The cost of having been pwned: a security service provider's perspective

Gergely Biczók[1], Máté Horváth[1], Szilveszter Szebeni[2], István Lám[2], and Levente Buttyán[1]

[1] CrySyS Lab, Budapest Univ. of Technology and Economics
{biczok,mhorvath,buttyan}@crysys.hu
[2] Tresorit
{szebeni,lam}@tresorit.com

Abstract. Account information from major online providers are getting exposed regularly; this gives rise to PWND services, providing a smart means to check whether a password or username/password tuple has already been leaked, rendering them "pwned" and therefore risky to use. However, state-of-the-art PWND mechanisms leak some information themselves. In this paper, we investigate how this minimal leaked information can speed up password cracking attacks of a powerful adversary, when the PWND mechanism is implemented on-premise by a service provider as an additional security measure during registration or password change. We analyze the costs and practicality of these attacks, and investigate simple mitigation techniques. We show that implementing a PWND mechanism can be beneficial, especially for security-focused service providers, but proper care needs to be taken. We also discuss behavioral factors to consider when deploying PWND services.

Keywords: pwned accounts, credential leakage, leaked password, dictionary attack, data breach, security service provider

1 Introduction

Data breaches at major online service providers exposing account information are increasingly prevalent. As a result of these breaches, many millions of user accounts, including usernames and passwords, have been *"pwned"*. In fact, as of June 2020, the number of pwned accounts is reported to be around 9.7 billion [6]; this number is based on already discovered incidents, therefore it should be treated as a a conservative lower bound. Such data leaks and the challenges they have been posing have not gone unanswered by the security community. Privately and publicly curated databases containing records of pwned accounts have emerged, providing a means to check if a given account (email address or email address/password tuple) is affected. Prominent examples include haveibeenpwned.com [6], SpyCloud [18] and Google's own database [20]. Services based on these repositories implement privacy-preserving mechanisms,

which we term as *PWND mechanisms*, enabling their users (both end-users and service providers) to check whether certain passwords or full login credentials are already known to be leaked without revealing them to the service.

A security service provider's perspective. Online service providers are able to check whether a user registering to their service is re-using a pwned password; this could improve account security and avoid potential reputation loss stemming from account compromise such as with iCloud[1]. Similarly, providers could use the same PWND mechanism for password change and logins of existing users to protect against the usage of a leaked password. We emphasize that here the goal of such checkup is not to alert users when their credentials become known to be leaked, but rather to avoid the potential reuse of a leaked password, regardless who used the password when it was leaked. Consequently, *we focus on checking passwords only*, to make pwned password lists useless in dictionary attacks. (Of course, incorporating the PWND mechanism is not enough; providers should take extra care with how they present the results of such checks to the user [5]). Providers of security services, such as end-to-end encrypted file storage or secure collaboration, are under even more scrutiny regarding incidents involving user accounts; hence, they have a strong incentive to implement PWND measures. However, not all PWND integration options [10,20] are equal in their potential appeal to such a provider. We argue that an on-premise PWND solution, built on downloadable public databases and run by the provider itself, might be preferable. Such a solution i) does not require trusting an explicit third party ii) works also with desktop and mobile apps, not just with browser-based service access iii) gives the control to the service provider, and requires no user action.

Information leakage. Proper PWND services obviously never require the client to send its cleartext password, moreso, they do not require the full (e.g., SHA-1) hash either; that would provide too much information and allow an adversary to reconstruct the original password. Instead, only the leading few [3] bytes of the password hash is sent upstream that enables the server to send down a filtered (therefore much smaller) version of the list of pwned password hashes. Then, the client can check locally whether the hash of his own password is in the received pwned password hash list. Such filtering greatly accelerates the checking process, however, it also leaks some information [10]. Obviously, a few bytes of the password hash is not enough to be used directly for reversing the original password, but it can accelerate password cracking.

