for a site. Using a user-specified salt would limit use to only those browsers where the salt had been stored. The relatively slow execution of a client-side Javascript implementation also limits the benefit to be gained from using a computationally-intensive hashing algorithm.

Finally the upper limit of 16-bytes entropy from the HMAC-MD5 is an unnecessary limitation of the implementation that could easily be relaxed. We have released the proof-of-concept update to PwdHash shown in Figure 5. This uses PBKDF2-SHA256 and addresses the other limitations described here[13]. However, it also highlights one of the difficulties with using a client-side hashing algorithm, since it's not compatible with the original. A user would therefore need to update all existing PwdHash passwords to use this new version to avoid having to use different versions on different sites.

6 Related Work

The vulnerability of client-side hashing to dictionary attacks is well known and was even highlighted in the original PwdHash paper [9]. More recently Das *et al.* briefly discussed the same issue in relation to PwdHash, stating that "An

[13] This updated version can be tested at `https://www.cl.cam.ac.uk/~dl551/pwdhash` and the code is available at `https://github.com/llewelld/pwdhash-poc`.


Site Address
http://www.example.com/
Site Password
••••••••••••••••
Hashed Password
DomrMBls5qzRCCK2Ncl

Config: User Salt
ChangeMe
Config: Iterations
5000
Config is stored locally in your browser across sessions

Fig. 5. The updated PwdHash proof-of-concept website mitigating the identified weaknesses.

attacker could still take a list of hashed passwords from one site and brute-force them (with the extra PwdHash round of hashing), then attempt to reuse recovered credentials at other sites (applying the PwdHash algorithm specific to the other site)." [2]. We have essentially demonstrated the effectiveness of this attack.

PwdHash is also not the only example of client-side hashing. Halderman *et al.* describe a similar approach, but with a focus on increasing the computation required to break the hash [3]. The output of iteratively hashing a concatenation of the user's master password and username is stored on the user's local machine. While the user only has to perform this once, an attacker must perform similar lengthy calculations for each of the breached accounts in a leak.

Both PwdHash and the Halderman algorithm have been extended in various ways. Yee and Sitaker's *Passpet* tool [18] includes a client-side password generator intended to mitigate phishing attacks. Their implementation is based on Halderman *et al.*'s approach. Reddy *et al.* [7] developed *PwdHash++*, which combines password hashing with a variety of other phishing countermeasures such as site-whitelisting.

Browser-based and commercial password managers such as LastPass [11] offer the most widely-used alternatives to client-side password hashing. Although they may not explicitly encourage password-reuse across sites, password managers still rely on end-users choosing (or generating) strong passwords. Stajano *et al.* propose the use of strong, totally random passwords of generous length that are generated and stored automatically for the user (so the user never sees the password) [13, 12] and provide similar characteristics to a public key. This addresses the danger posed by brute-force attacks, but at some loss of convenience for the user, who must rely on access to a password database to log in to accounts.

A recent paper by Ruddick *et al.* [10] considers different approaches for optimising hashing algorithms using OpenCL kernel code. Steube, one of the authors of hashcat, has also presented on the topic [14], providing details of both the initial-step and early-exit optimisations that our approach interacts with. Both provide helpful insight into optimising GPU-based hash algorithms, and with the release of hashcat as open source in December 2015, the code for these optimisations is also now easily accessible.

7 Conclusions and Further Work

The main takeaway we want to emphasise from this work is that PwdHash is only as secure as the master password it's used with. The authors of PwdHash make clear the dangers of dictionary attacks on the master password, but the solutions they proposed were not implemented in the released version. Moreover, our results suggest that the message hasn't been absorbed by PwdHash users, many of whom are using weak master passwords. Although the generated site-specific passwords *appear* strong, they are in fact no stronger than the original master password.

We provide an implementation that mitigates some of the vulnerabilities of the original PwdHash implementation, using the PBKDF2 key extension algorithm, fixing the effective 22 character password limit and implementing the global password (user-specified salt) described in the original paper.

We believe our hashcat extension can be optimised to yield better performance, especially in the case of HMAC-MD5 where the reason for the significant decrease in speed is unclear. In our future work we hope to look at this, as well as performing a more detailed review of our proposed mitigations. We are particularly interested to establish the extent to which a client-side Javascript implementation of Argon2 can be used to counter an offline brute-force attack on a leaked database of hashes.

