The Ginger: Another Spice to Hinder Attacks on Password Files

Francesco Buccafurri[a], Vincenzo De Angelis[b] and Sara Lazzaro[c]
Department of Information Engineering, Infrastructure and Sustainable Energy (DIIES), Università Mediterranea di Reggio Calabria, Via dell'Università 25, 89122 Reggio Calabria, Italy

Keywords: Passwords, Authentication, Salt, Dictionary Attacks.

Abstract: One of the threats to password-based authentication is that the attacker is able to steal the password file from the server. Despite the fact that, thanks to the currently adopted security mechanisms such as salt, pepper, and key derivation functions, it is very hard for the attacker to reverse the password file, dedicated hardware is available that can make this attack feasible. Therefore, there is still a need to better counter password-file reversing. In this paper, we propose a new mechanism, called *ginger*, which can be combined with the above mechanisms, to increase the robustness of password-based authentication against password-file reversing. Unlike pepper and salt, ginger is stored client-side, and enables a stateful authentication process. A careful security analysis shows the benefits of the proposed innovation.

1 INTRODUCTION

Password-based authentication is the most widely used authentication method for web applications. However, over the years, several studies in the literature have shown that it suffers from many security problems (Bonneau et al., 2012). The main issue lies in how users choose their passwords. Ideally, passwords should be long random strings so that they cannot be easily guessed. However, since humans have poor memory capacity, they tend to choose short and mnemonic passwords (NordPass, 2022). Among others, an important factor threatening password-based authentication is represented by the compromises of password files stored server-side. As a matter of fact, over the past decades, several compromises have occurred thus leading to the leakage of billions of login credentials (Shadow, 2022). Unfortunately, these attacks do not concern compromised websites alone but might increase the risk of account takeover at other websites. This is due to password reuse, induced by the proliferation of accounts per user (25 accounts on average (Das et al., 2014)). Obviously, password files cannot trivially store passwords in the clear, since a simple server-side compromise would immediately reveal all the users' passwords. Therefore, best-known security practices suggest storing the hashes of each password combined with a random string called *salt*. This way, the precomputation of possible password files (to build the so-called *rainbow tables* (Kumar et al., 2013)) is not possible, and the only possibility for the attacker is to perform a dictionary-based attack once both password file and salt have been stolen from the server. To further hinder dictionary attacks additional security mechanisms, such as *pepper* (Manber, 1996) and *key derivation functions* (Yao and Yin, 2005), can be adopted with the aim of increasing the time needed for a dictionary attack. Nevertheless, dictionary attacks are still feasible, since attackers can leverage dedicated hardware platforms, ranging from traditional application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs) to modern graphic process units (GPUs) and hybrid CPU-FPGA chips (Zhang et al., 2020). Moreover, dictionary attacks have also become more sophisticated across the years. Attackers typically build their dictionaries aggregating passwords leaked from past data breaches and plain-word dictionaries. Then such dictionaries can be virtually extended through mangling rules (Pasquini et al., 2021) that consist of a series of transformations that can be applied to a basic dictionary to produce multiple candidate passwords from each word of the dictionary. Mangling rules are included in many popular password-cracking software such as Hashcat (Jens 'atom' Steube, Gabriele 'matrix' Gristina, 2015) and John the Ripper (OpenWall,

[a] https://orcid.org/0000-0003-0448-8464
[b] https://orcid.org/0000-0001-9731-3641
[c] https://orcid.org/0000-0002-0846-4980

Buccafurri, F., De Angelis, V. and Lazzaro, S.
The Ginger: Another Spice to Hinder Attacks on Password Files.
DOI: 10.5220/0011576200003318
In *Proceedings of the 18th International Conference on Web Information Systems and Technologies (WEBIST 2022)*, pages 166-173
ISBN: 978-989-758-613-2; ISSN: 2184-3252

1996). The problem is therefore still open.

In this paper, we propose a new mechanism, called *ginger*, which can be combined with existing techniques, to make dictionary attacks even more impractical. The core of our proposal consists in moving from a *stateless* authentication mechanism to a *stateful* process. In fact, we require the client to maintain the state of each authentication. As we will show in the rest of the paper, there are no technical issues limiting the implementation of our approach. Moreover, our approach does not require extensive server-side changes. On the contrary, it can be used in conjunction with the commonly used approaches for server-side password storage (salt, pepper, and key derivation functions), hence strengthening the protection they offer against dictionary attacks.

