Dono: A Stateless Password Manager

Panos Sakkos
panos.sakkos@protonmail.com

ABSTRACT

- Draft - Passwords are the cornerstone of privacy and security but they are cumbersome to handle. Password managers offer to take away the pain of maintaining one regularly changing and strong password per service by creating an account for their user and using it as the master key of accessing all the passwords. For purposes of usability, they offer syncing of the data between the different devices that the user uses and for that purpose, they have to store all the passwords in their backend service. We propose a purely computational password manager, which requires no data transfer between devices and doesn't store any password. There is no need for a service that syncs data across different devices and all the implementation offered is in the form of native apps. By open-sourcing `Dono`, the users are able to trust that all their passwords live at their devices only, while computing the same passwords across all the different devices used.

1. INTRODUCTION

Ideally a person should handle its passwords by remembering a strong and unique password per service used and keep updating it regularly with a new unique and strong password. In practice this is challenging and something that an average person cannot achieve without investing a lot of time on it. Password managers provide solutions that offer handling of all the passwords by maintaining a username and a master key of the user.

Local password managers as i.e. Password Safe [2], offer only client apps and usually are open-sourced, therefore they can be trusted by the users. On the other hand, they hurt usability, since there is an inherent need to import the encrypted passwords after each new installation of the offered clients. As a result, there are users who are using storage cloud services, like Dropbox i.e., in order to sync their encrypted password database across their devices. Therefore, the passwords are can be brute-forced by the companies that offer the storage services, or handed over to a government, in order to be decrypted.

The usability gap of the local password managers is addressed by the online password managers, like i.e. LastPass, which sync the passwords to their backend services in order to have them ready to be shipped to any new device that the user logs in with the same account. This approach cannot be trusted since the user is not in a position to know the source code that is handling the passwords, even if these services are open-source. Apart from the lack of trust from the online password managers, they expand significantly the attack surface, resulting to multiple vulnerabilities [7].

`Dono` provides the benefits from both the above approaches by computing, instead of storing, the passwords. From a user's perspective, the user needs to provide a self-defined Key and a fixed description of the service that needs to retrieve the password for, the *Label*. For example, the *Labels* for a LinkedIn and a GitHub account could be "$LinkedIn$" and "$GitHub$" respectively. In the case where multiple passwords are needed for a service, then the *Labels* can be namespaced. For example the tags for ProtonMail's Login and Mailbox passwords could be "$ProtonMail.Login$" and "$ProtonMail.Mailbox$". The purpose of the *Label* is to generate a unique password per service but at the same time to be easy for the user to remember it. More specifically, the user does not need to remember it, because this information, based on Section 4, can be publicly available. The Key and the *Labels* are handled in a way that a new Key will generate a whole new set of passwords for each *Label*. As a result, when the user wants to update all the passwords, then the only information that needs to be updated is the Key.

Given that the computation of the passwords is deterministic, there is no need to sync any data between the user's devices, which means that the computed passwords don't have to leave the device and reach any backend service. `Dono` is offered only as open-source native apps. As a result, the users can trust the `Dono` apps, by verifying the signatures of the respective produced binary. A client-only web-app version of `Dono` is possible, but it's deliberately avoided in order to mitigate online phishing attacks [3].

In the case where a computed password for a service is compromised in plain text by an attacker, then the attacker will compromise only this service, given that the user is using different *Labels* per service and, based on Section 4, the attacker will not be in a position to compute the Key in a reasonable time.

Summarizing, `Dono` provides the following:

1. Establishing trust with the users about through transparency
2. Reducing significantly the attack surface of password managers, by replacing their storage needs with computations

2. RELATED WORK

`PwdHash` [11] was a first attempt to utilize cryptographic hash functions in order to offer an extra layer of security for passwords. It was implemented only as a Firefox plugin and was focused on mitigating a category of phishing attacks which could execute javascript code in order to steal plain text passwords from the plugin. `PwdHash` was generating one password per website but it was expecting existing passwords as an input, which led to confusion among users [3]. Also, the salt was not secret, and it computed based on the website domain, which also proved to be challenging during the implementation. The password was not stretched with iterations, but it was suggested as a future step. Last, web plugins are not anymore a viable solution, given the rise of the small screen portable devices which usually don't allow third party plugins to extend the browser capabilities.