Strong adversary and expedited attacks. When considering security services (e.g., secure storage), from the user's perspective *the service provider is the strongest potential adversary* who claims protection even from itself. In the corresponding security model the adversary obtains every bit of leaked information related to a specific user; such a model has not yet been analyzed in the context of PWND services in related work [10,20]. Such an adversary can launch a sped-up password cracking attack: given a targeted user he computes a sim-

[1] https://mashable.com/2014/08/31/celebrity-nude-photo-hack/

[3] e.g., 5 half-bytes for haveibeenpwned.com and 2 bytes for Google's Password Checkup

ple SHA-1 hash of the password candidate; if the first few bytes do not match with the leaked bytes, the adversary can skip to the next password. This can accelerate the cracking process by a significant factor, in the order of thousands or millions depending on the actual extent of information leakage. Variants of this idea can be used both for dictionary and brute force attacks; furthermore, a non-targeted brute force attack could also be carried out with increased speed. Note that such an adversary is not slowed down by rate limiting API calls.

Our contribution. In this paper, focusing on PWND mechanisms using only password hashes, we analyze expedited dictionary and brute force attacks by a strong adversary against users of an online service provider utilizing an on-premise PWND mechanism. We characterize attack costs for each attack and password strength category, and take a look at simple potential countermeasures. Our quantitative results indicate that i) public PWND databases can be used as dictionaries for password cracking attacks, and ii) the proposed hash stretching indeed renders brute force attacks on "medium-strength" passwords impractical. Furthermore, we discuss how user behavior and the composition of a given service provider's user base affects the introduction of a PWND mechanism from a cost-benefit standpoint. Finally, we discuss why a PWND solution based on Private Set Membership (PSM) protocols is not (yet) practical.

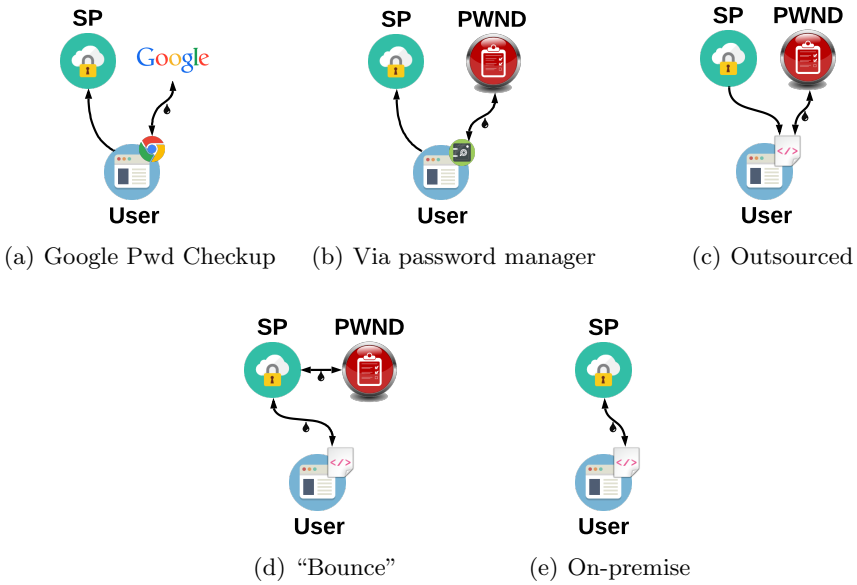
The rest of the paper is organized as follows. Section 2 describes potential PWND implementation alternatives for online service providers and the details of the most popular PWND mechanisms. Section 3 introduces and analyzes the potential attacks in detail and discusses simple mitigation techniques. Section 4 takes a look at the expected costs of the above attacks and the trade-off associated with deploying a PWND mechanism with regard to both service providers and end-users. Section 5 discusses the limitations of a potential PWND mechanism based on PSM protocols. Finally, Section 6 concludes the paper.

2 Background

Here we introduce the most important concepts regarding PWND services and position our contributions with respect to related work.

2.1 PWND: architectural alternatives

From the service provider's aspect, a PWND service can be realized in a variety of ways. Fig. 1(a) and 1(b) represent solutions where checking is initiated by the user (e.g., by using a specific browser (extension) [20] or password manager [7] that alerts in case of a password becoming leaked). While these solutions have their own merits, they also have inherent weaknesses. In these scenarios, the service provider has no influence over whether the user has checked his candidate password against a PWND database. The provider may display a recommendation in its registration dialogue, and nudge the user towards checking, but nothing more. Therefore, the service provider relies on the voluntary actions of the user and either risks account compromise, or is forced to also implement



(a) Google Pwd Checkup (b) Via password manager (c) Outsourced

(d) "Bounce" (e) On-premise

Fig. 1. PWND architectural alternatives: initiated either by the user or by the service provider

his own PWND mechanism. Furthermore, if the service requires a desktop or mobile app for client access, browser extensions and non-system level password managers will not work. Last, but not least, a major issue is trust: a provider with mostly security-conscious users (whether end-users or enterprises) will find it hard to justify opting for a PWND mechanism administered by a third party.