Commonly, to minimize the impact of such attacks, it is suggested that users should adopt password managers or enable multi-factor authentication (Shadow, 2022). Hence, we also perform a security comparison between our solution and password manager and OTP-based authentication. As for password managers, their main vulnerability is that they are a single-point-of-failure. Indeed, they base their security on a mnemonic password (the so-called master password) which, in principle, might be weak. Hence the security of the random-generated passwords depends on the security of the master password. On the other hand, concerning OTP-based mechanisms, they act as a second factor for the authentication but do not offer any protection of the password. All these aspects are carefully analyzed in this paper. In particular, our analysis shows that our solution outperforms both the two above-mentioned approaches in terms of protection offered against a dictionary attack.

This paper is organized as follows. In Section 3, we provide some background notions about the best security practice to protect password files stored server-side. Then, in Section 4, we describe in detail our proposal. Practical aspects concerning our proposal are discussed in Section 5. In Section 6 we evaluate the security of our approach. In Section 2, we investigate the related literature. Finally, in Section 7, we draw our conclusions.

2 RELATED WORK

In the last 30 years, several web authentication schemes have been proposed in the literature. For a comprehensive overview, the reader can refer to (Bonneau et al., 2012). In the paper, the authors also provide a comparative framework for web authentication scheme taking into account usability, deployability, and security aspects. Even with alternative proposals, password-based authentication represents the predominant approach in real life (Mohammed et al., 2017; Andrews et al., 2020). As a matter of fact, alternative proposals (e.g., biometrics and OTP) are used as a second factor to passwords. An attempt to replace textual passwords is represented by graphical passwords (Chiasson et al., 2011; Tao, 2006), in which the users have to remember images in place of text. However, they lack in usability (Bonneau et al., 2012) and are not largely adopted.

In the following, we report the main approach to strengthen password-based authentication.

We start from password managers (Li et al., 2014), whose description is also provided in Section 6.2. They store users' passwords in encrypted form, that are decrypted locally through a single master password. The problem with password managers is that they are vulnerable to data leaks (Last Pass, 2022) leading to dictionary attacks on the master password. Recent approaches to solving this problem are presented in (Shirvanian et al., 2021; Shirvanian et al., 2017), in which the password manager does not store the passwords of the user (even in encrypted form), but they are computed using an oblivious pseudo-random function password hardening approach that receives the master password from the user and a secret key from the password manager server.

Another approach allowing the user to authenticate with several web services by remembering just a password is *federated authentication* (Boehm et al., 2008; Lenz and Zwattendorfer, 2015). Therein, a trusted identity provider attests the identity of the user and allows them to authenticate with a web service. From the point of view of the security of the passwords, federated authentication suffers from the same drawback as password manager since the effort of the attacker moves from compromising the web service in which the user authenticates to compromising the identity provider.

In many real-life applications, password-based authentication is often strengthened by introducing an OTP as a second factor (Buccafurri et al., 2020; Eldefrawy et al., 2011). However, as highlighted in Section 6.3, this second factor is not included in the hash stored server-side, so it protects the authentication of the user but does not prevent password discovery.

We conclude this section, by discussing FIDO2 (Chakraborty and Bugiel, 2019), a new password-less authentication framework. It is based on a challenge-response mechanism and public-key cryptography. However, to achieve strong security features, dedicated hardware (e.g., the Yubikey) is needed.

3 EVOLUTION OF PASSWORD-BASED AUTHENTICATION SCHEMES

The basic idea underlying a password-based scheme is the following. Suppose a user U is registered at the server S with a password pw_U. S locally stores a function of the password pw_U, say $F(pw_U)$. When U wants to authenticate at the server S, they send S the string pw'_U (hopefully, coinciding with pw_U). Then, S simply checks if $F(pw'_U) = F(pw_U)$ to decide whether or not the user is who claims to be.

A naive way for S to handle passwords is to memorize, for each user U, $F(pw_U) = pw_U$. Clearly, if the attacker is able to steal the password file, then all the passwords stored at S are exposed in the clear.

This is clearly unacceptable. A step ahead to improve the protocol, is to implement F as a cryptographic hash H. Indeed, H being a one-way function, the password file cannot be trivially reversed.