`Password Multiplier` [4] was a Firefox plugin, which was computing unique passwords per website name, given a mas-

ter key and the user's username. The salt used was the username and there were two rounds of hash iterations. The first round of iterations was computed after the first run of the plugin and the second when requesting the password for a website. The first round of iterations was cached, and in the case of its compromise by the attacker, then the computing cost per password for the attacker was reduced dramatically. Apart from the non-ideal salt used, the assumed attacker that was brute-forcing the algorithm was a "modern PC" which would need around 100 seconds in order to guess a password. Since then, the trend of cryptocurrencies has driven the cost of ASIC hash computers to very low costs and as a result anyone with a budget of a few thousand dollars can buy a computational power in the order of GH/s. Also, the computations were not taking into account

`Passpet` [12] was the continuation of `Password Multiplier` and was focusing on extending `Password Multiplier` for better usability. It remained a Firefox plugin and petnames were introduced, which were labels picked by the user and assigned to websites. `Passpet` introduced a remote server, which was storing the information of the number of the first round of hashing iterations and the user's labels.

3. PASSWORD COMPUTATION

The *Label*, l, is converted to lower case and leading and trailing spaces are trimmed, in order to not force the user remembering the case-sesitivity of the used *Labels* and to avoid accidental spaces added by phone and tablet keyboards.

A unique per combination of (Key,*Label*) salt, s, is computed by hashing user's Key, k, appended by the *Label*, l, and a fixed *magicsalt* in order to mitigate encrypted dictionaries in advance [8]. In our case the *magicsalt* is equal to "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127 b7afdeda33b".

k, l and s are appended and hashed in order to derive the password, d, for l.

In order to slow down the attacker [5], d is hashed with a salt multiple times before deriving the final d. The number of iterations is determined upon the size of k, in order to protect users from picking a short k. For example, a k with length 15, will need so many iterations that it will not derive the final d in a reasonable time, on a computing device.

Algorithm 1 Password computation

procedure ComputePassword(k, l) ▷ The Key and the *Label*
  $l \leftarrow l.LowerCase().TrimStart(\text{“”}).TrimEnd(\text{“”})$
  $s \leftarrow$ "$4a5e1e4b...76673e2cc77ab2127b7afdeda33b$"
  $s \leftarrow SHA256(k + l + s)$
  $d \leftarrow SHA256(k + l + s)$
  $rs \leftarrow DecideRounds(k)$
  for $i = 0, i < rs, i \leftarrow i + 1$ **do**
    $s \leftarrow SHA256(d + s)$
    $d \leftarrow SHA256(d + s)$
  end for
  return d
end procedure

In order to define the number of derivation rounds, rs, we need to define an attacker and the computational time that we want to enforce for brute-forcing all the possible values of k. The function *DecideRounds* is described in Section 4.

Approaches like bcrypt [10] for Algorithm 1 are not suitable because of the client-side purpose of use.

4. ROUNDS OF DERIVATION

4.1 Attacker

In order to define the computation of rs for function *DecideRounds*, we need to define the attacker that we are protecting against and the computational time that we want for the brute-forcing of every value of k, given a fixed k length, l.

Following Kerckhoffs's principle [6] we assume that the attacker has knowledge of everything except the Key:

1. Knows that the victim used Algorithm 1 for computing passwords
2. Knows all the victim's computed passwords
3. Knows all the victim's *Labels* that were used for computing the passwords
4. Knows the character set that was used for the victim's Key and its length, l

We have assumed that the length of the victim's key is known in order to simplify the calculations.

Moreover, we assume that the attacker is capable of the following computing capabilities:

1. Has infinite storage
2. Has compromised all the nodes of the Bitcoin [9] network and uses them to compute SHA256 hashes

We have assumed that the attacker has compromised the Bitcoin network, because it's the most powerful hash computer that is known.

4.2 Victim

We assume the following for the victim of the attacker described in 4.1

1. Victim's Key consists only of Latin letters, all in the same case-sensitivity (i.e. only non-capital letters)
2. Victim's *Labels* remain the same forever
3. The victim's Key length, l, remains the same forever

4.3 The attack

4.3.1 Goal

The goal of the attacker is to find the Key of the victim.

4.3.2 Computational power

The current global maximum of Bitcoin network's Hash Rate (as of 28th of March 2016) is 1,375,500,956 GH/s[1], or $13.75500956 \times 10^{17}$ hash computations per second and this is the computing power the attacker has available. The cost, in terms of hashes, of computing one password computed by Algorithm 1 is $2 + 2 \times rs$ hashes. The *Password Rate*, pr, computed passwords per second, of the attacker for Algorithm 1 is:

$$pr = \frac{13.75500956 \times 10^{17}}{2 + 2 \times rs}$$

4.3.3 Computational time

The attack is performed by brute-forcing k in Algorithm 1, since all l and final d values are known.