Contrarily, Fig. 1(c), 1(d) and 1(e) depict checking mechanisms triggered by the service provider. In such cases, the control is with the service provider, so that it can enforce PWND, requiring no action from the user. This can be especially crucial for providers of security-related services, such as end-to-end encrypted file storage, online virus checking, secure collaboration, etc. Also, requiring no extra functionality at the client side, these solutions handle client heterogeneity inherently. The outsourced and "Bounce" [17] alternatives still suffer from the third party trust issue mentioned above. In addition, the outsourced case presents a sizable challenge in terms of API keys to the PWND service: since PWND-related data do not flow through the service provider, the API is either public and free (which is shown to be problematic[4]), or utilizes an API key for authentication (and usually charges a fee for API usage). In the latter case, the outsourced PWND architecture requires the sharing of the API key of the service provider with all its users; although this is doable, it might result in complex policies at the service provider's side and does not fall under best

[4] https://www.troyhunt.com/authentication-and-the-have-i-been-pwned-api/

security practice. Furthermore, assuming an adversary who sees everything the service provider sees, the "Bounce" architecture leaks just as much information as the on-premise solution, but with adding the trust issue with a third party PWND service. Based on the above, we argue that an on-premise PWND mechanism is a sensible choice for providers of security-related services and/or with security-conscious users. In fact, GitHub implements the on-premise model [11].

2.2 *K*-Anonymity Preserving Method

We say that some data has K-anonymity property [15] if the information about any entry, contained in the data cannot be distinguished from at least $K-1$ other entries. [1] used this type of privacy guarantee to protect the process of testing password leakage with the algorithm in Fig. 2.

We recall the most notable instantiations of the idea of [1]. The first one is the haveibeenpwned.com[5] website, which uses the SHA-1 hash function that has an output of 40 hexadecimal digits (thus $m = 160$) out of which a 5 digit long prefix ($n = 20$) is used for the partitioning of the database. In Google's Password Checkup Chrome extension, the same K-anonymity based method guarantees the privacy of user credentials (see subsection 2.3 for differences). The extension relies on the Argon2 hash function with $m = 128$ bit output length and a configuration using a single thread, 256 MB of memory, and a time cost of three [20]. This results in a computationally expensive calculation (modelled by an inefficient oracle) that is unnoticeable for honest users performing a single query but causes significant slowdown for brute force attackers (see details in Section 3.3). The prefix length, sent by the client, is 2 bytes ($n = 16$).

Let $H : \{0,1\}^* \to \{0,1\}^m$ be a hash function and L a database of leaked passwords.
Input / Output:
Server: A list L of leaked passwords / -
Client: Password p^* to be checked / {0,1}
Protocol:

Offline phase For all $p^i \in L$, the server computes $H(p^i) = h^i$ and organizes the resulting hash values into 2^n sets based on their length n prefixes i.e., $S_{\mathsf{pref}^j} = \{h^i | h^i_\ell = \mathsf{pref}^j_\ell \text{ for } \ell = 0, \ldots, n-1\}$.
Online phase :
1. Client computes $H(p^*) = h^*$ and sends pref^*, its first n bits to the Server.
2. Based on pref^*, the Server sends S_{pref^*} to the Client.
3. The Client checks if $h^* \in S_{\mathsf{pref}^*}$. If yes, the password is a leaked one and outputs 1, otherwise $p^* \notin L$ and returns 0.

Fig. 2. K-anonymity preserving protocol for checking whether a password has already been leaked (for $K = 2^{m-n}$).