Acknowledgements. The authors thank Frank Stajano and the anonymous reviewers for their helpful discussion and comments on the paper. David Llewellyn-Jones thanks the European Research Council for funding this research through grant StG 307224 (Pico). Graham Rymer thanks the Cabinet Office/OCSIA for their financial support.

A Mangle Code

The following code represents a conversion of the PwdHash algorithm to run on a GPU using OpenCL. This code is also available on GitHub (`https://github.com/llewelld/hashcat`).

```
// Domain to use for mangling
__constant u8 domain[] = "linkedin.com";
__constant u32x domain_len = 12;

// Perform the mangle operation on the string to be hashed
// w0, w1 - the string to be mangled
// in_len - the length of the string to be mangled
```

```
u32x mangle (u32x w0[4], u32x w1[4], const u32x in_len) {
  u32x out_len = in_len;
  u32 digest[4];  u32 data[8];  u32 hash[8];
  u32 i;  u32 size;  u32 extrasize;
  u32 startingsize;  bool nonalphanumeric;
  u32 extrapos;  u8 next;

  hash[0] = w0[0];  hash[1] = w0[1];
  hash[2] = w0[2];  hash[3] = w0[3];
  hash[4] = w1[0];  hash[5] = w1[1];
  hash[6] = w1[2];  hash[7] = w1[3];

  // HMAC-MD5 the domain name using the password as the key
  md5hmac_domain_mangle ((u8 *) hash, in_len, (u8 *) digest);

  // Check whether the original password contains non-alphanumeric values
  nonalphanumeric = containsnonalphanumeric_mangle ((u8 *) hash, in_len);

  w0[0] = digest[0];  w0[1] = digest[1];
  w0[2] = digest[2];  w0[3] = digest[3];
  w1[0] = 0;  w1[1] = 0;  w1[2] = 0;  w1[3] = 0;

  // Base64 encode the HMAC; this will be used to generate the password
  out_len = b64_encode_mangle ((u8 *) hash, 16, (u8 *) w0);

  // b64 encoding will produce 24 bytes output, but last two will be "=="
  out_len = 22; extrasize = 22; size = in_len + 2;

  startingsize = size - 4;
  startingsize = (startingsize < extrasize) ? startingsize : extrasize;

  // Transfer the intial portion for output
  for (i = 0; i < startingsize; i++) {
    ((u8 *) data)[i] = ((u8 *) hash)[i];
  }
  for (i = startingsize; i < 32; i++) {
    ((u8 *) data)[i] = 0;
  }
  extrapos = startingsize;

  // Add extra capital letter
  next = (extrapos < extrasize) ? ((u8 *) hash)[extrapos] : 0;
  extrapos++;
  if (!contains_mangle ((u8 *) data, startingsize, 'A', 'Z')) {
    next = 'A' + (next % ('Z' - 'A' + 1));
  }
  ((u8 *) data)[startingsize] = next;
  startingsize++;

  // Add extra lower case letter
```

```
next = (extrapos < extrasize) ? ((u8 *) hash)[extrapos] : 0;
extrapos++;
if (!contains_mangle ((u8 *) data, startingsize, 'a', 'z')) {
  next = 'a' + (next % ('z' - 'a' + 1));
}
((u8 *) data)[startingsize] = next;
startingsize++;

// Add extra number
next = (extrapos < extrasize) ? ((u8 *) hash)[extrapos] : 0;
extrapos++;
if (!contains_mangle ((u8 *) data, startingsize, '0', '9')) {
  next = '0' + (next % ('9' - '0' + 1));
}
((u8 *) data)[startingsize] = next;
startingsize++;

// Add extra non alphanumeric
if (containsnonalphanumeric_mangle ((u8 *) data, startingsize)
  && nonalphanumeric) {
  next = (extrapos < extrasize) ? ((u8 *) hash)[extrapos] : 0;
  extrapos++;
}
else {
  next = '+';
}
((u8 *) data)[startingsize] = next;
startingsize++;

// If there's no alphanumeric values in the original password
// remove them from the result
if (!nonalphanumeric) {
  for (i = 0; i < startingsize; i++) {
    if (containsnonalphanumeric_mangle (((u8 *) data) + i, 1)) {
      next = (extrapos < extrasize) ? ((u8 *) hash)[extrapos] : 0;
      extrapos++;
      next = 'A' + (next % ('Z' - 'A' + 1));
      ((u8 *) data)[i] = next;
    }
  }
}

// Rotate the result to randomise where the non-alphanumerics are
next = (extrapos < extrasize) ? ((u8 *) hash)[extrapos] : 0;
rotate_string_mangle ((u8 *) data, startingsize, next);
((u8 *) data)[startingsize] = 0;
out_len = startingsize;

// Copy the result into the output buffer
w0[0] = data[0];  w0[1] = data[1];
```

```
  w0[2] = data[2];  w0[3] = data[3];
  w1[0] = data[4];  w1[1] = data[5];
  w1[2] = data[6];  w1[3] = data[7];

  return (out_len);
}
```