For a given k length, l, the possible $Keys$ of the victim are 26^l. The time, in (365 days long) years, that the attacker needs to compute all the passwords for a length l, T_l, is:

$$T_l = \frac{26^l}{31536000 \times pr}$$

By upper bounding T_l to 1000 years, for each l we can compute $rs(l)$:

$$rs(l) = \lceil \frac{500 \times 13.75500956 \times 10^{17} \times 31536000 - 26^l}{26^l} \rceil$$

By encapsulating the calculation of $rs(l)$ in the *DecideRounds* function, we complete Algorithm 1 with Algorithm 2.

Algorithm 2 Determine rounds of derivation

```
1: procedure DecideRounds(k)                    ▷ The Key
2:     l ← k.length()
3:     t ← 1000                                 ▷ Target brute force years
4:     c ← 13.75500956 × 10^17                  ▷ Computational power
5:     rs ← ⌈(t/2 × c × 31536000 − 26^l) / 26^l⌉
6:     return rs
7: end procedure
```

A periodic update of the target computational power, c, of the *PasswordRate* is encouraged, in order to keep up with the hardware improvements. This update will need to provide a migration path to the users, because the change in the computation will lead to different results, for each pair of (k, l).

Table 4.3.3 lists the different rounds of derivation per Key length. $Keys$ starting from a length of 17 can be computed in a responsive time even on mobile clients.

Table 1: Rounds of Derivation for different Key lengths

l	rs
0	216888990742079999999999999
1	83418842593107692307692307 5
2	32084170228118343195266271
3	12340065472353208921256 24
4	47461790278281572774061
5	1825453472241598952847
6	70209748932369190493
7	2700374958937276556
8	103860575343741405
9	3994637513220822
10	153639904354646
11	5909227090562
12	227277965020
13	8741460192
14	336210006
15	12931153
16	497351
17	19127
18	734
19	27
20	0

The attacker, given the infinite storage capabilities, can store the computed values of $ComputePassword(k, l)$ for each k that is brute-forced, in order to compromise service with *Label* l, for every k in the future. This is mitigated by changing the *magicsalt* of Algorithm 1 periodically, i.e. while updating the computational power, c, of the $PasswordRate$.

4.3.4 Cost of the attack

We assume that the attacker is not charged for electricity costs and there are no hardware failures while the attack is taking place. The arguably best ASIC hash computer with respect to GH/s per $ is the `Spondoolies Tech SP20 Jackson`, which costs $500 and its advertised capacity is 1,500 GH/s, resulting to $0.33 per GH/s. For the computational power of the assumed attacker, this would cost $45,850,031. For a high value target, the attacker could invest more money in the attack, in order to find the victim's Key in a reasonable time, i.e. within a year. In this case that would cost $45,850,031,000.

An advanced user that believes that is a high value target, can download the source code, update the target brute force years and the computational power and then compile and deploy privately the new hardened `Dono` binaries. Modifications like this will require a longer Key in order for the password computation to finish in a responsive time.

5. IMPLEMENTATION

6. CONCLUSIONS

7. REFERENCES

[1] Hash rate of bitcoin network. `https://blockchain.info/charts/hash-rate`, Dec. 2015.

[2] Password safe. `https://www.schneier.com/passsafe.html`, Oct. 2015.

[3] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *Usenix Security*, volume 6, 2006.

[4] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. In *Proceedings of the 14th international conference on World Wide Web*, pages 471–479. ACM, 2005.

[5] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. In *Information Security*, pages 121–134. Springer, 1998.

[6] A. Kerckhoffs. *La cryptographie militaire*. University Microfilms, 1978.

[7] Z. Li, W. He, D. Akhawe, and D. Song. The emperor's new password manager: Security analysis of web-based password managers. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, 2014.

[8] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

[9] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.

[10] N. Provos and D. Mazieres. Bcrypt algorithm. USENIX, 1999.

[11] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *Usenix security*, pages 17–32. Baltimore, MD, USA, 2005.

[12] K.-P. Yee and K. Sitaker. Passpet: convenient password management and phishing protection. In *Proceedings of the second symposium on Usable privacy and security*, pages 32–43. ACM, 2006.