[5] For details: https://haveibeenpwned.com/API/v2#PwnedPasswords

2.3 Related Work

In February 2019, Google has introduced a Chrome extension[6], called Password Checkup, with the goal of enabling users to check whether their credentials were part of a former data breach. The extension checks possible data leakage in a privacy-preserving manner, whenever the user registers or logs in to a third party service. While Password Checkup offers a service similar to the one we are considering in this paper, it is important to notice the essential differences in their objectives. Password Checkup handles usernames and passwords together aiming to alert the user only if she or he was involved in a data breach [20]. Contrarily, we are not interested in detecting whether a specific user is affected by any of the breaches. Our goal is to make users avoid the use of leaked passwords regardless of where, when and who used the specific password. Our motivation for avoiding leaked passwords, even for different usernames, roots in the considered scenario. In case of security-focused services, like the end-to-end encrypted file sharing service provided by Tresorit[7] or the private-by-design collaboration platform CryptPad[8], the service provider often wants to prove that their users are not only protected from outside attackers but also from inside attacks (i.e. from the service provider itself). While in the former case dictionary and brute force attacks can be easily prevented, e.g. by limiting the number of failed log in attempts, this is not straightforward when the attacker is the service provider itself (or, equivalently, bears the entire knowledge of the service provider). Assuming that users are exposed in such a manner, it becomes important to protect against attacks that make use of leaked data, e.g., by building a dictionary from it. In this paper, we focus on this scenario by choosing to protect users who would reuse already leaked passwords (even under different usernames) [19].

Furthermore, as we argued in Section 2.1, an on-premise PWND solution, built on downloadable public databases and run by the provider itself, could be the preferred implementation alternative to providers of security-focused services. Such a scenario has not been analyzed before with respect to information leakage [10] Another difference between our work and [20] is that we consider the use of publicly available databases of leaked passwords; obtaining access to a private database either costs significant money [18] or is just plain unlikely to happen [20]). Google uses its own database, consequently, their attacker model is stricter than ours as it also aims to protect the database besides the user's credentials during the checkup.

3 PWND: Security Issues and Mitigation Techniques

3.1 Scenario description and assumptions

Our investigation is motivated by the applicability of a PWND mechanism by Tresorit, an end-to-end encrypted cloud data storage, syncing and sharing service

[6] https://chrome.google.com/webstore/detail/password-checkup-extensio/pncabnpcffmalkkjpajodfhijclecjno
[7] https://tresorit.com
[8] https://cryptpad.fr

provider. In the Tresorit system [21], user passwords are stretched and used as input for encrypting randomly generated private keys with which all user-related data is encrypted in an end-to-end manner. Neither unencrypted user data, nor user passwords, nor private keys leave the devices of users at any time. Given that Tresorit sells a value-added storage service with a distinct focus on security, in line with our argument is Section 2.1, we assume that the PWND service runs directly at Tresorit following the on-premise model (see Fig. 1(e)). Tresorit's own PWND database can be built by obtaining the SHA-1 hashes of pwned passwords from the haveibeenpwned.com service database and/or compiling their own from various sources. Naturally, the PWND database is updated continuously.

We assume an incremental roll-out of the PWND mechanism: already registered users are only checked when their session expires and they have to log in with their password again; before doing so, Tresorit does not have extra information about their password hashes. Since Tresorit does not store password hashes, offline checking of existing users, and selectively triggering password change for pwned accounts is technically infeasible. Triggering session expiration for the whole existing user base upon the introduction of PWND is not a realistic option, because Tresorit users expect their data to be continually synced with the service. Of course, users can change their passwords voluntarily; in such a case, the PWND mechanism could be triggered.

Note, that in case of finding a pwned password, the service provider forces the user to choose another password. We also assume that weak passwords are rejected during registration via a combination of rule-based and client-side (small) dictionary based filtering; in this case the use is prompted for another password of adequate strength. Note that we assume no mandatory 2-factor authentication for user login [13].

3.2 Potential attacks

Attacker model. Motivated by the service provider's point of view, we assume a powerful adversary. Such an adversary is assumed to break into the service provider's system gaining access to all the information stored at and communicated by the provider. In this context, the adversary has the same capabilities as the service provider. Even though the adversary has the same access to resources as the service provider, neither can access user data due to the end-to-end encryption (see Section 3.1). The adversary's aim is to crack user passwords in order to get access to user accounts and the data stored there. It is worth noting that any mitigation technique proposed for the attacks below should make user passwords and data more secure against both the adversary *and* the service provider. We also assume that the attacker does have access to public PWND databases (even if the service provider does not implement any PWND mechanism). Note that traditional API security measures, such as account locking and rate limiting, do not protect against such an adversary, as the attacker is equivalent to an insider.