B List of Discovered Hashes

The following shows the full list of cracked hashses with master passwords.

Stratfor.com

```
e9c0873319ec03157f3fbc81566ddaa5:frogdog
```

Trivia: `frogdog` is a slang name for a French bulldog, also an alien species in the Star Wars universe.

Rootkit.com

```
2261bac1dfe3edeac939552c0ca88f35:zugang
43679e624737a28e9093e33934c7440d:ub2357
dd70307400e1c910c714c66cda138434:erplan
```

Trivia: `zugang` is German for "access"; `ub2357` is a prime number with some interesting properties, also a minor planet designation of one particular Jupiter Trojan; `erpland` is the second studio album of British psychedelic rock band Ozric Tentacles.

LinkedIn.com

```
013c6c38365f013ddd523adf5e048480227687ec:frogdog
065f39f31fe3e8ec6f973edc2071f278f327d84c:albinos
0cda088dc06a5373a3b700454b0eb6a885983fb3:fozzie13
0d5852a79a0b739235c329c0b1f6b649bbf73d73:lakee
11ceb5cbadfa201d57d485fc316bbd7ebd7a54a9:pissoff
1b7a9350d0caec8c6796043d30c4ac761bdac2d3:panda179
1ffadeba7773f4eb4a494f7aee8d4353f6a09cf1:fuzzyness
20dfab0200df0fc7c280d0e51307a9e0ec8f9fa4:abababab
23ddf01341a03755149c5da433d6df17b7049892:gorgias
25309bbd07529618ed70fa2f7ea8390b02a0e8c0:born2be
2a1542b0c8a62c2374cf6112eb750edeaa997dd8:phphph
2a41e985df8b697b4182c9705ec249995daee83c:roscoe02
2d714f7a9a7ddd26f104231b589eef57638ed533:zaphod00
34e2377e0bcbf5db4bf564a2ba6345f4e6cbd24d:sweets
361e36a3cf0f27f499e790792640590e68682ff3:oldking
```

```
42df0c4dd7af01f08cead26ca9b43a3ffdb4cc8f:brandy1
44ef1e52ff3a25521581d59b906220a89913a5d9:psikopat
48b09b1dae6b32f6d1cd260320fbaa2b35976670:katari
4a26da695097044bd5eacec3b9eef4ad44cabe9f:p00ki3
4abe8e3dcc110cd5905388e3d6bc09ea113162dc:tytus123
4bbfdd7efd11767a448f1ce361da914ad0503485:x-file
508c2195f51a6e70ce33c2919531909736426c6a:5tgb6yhn
52bc94618a819344309c4eeace0736586df03940:sodapop
5a82cfbc3abdce6058eaac6f066a3648a1bc23c9:eatme21
5a9e7cc189fa6cf1dac2489c5b81c28a3eca8b72:Fru1tc4k3
5b8038536d57b98eaf60dc6881f2efd86cc7a6b6:hastalavista
5be537b7fcc3f33ab694b7c0e17c4216ff5dc0ab:threnody
5c78dc41481dbe25e83b4d3cb6eb7bbb46244486:orwell13
5e3a3b67ccb3b69b6ec4fecb6d42dab03aed55b2:strider09
6103143d952b6e526e00b7b39d3d718b76c7d4d8:egghead
64769f9672c354536add3ed25b84b0415cef88db:pitchfork
692e23e2a35c3d610554e096f89c06ceaf699cb5:mono300
6e6b4e1dff925d7222de02cc46937f6b895d7d60:bilbo1
6ee8fae6e0904f7d6d676a1e0c95753024775ec8:ShaDow
6f1e96bbad3bf5db4110c39d2b7592274d748cda:midnight
79b73acb576701048ddd7919516ab0c5707b32d8:kakamaka
7da1e59dd4ecbcba4b98c5d0f14329c03b4ac8eb:bonsai
82e393197b276c2c3d32a13e986ea6b53fdd4ee6:dotmatrix
