Dono: A Password Derivation Tool

Panos Sakkos
panos.sakkos@gmail.com

Stanko Krtalic Rusendic
stanko.krtalic@gmail.com

Vincent Olislagers
vincento@posteo.net

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 passwords should be handled by remembering a strong and unique password per service used and keep updating them 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 [1], 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 encrypted passwords can be brute-forced to be decrypted, by an attacker of the storage service, the storage service itself, or a government that requested access to this encrypted data.

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 [9].

`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 $Label$.

Given that the computation of the passwords is deterministic, there is no need to sync any data between the user 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 [2] native apps. A client-only web-app version of `Dono` is possible, but it's deliberately avoided in order to mitigate online phishing attacks [4].

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. Making passwords available at all the user's devices, without the information leaving out of them

2. Reducing significantly the attack surface of password managers, by replacing their storage needs with computations

2. RELATED WORK

`PwdHash` [12] 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 [4]. 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` [5] was a Firefox plugin, which was computing unique passwords per website name, given a master 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` [13] 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. CORE KEY DERIVATION FUNCTION

The Core Key Derivation Function of `Dono`, aims to slow down the attacker [7] and to mitigate encrypted dictionary attacks [10] by performing key stretching using PBKDF2 [6] on the user's Key k, in order to derive the password d for the Label l. The pseudorandom function used in PBKDF2 is SHA-256 and the salt s is computed by hashing the Key k and the Label l along with a fixed salt, *magic salt*. The number of derivation iterations c is left as an argument to the caller.

The unique per combination of (Key,$Label$) salt, s, is computed by using SHA-256 to hash the user's Key, k, concatenated with the $Label$, l, and the fixed *magic salt*. The *magic salt* is equal to "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127 b7afdeda33b".

Algorithm 1 Core Key Derivation Function

```
1: procedure DerivePassword(k, l, c)   ▷ The Key, the Label and the derivation iterations
2:     s ← "4a5e1e4b...76673e2cc77ab2127b7afdeda33b"
3:     s ← SHA256(k + l + s)
4:     d ← PBKDF2(SHA256, k, s, c, 256)
5:     return d
6: end procedure
```

4. ITERATIONS OF STRETCHING

In order to give a meaning to the number of iterations c that Algorithm 1 expects to be called with, we will define an imaginary attacker that we are defending the user from. The number of iterations c will be defined by forcing the assumed attacker to a specific time frame of brute-force computing.

4.1 Attacker

Following Kerckhoffs's principle [8] we assume that the attacker has knowledge of the password derivation algorithm, but lacks the knowledge of the Key k. Moreover we assume the knowledge of the following:

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

The assumption that the length of the victim's key is known, is made in order to simplify the calculations. With respect to the attacker's computing capabilities, we assume that the attacker:

1. Has infinite storage
2. Has compromised all the nodes of the Bitcoin [11] network and uses them to compute SHA-256 hashes

The assumption that the attacker has compromised the Bitcoin network is made in order to help the reader quantify the attacker's computing capabilities.

4.2 Victim

We assume the following for the victim of the attacker described in 4.1, in order to simplify the calculations.

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 global maximum of Bitcoin network's Hash Rate (as of 1st of August 2016) is 1,796,101,466 GH/s[3], or $17.96101466 \times 10^{17}$ SHA-256 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 $1 + c$ hashes, as long as the target length of the derived password is not longer than 256 bits. The computed passwords per second, pr, of the attacker for Algorithm 1 is:

$$pr = \frac{17.96101466 \times 10^{17}}{1 + c}$$

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 $c(l)$:

$$c(l) = \lceil \frac{1000 \times 17.96101466 \times 10^{17} \times 31536000 - 26^l}{26^l} \rceil$$

The calculation of $c(l)$ is encapsulated in the *GetIterations* function of Algorithm 2.

Algorithm 2 Determine rounds of derivation

1: **procedure** GETITERATIONS(k, t, p) ▷ The Key, the target brute force years, the attacker's computational power
2: $l \leftarrow k.length()$
3: $c \leftarrow \lceil \frac{t \times p \times 31536000 - 26^l}{26^l} \rceil$
4: **return** c
5: **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 iterations of stretching per Key length. $Keys$ starting from a length of 17 can be computed in a timely manner even on mobile devices.