There are three types of password cracking attacks which can benefit directly from the extra information revealed by the PWND mechanism. The first two

attacks are sped up by filtering out potential password candidates quickly, while the third attack is accelerated by shrinking the set of users who can potentially own a given candidate password. Note that only users who registered or changed their passwords after the roll-out of the PWND service are affected.

Dictionary attack. The objective of this attack is to crack the password of a given individual user and gain access to the data stored at her account. The dictionary attack uses an existing list of popular passwords which the adversary tries systematically; e.g., the PWND database itself or the list of the 10 million most popular passwords[9]. In our case, the adversary observes the top-n bits of the hash. Based on these the adversary filters out candidate passwords (starting with the top-n bits) from its dictionary. It then tests the candidate passwords by stretching them and trying to decrypt the encrypted private key of the user according to Tresorit's process [21]. Once done, the adversary tries to decrypt the given user's data stored at Tresorit with the derived candidate private key. If this attempt is successful, i.e., it yields meaningful data, then the password is valid[10]. The extra information from the PWND mechanism accelerates this attack. The acceleration factor depends on the composition of the dictionary; assuming uniformity across the top-n bits, the speedup is by a factor of 2^n. Note that only users, whose password appeared in the list of pwned passwords *after* they had registered (following an update of the PWND database), are affected. (Obviously, their password would have been rejected during registration otherwise.)

Brute force attack A. This attack is similar to the dictionary attack both in its objective and speedup factor. The only difference is that there is no pre-computed database of candidate passwords: the adversary generates the candidate password on-the-fly using a structured approach (aided by tools such as John the Ripper[11]). This attack can also be accelerated by discarding potential candidate passwords quickly by hashing the password and calculating its top-n bits and comparing it to the leaked bit of the particular user. If the top-n bits match, the attacker can test the password which is more expensive. Ignoring the small cost of the top-n bit test leads to an acceleration factor of 2^n.

Brute force attack B. This attack is not targeted: the objective here is to crack the password of an arbitrary user or group of users. The adversary partitions the users into 2^n groups based on the leaked top-n bits. When trying a password candidate the adversary looks up the top-n bits and tries the candidate only for users in the given group. The extra information from the PWND mechanism accelerates this attack to the realm of being practical for non-pwned passwords which are neither particularly weak nor particularly strong[12]. The acceleration factor depends on the distribution of actual end-users across the top-n bits; assuming uniformity of users across top-n bit induced groups and at least 2^n affected users, this method accelerates password cracking by a factor of 2^n.

[9] https://github.com/danielmiessler/SecLists/tree/master/Passwords
[10] all three attacks follow this logic for checking if a password candidate is valid
[11] https://www.openwall.com/john/
[12] e.g., 8-character extended alphanumeric passwords

Note, that rainbow table based attacks [12] are not accelerated in this context. Rainbow tables are based on pre-computed chains of available password candidates and hashes, however, the PWND service eliminates those. If the user's registration password is pwned, i.e., the service provider's PWND database contains it, the service provider forces the user to choose another password. If the password is not pwned, there is no complete leaked hash value to build the rainbow table on. The top-n bits of the password hash, leaked by the PWND mechanism, do not help enough in the creation of the rainbow table, as the non-leaked part is still dominant.

3.3 Mitigation

Stretching. A simple way to prevent the acceleration of potential attacks is to use basic key stretching [8]. The user computes a stretched hash, iterating over a hash several thousand times, i.e., taking the output hash and feeding it back into the algorithm as an input. Such a stretched hash takes a set amount of time to complete, which will be several thousand times larger than for single iteration. Afterwards, the user sends the top-n bit of the stretched hash to the on-premise PWND service, and the rest of the PWND protocol continues according to Fig. 2. As such, the privacy-preserving manner of the password checking process is conserved.