84a4e026fd3fba1beb613f2f06bbc066d2beb403:fishes
8a8181a463e269c317fc8250068527ee07e5a800:simple
8d70b23f0405b6a46178781ab7a51288ce03d0fd:saturn
8e332b9555ecf2ef094252457541cfc4b48b2b91:qwertyou
90c67311ec0c318604a1118c49966a4692e8b110:hanker
90f57edc2400e34ea379d7c9f5e0267eae00f46a:nopass
970a8e31a082faa0e64700b9449a2adfd2675e8c:hirens
98b6dc2ddfd4c710eb1b14d49519fedd1c80c3a4:poohsticks
9b66fa54dc9bf6b06a1d3466f866a0ebd106796d:bkg123
a0a0c99be8fb9be1465419ffd6b72fb496a0d391:bredin
a0d274a14386f253de69442652269ba5bc1e02f7:tadeusz
a1e5602049a8725c276d5627f53f193dfd4f27f5:cyborg
a9cb6713c09edfee36d8fbd634ba44b794bf2229:simpson
b875e2c11c251a932c3a89785ceec32a3db30f11:dedalus
b9482e0e24186321bcf8f0d22c990e4803a3b37e:hulabula
ba1c6d86860c1b0fa552cdb9602fdc9440d912d4:meideprac01
ba536f61ca60ec0730f5f716d6dd0e18f8d2577c:gallia
bb664d001f1ccbdb920c5eec4a573008690236d2:123cover
bf077dfc91a4a9b914fab757ec6889c1742f1473:azazello
c083787892bdf1241c3beef86ad7369a939263f3:vth0kies
c321cc91176f721b933529ae8014216eb9fb8660:ch0c0l0c0
c748b0d8344800a9e432e9da75f60163eb302d78:mypass3
```

```
cef0f2f65ec74f7f412fde1e525ee6c337c3cec3:hertz1
d21ccd5ee99961c0c609c37596c29dc3b16483cb:f3rrar1
d44e6eee9560e6056454b0f6d7a48533069b9c12:caroline
d625c19891d6b49877d161a4d7bdef4ffcc726ce:sandworm
dc42cb9c6ea3fbca4ae4e04c5ba0796a302bd7c4:wooly11
e763ce9548d1fe459076ba7f6575da2f1a01ed0a:thunder
ec6333d07b7fea7e8030b14fa727ab7c864344e5:abidali
ed92efc65521fe5074d65897da554d0a629f9dc7:Superman1938
f15aed752c44b3b07cac23e2c285be8382307cc3:muthos
f2788c12bfa27c3c36e4085f8dfa8b01a5499801:spass123
f6ae0634423c19b02b4afdd805c3293ce9231785:fkv123
fc5622540175039fb1fe8d20172d411afbcdbd3b:gremlin56
fd08064094c29979ce0e1c751b090adaab1f7c34:jose0849
fd708fbbea7529918e58af9f5674e5d40337ca4f:whatuwant
fdc4e06a607c7da9f692f12f08e89702284284f3:malachi
```

Additional cracking on LinkedIn.com (toggle attack + domain):

```
2931c5175d7f7f6e90f2bd707d9eaa767a86674f:DogHeaven
5264d95e1dd41fcc1b60841dd3d9a37689e217f7:linkedin
b3917ad1d7bd2ff3975bdf0eaa60696fe5037088:Beaujeu
```

Trivia: Zaphod Beeblebrox (`zaphod00`) is a character in Hitchiker's Guide to the Galaxy; `psikopat` is Turkish and Indonesian for psychopath; `5tgb6yhn` is a common keyboard pattern; `Fru1tc4k3` is 1337 speak; `threnody` is a Marvel Comics character; `azazello` is a character from the novel *The Master and Margarita* by the Russian writer Mikhail Bulgakov; the first Superman comic was printed in 1938 (`Superman1938`); `muthos` is Greek for story; `jose0849` is likey a DoB.