Table 1: Iterations of derivation for different Key lengths

l	rs
0	566418558317759999999999999999
1	217853291660676923076923076 8
2	83789727561798816568047336
3	3222681829299954483386435
4	123949301126921326284092
5	4767280812573897164771
6	183356954329765275567
7	7052190551144818290
8	271238098120954548
9	10432234543113635
10	401239790119754
11	15432299619989
12	593549985383
13	22828845590
14	878032521
15	33770480
16	1298863
17	49955
18	1920
19	72
20	1

The attacker, given the infinite storage capabilities, can store the derived passwords d for each Key k that is brute-forced, in order to compromise the service with *Label* l, for every Key k in the future. This is mitigated by changing the *magic salt* of Algorithm 1 periodically, i.e. while updating the computational power, c, of the *Password Rate*.

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 $592,713,483. 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 $592,713,483,000.

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 timely manner.

5. IMPLEMENTATION

`Dono` accepts the user's Key k, the target Label l. If the given Key k is less than 17 characters, then the computation is rejected, because the mobile devices are not in a position to perform the required computation in a responsive time. The *Label*, l, is transformed to lower case and the leading and trailing spaces are removed, in order to not force the user to remember the case-sensitivity of the used *Labels* and to avoid accidental spaces added by phone and tablet software keyboards.

Algorithm 3 `Dono` Password Derivation

1: **procedure** COMPUTEPASSWORD(k, l) ▷ The Key and the Label
2: **if** $k.length < 17$ **then**
3: **return** *error*
4: **end if**
5: $l \leftarrow l.LowerCase().Trim()$
6: $c \leftarrow GetIterations(k, 1000, 17.96101466 \times 10^{17})$
7: $d \leftarrow DerivePassword(k, l, c)$
8: **return** $d.ToHex().Substring(0, 64)$
9: **end procedure**

`Dono` has been implemented for iOS and macOS and is offered in the respective App Store as a sandboxed app. The Android and Windows 10 apps are under development, as well as a Command Line Interface and an Application for Linux.

By default the apps are not storing the Key k, but the user can configure to persist the Key k between the app launches. If the user opts-in for remembering the Key between the app launches, then the Key is stored in the Operating System's secure storage, i.e. Apple's Keychain Access. It is recommended that online backup of the secure storage is disabled, in order to persist the Key k only on the devices running `Dono`.

The user can opt-in for Passcode Lock and biometric authentication in the mobile apps, if any available. The Passcode Pin is stored in the Operating System's secure storage as well. After three wrong Passcode unlock attempts, then the Key (if any) and the Passcode are deleted from the secure storage. If the Key k of the user is lost, then all the derived passwords will be lost as well, but this is considered a feature.

There are services (like i.e. Apple ID) that will not accept `Dono`'s derived passwords. That will happen due to the lack of a symbol or a capital character or even due to the password's length. `Dono` will not compromise its simplicity or security in order to favor services that actually require weaker

passwords. Instead, these services should update their password policies in order to accept stronger passwords, like the ones that `Dono` generates.

`Dono` is open-sourced under the GPLv3 license. Users who don't trust that the open-sourced code is what is published in the respective App Store, they can compile and deploy the apps themselves.

6. CONCLUSIONS

We presented `Dono`, a password derivation tool which derives passwords from a master *Key* by using short descriptions of the destination service, the *Labels*. The derivation of the final password is done in a way that makes it practically impossible to find the user's *Key*, even though if the attacker has access to user's *Labels* and their derived passwords. By using `Dono` for handling passwords, users are in a position to know that their passwords are available to all their devices and only for them, at the same time.

7. REFERENCES

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

[2] Dono source code. `https://github.com/dono-app/`, Aug. 2016.

[3] Hash rate of bitcoin network. `https://blockchain.info/charts/hash-rate`, Aug. 2016.

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

[5] 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.

[6] B. Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. 2000.

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

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

[9] 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.

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

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

[12] 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.

[13] 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.