On the one hand, this slowdown is easily tolerable for both end-users and service provider, as it is done in a one-off manner when registering for the service or changing passwords; also, calculating such a stretched hash takes typically from a few hundred milliseconds to a few seconds. On the other hand, the attacker suffers as he has to compute the same stretched hash for every step of the dictionary and brute force attacks. In fact, stretching gives the service provider a flexible "pacing" tool that can be adjusted as its system (number of users, PWND mechanism, etc.) evolves. Note that since rainbow table attacks are not accelerated in our application scenario, salting is not needed.

Increasing anonymity set size. Recall that the reason of leaking information during the checkup is to keep the overhead of the protocol unnoticeable for the user. Consequently, the choice of protocol parameters is a trade-off between password privacy and user experience. In order to minimize the number of false positives, the probability of hash collisions should be kept minimal (when modeling the hash function with a random oracle, it means that its output length should be long enough), so that different passwords can be represented with different hash values. The prefix length determines the size of the anonymity set that belongs to a given password and also the number of hash values the server has to transmit to the user to detect password leakage (the shorter the prefix is, the bigger the anonymity set and the communication cost). One could increase the anonymity set size either by increasing false positive probability (applying shorter hash values) or by increasing the protocol overhead. As the latter one is undesirable, we investigate the other opportunity. Shortening the leaked hash prefix with n' bits reduces the attackers advantage with a factor of $2^{n'}$ by increasing the anonymity set size with the same factor. To preserve

the original communication costs the server can either use a hash function with shorter output length or does not change the hash itself, but returns only a part of the hash values that match with the prefix of the user.

Note that we include stretching in our cost analysis, but leave the numerical investigation of manipulating anonymity sets to future work.

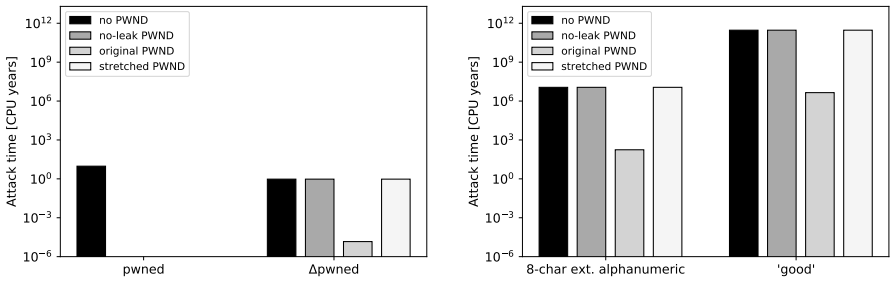
4 Cost-benefit analysis

4.1 Attack cost estimation

Here we estimate the cost of attacks defined in Section 3.2 in case of different versions of the on-premise PWND mechanism in terms of effort and monetary cost needed by the attacker to crack a single password. The basis of our simple calculations are the following: the size of the pwned password database, categories of password strengths and the number of potential password candidates in each category, hash computation times and the cost of CPU time.

Parameters. At the time of writing this article, the popular site haveibeenpwned.com claims to know about more than 8 billion pwned accounts. Accounting for the significant overlap among actual passwords, we estimate the number of pwned passwords at 600 million. We assume that PWND mechanisms use a 16-bit prefix, i.e., this characterizes the information leakage. We also assume that a PWND database update brings with itself 10% new password hashes, i.e., 60 million new hashes in our case. In our scenario, it is sensible to take into account four different password strength categories: pwned passwords, passwords that were not pwned at the time of registration but have become pwned since then (we refer to those as Δpwned), "medium" strength passwords (defined as 8-character-long alphanumeric passwords with basic special characters) and "good" passwords (conservatively defined as having 64 bits of entropy, corresponding to 10-11 character long extended alphanumeric passwords). Note that weak passwords are excluded in line with our assumption on the service provider's policy. As the computation time of an SHA-1 hash is in the order of a few milliseconds we take it as zero, while we parametrize our stretched hashing in a way that it takes exactly 1 second to perform (inclusive of the decryption attempt of the user's data, see Section 3.1). To estimate the cost of CPU time, we turn to the price list of publicly available cloud computation instances, where a 16-core processor can be rented for \$4 per hour, which equals to \$2,190 for a single core per year. Note that here we omit attack preparation costs, such as acquiring/compiling a dictionary (anyway, the public PWND database is readily available to the attacker).