References

1. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: New generation of memory-hard functions for password hashing and other applications. In: 2016 IEEE European Symposium on Security and Privacy (EuroS P). pp. 292–302 (March 2016)
2. Das, A., Bonneau, J., Caesar, M., Borisov, N., Wang, X.: The tangled web of password reuse. In: NDSS. vol. 14, pp. 23–26 (2014)
3. Halderman, J.A., Waters, B., Felten, E.W.: A Convenient Method for Securely Managing Passwords, p. 471479. WWW 05, ACM (2005), http://doi.acm.org/10.1145/1060745.1060815
4. Horsch, M., Schlipf, M., Braun, J., Buchmann, J.: Password Requirements Markup Language, pp. 426–439. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-40253-6_26
5. Kaliski, B.: PKCS #5: Password-based cryptography specification version 2.0. IETF (Sep 2000), https://tools.ietf.org/rfc/rfc2898.txt, RFC2898
6. Morris, R., Thompson, K.: Password security: A case history. Commun. ACM 22(11), 594–597 (Nov 1979), http://doi.acm.org/10.1145/359168.359172

7. Reddy, V.P., Radha, V., Jindal, M.: Client side protection from phishing attack. International Journal of Advanced Engineering Sciences and Technologies (IJAEST) 3(1), 3945 (2011)
8. Ross, B., Jackson, C., Miyake, N., Boneh, D., Mitchell, J.C.: Pwdhash, https://www.pwdhash.com/
9. Ross, B., Jackson, C., Miyake, N., Boneh, D., Mitchell, J.C.: Stronger Password Authentication Using Browser Extensions. In: 14th USENIX Security Symposium (2005), http://crypto.stanford.edu/PwdHash/
10. Ruddick, A., Yan, J.: Acceleration attacks on pbkdf2: Or, what is inside the black-box of oclhashcat? In: 10th USENIX Workshop on Offensive Technologies (WOOT 16). USENIX Association, Austin, TX (2016), https://www.usenix.org/conference/woot16/workshop-program/presentation/ruddick
11. Silver, D., Jana, S., Boneh, D., Chen, E., Jackson, C.: Password Managers: Attacks and Defenses, p. 449464. USENIX Association (2014), https://www.usenix.org/node/184476
12. Stajano, F., Christianson, B., Lomas, M., Jenkinson, G., Payne, J., Spencer, M., Stafford-Fraser, Q.: Pico Without Public Keys, p. 195211. Lecture Notes in Computer Science, Springer International Publishing (Mar 2015), http://link.springer.com/chapter/10.1007/978-3-319-26096-9_21
13. Stajano, F., Spencer, M., Jenkinson, G., Stafford-Fraser, Q.: Password-Manager Friendly (PMF): Semantic Annotations to Improve the Effectiveness of Password Managers, p. 6173. Lecture Notes in Computer Science, Springer International Publishing (2015), http://link.springer.com/chapter/10.1007/978-3-319-24192-0_4
14. Steube, J.: Optimizing computation of hash-algorithms as an attacker. Passwords13 Las Vegas (Jul 2013), https://hashcat.net/events/p13/js-ocohaaaa.pdf
15. Stobert, E., Biddle, R.: The password life cycle: User behaviour in managing passwords. In: Symposium On Usable Privacy and Security (SOUPS 2014). pp. 243–255. USENIX Association, Menlo Park, CA (2014), https://www.usenix.org/conference/soups2014/proceedings/presentation/stobert
16. Stobert, E., Biddle, R.: Expert Password Management, pp. 3–20. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-29938-9_1
17. Wright, J.: 2 years of @dumpmon (May 2015), http://jordan-wright.com/blog/2015/05/26/two-years-of-at-dumpmon/
18. Yee, K.P., Sitaker, K.: Passpet: Convenient Password Management and Phishing Protection, p. 3243. SOUPS 06, ACM (2006), http://doi.acm.org/10.1145/1143120.1143126