We consider 4 different PWND versions: i) a baseline with no PWND implemented, ii) an ideal, no-leak PWND (not practical owing to the large amount of data transfer that interferes with user experience), iii) the original PWND detailed in Fig. 2, and iv) a version with stretched hashing. Fig. 3 shows the expected required resources for a successful attack on a single user in CPU years. Note that both plots use the same log y-axis for the sake of comparison. Also



(a) Dictionary attack (b) Brute force attack

Fig. 3. Expected attack times in CPU years

note that attack times and costs for the two different brute force attacks are the same, albeit the two attacks are different (targeted vs. non-targeted) and also accelerated differently (see Section 3.2).

Fig. 3(a) shows that not implementing any PWND mechanism allows a dictionary attack using the public PWND database as a dictionary of hashes to be completed in less than 10 CPU years ($21,900) on average; this works only for pwned passwords, of course. Any PWND mechanism would mitigate this special case. On the other hand, for passwords that have become pwned after they were used for registration (Δpwned), quick attacks can be mounted. This follows from the fact that the adversary can construct the Δ between the two versions of the database, and use this shorter list as its attack dictionary. Specifically, if the provider implements the regular PWND mechanism detailed in Fig. 2, the attack requires roughly 8 minutes to succeed on a single CPU core, for this special case. Stretched hashing increases the attack cost, but does not make the attack impractical (≈ 1 CPU year or $2,190).

Clearly, implementing a PWND mechanism is advisable, especially for encrypted cloud storage, where the sensitive content is likely to be stored. Furthermore, additional safeguards for database updates are needed; note that the adversary learns about the updates of public databases at the same time as the provider. Designing such safeguards constitute important future work.

Regarding brute force attacks, a "good" password can be considered safe against a brute force attack as it would take $\approx 4.5 \cdot 10^6$ CPU years even with the original, top-n bit leaking PWND mechanism to crack such a password. Clearly, the most interesting case concerns "medium-strength", 8-character-long extended alphanumeric passwords. In this case, when using the state-of-the-art PWND mechanism, it would take ≈ 175 CPU years on average to crack one password; this amounts to $\approx$ $382,500. Such monetary cost can be considered borderline practical; more so with the declining cost of computation resources. We observe that stretching the hash mitigates the information leakage issue successfully in this case, increasing the cost of a successful attack to $1.15 \cdot 10^7$ CPU years and more than $25 billion. We can conclude that utilizing an on-

premise PWND mechanism with stretching allows for a milder password strength policy that users may like.

4.2 User behavior

User behavior when facing forced password change and stricter security policies is shown to be mixed [16]. Users of a service provider can be partitioned into multiple categories with respect to their password habits and willingness to comply with an additional security mechanism such as compulsory password change. As such, some users get increased security from a deployed PWND mechanism, while some could actually suffer a decrease in security (users with "medium-strength" passwords, if the provider utilizes the regular PWND mechanism). In addition, even with benefiting from better security, users forced to change passwords may just leave the service altogether because of the bother[13].

Obviously, a service provider can suffer the costs of users leaving (either because of bother or decreased security), but may pay a larger reputation cost if an attacker can access a user account with a pwned password. In fact, providers selling a security-related service are probably better off deploying an enhanced PWND mechanism. Ultimately, the actual benefit of a proper PWND mechanism depends on the composition of the service provider's user base with regard to password habits.

5 Discussion

The cost of checking whether a password is known to be pwned can also be computational and/or communication overhead besides the investigated security loss, caused by the k-anonymity approach. However, the efficiency of cryptographic protocols, called Private Set Intersection or more specifically Private Set Membership (PSM), that could eliminate the information leakage entirely, seem to be prohibitive in case of the considered application.

The goal of PSM protocols is to enable two parties (typically a server and a user) to securely decide whether a value, determined by one of them (user), is an element of a set belonging to the other (server). The security guarantee of PSM informally says that from their interaction, neither the user nor the server should learn anything about the other's input beyond the result[14] of the membership test. Authors of [3] studied PSM first, and showed its connection to Oblivious Transfer (OT), a fundamental cryptographic protocol.

When trying to apply PSM protocols for password verification, the main source of inefficiency is that both communication and computational costs depend on the set size, the order of magnitude of which is upwards of 10^8. According to the measurements of [14, Table 5-6] on desktop PCs, for passwords of 8

[13] This seems to be an existing threat: the online payment website of a major Hungarian mobile provider offers the alternative to the user to keep their old password even after prompting them to change it due to expiration!

[14] Different variants exist based on who receives the output: only the user [14], only the server [9] or both of them in a secret shared form [2].

characters it would take 13.8 sec and a communication cost of 78.3 MB to check membership in a set of only 2^{18} elements; far less than the PWND database. We consider such a delay already impractical [4].

6 Conclusion

In this paper, we investigated the effect of information leakage when using state-of-the-art PWND mechanisms implemented at the service provider itself. We also presented simple techniques based on hash stretching and anonymity set design that could negate the acceleration of password cracking attacks owing to the usage of PWND. Our attack cost calculation showed that i) public PWND databases can be used as dictionaries for password cracking attacks ii) stretching-based mitigation is effective concerning the potentially vulnerable users using "medium strength" passwords. We also discussed how cryptographic solutions leaking no information are not yet practical in the PWND context. We have barely scratched the surface: as future work, we plan to analyze PWND mechanisms based on full account information, improve existing schemes by anonymity set engineering and handling PWND database updates, conduct a survey on PWND usage among users and service providers, and devise a formal, in-depth cost-benefit analysis.

References

1. J. Ali. Validating leaked passwords with k-anonymity, Feb. 2018. https://blog.cloudflare.com/validating-leaked-passwords-with-k-anonymity/, Accessed: 15-06-2020.
2. M. Ciampi and C. Orlandi. Combining private set-intersection with secure two-party computation. In *SCN*, volume 11035 of *Lecture Notes in Computer Science*, pages 464–482. Springer, 2018.
3. M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 303–324. Springer, 2005.
4. D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 5(1):1, 2004.
5. H. Habib, J. Colnago, W. Melicher, B. Ur, S. Segreti, L. Bauer, N. Christin, and L. Cranor. Password creation in the presence of blacklists. In *Proceedings of USEC'17*, page 50, 2017.
6. Have I Been Pwned. Website. https://haveibeenpwned.com. Accessed: 15-06-2020.
7. T. Hunt. Have I Been Pwned is Now Partnering With 1Password. https://www.troyhunt.com/have-i-been-pwned-is-now-partnering-with-1password/, 2018. Accessed: 15-06-2020.
8. J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. In *International Workshop on Information Security*, pages 121–134. Springer, 1997.
9. V. Kolesnikov, M. Rosulek, N. Trieu, and X. Wang. Scalable private set union from symmetric-key techniques, 2019.

10. L. Li, B. Pal, J. Ali, N. Sullivan, R. Chatterjee, and T. Ristenpart. Protocols for checking compromised credentials. In *Proceedings of ACM CCS*, 2019.
11. N. Matatall. New improvements and best practices for account security and recoverability. https://bit.ly/3ftvCcA, 2018. Accessed: 15-06-2020.
12. P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.
13. T. Petsas, G. Tsirantonakis, E. Athanasopoulos, and S. Ioannidis. Two-factor authentication: is the world ready?: quantifying 2fa adoption. In *Proceedings of the eighth European workshop on system security*, page 4. ACM, 2015.
14. B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. In *USENIX Security Symposium*, pages 797–812. USENIX Association, 2014.
15. P. Samarati and L. Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report, technical report, SRI International, 1998.
16. R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor. Encountering stronger password requirements: user attitudes and behaviors. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, page 2. ACM, 2010.
17. J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.
18. SpyCloud. Website. https://spycloud.com/. Accessed: 15-06-2020.
19. E. Stobert and R. Biddle. The password life cycle: user behaviour in managing passwords. In *10th Symposium On Usable Privacy and Security (SOUPS-2014)*, pages 243–255, 2014.
20. K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, P. G. Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh, and E. Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *Proceedings of the USENIX Security Symposium*, 2019.
21. Tresorit. White Paper. https://tresorit.com/files/tresoritwhitepaper.pdf. Accessed: 15-06-2020.