An attacker can still mount an offline brute force attack to find all possible passwords matching the hashes stored by S. Such an attack tries all possible passwords in a systematic way, thus it might require a significant amount of time. However, taking into account how users choose passwords, a more efficient attack can be performed. Indeed users tend to build their passwords by using common words, possibly combining them in a predictable way. The list resulting from this combination, including all the commonly adopted passwords, is named *dictionary*. Many tools allowing password cracking based on dictionaries are available (John the Ripper is the most famous (OpenWall, 1996)) and many dictionaries can be found on the Internet. Consider that, the above attack, can be also performed by pre-computing all possible hashes (by obtaining the *rainbow tables* (Kumar et al., 2013)), so that passwords can be later found once the password file is stolen.

To prevent rainbow-table-based attacks, the server S, for each user U, should store a string s_U (called *salt*), chosen at random, along with the hash of the user's password computed as $H(pw_U||s_U)$. This also makes unlikable password files stolen from different servers even in case of password reuse. Observe that, despite the use of salted hashes, an attacker, able to steal the password file and the salts stored in S, is still able to perform a dictionary attack.

Another mechanism to further increase the attacker's workload is to include in the hash computation a short string, say p_U, chosen at random for each user U, called *pepper*. Thus, the server S, for each user U, stores the hash computed as $H(pw_U||s_U||p_U)$. However, differently from salts, peppers are not locally saved, but they must be guessed every time users need to authenticate. Hence, a pepper should be short enough (a typical choice is 12 bits (Boneh and Shoup, 2020)), so that it can be easily brute-forced by the server when a user needs to authenticate, without introducing relevant delays. However, a 12-bit string used as pepper slows down an offline dictionary attack by a factor of 4096, thus increasing the time needed to perform such an attack.

We have also to point out that hash algorithms are designed to be quickly computed, thus making the dictionary attack less computationally expensive. To increase the computational effort of an attacker, servers might replace hash functions with *key derivation functions*, such as PBKDF2 or BCrypt, which are cryptographic one-way functions whose computation is much slower than standard hash functions. The basic idea is the following. Suppose a hash function takes x milliseconds to be computed on a single input. The adoption of slower functions requiring, for instance, $10,000$ times the time x, would not significantly impact the user experience during the authentication process. However, it would increase the time to perform a dictionary attack by a factor of $10,000$, thus increasing the attacker's workload.

Observe that the effect of key derivation functions is the same as the pepper. The only difference is that the delay introduced by the former is deterministic.

To conclude, we want to highlight that there exists another definition of pepper that is a string stored in a more secure and different location with respect to the password file. However, this secure location is not always available and if it is available we might store directly the password file on it. Then, we do not consider this definition.

4 THE PROPOSED APPROACH

Consider a password-based authentication scheme adopting both salt and pepper (see Section 3). The idea of our approach is to strengthen the robustness of the scheme against off-line dictionary attacks by adding a new value, called *ginger*, which is concatenated with the password along with the salt and the pepper, when applying the key derivation function. However, the idea of the ginger is not a trivial extension of salt/pepper, because it incorporates a number of relevant differences, which make the scheme definitely more robust against off-line dictionary attacks, as we extensively prove in Section 6.

The main difference is that, while the salt is kept server-side, the ginger is stored client-side and sent, on the fly, during the authentication process. This

way, the sole data exfiltration from the server is not enough for the attacker to gain success. Moreover, a refresh mechanism (described in more detail below), results in an expected considerable reduction of the time window in which the attacker should successfully complete the dictionary attack. A nice feature of our scheme is that, unlike the pepper, the ginger does not introduce additional workload server-side, thus scalability is preserved. Moreover, the adoption of the ginger does not conflict with salt and pepper. Therefore, all these mechanisms can be combined to obtain a net increase of security.

From a formal point of view, the ginger introduces a significant change of authentication. Indeed, authentication becomes *stateful*, while password-based authentication even with salt and pepper is *stateless*. Specifically, the ginger represents a sort of *safe* state from which the client moves towards another safe state at the next authentication. This involves the fact that the client must store state information and, thus, a number of practical implications, which we will discuss in Section 5. However, as shown in that section, these implications do not represent actual limitations or technological obstacles.

Now, we focus on the detailed description of the proposed protocol. Our protocol involves two actors, a client C and a server S, and is composed of two phases, the *registration phase* and the *authentication phase*.

4.1 Registration Phase

This phase starts with the generation by C of a random value g_C, called *ginger*. Then, C sends it along with the password pw_C to the server S. The value g_C is stored locally by the client, while the password is not (as usual, password is required to be mnemonic, even if robust).

When S receives pw_C and g_C, it generates a random salt s_C and a random (in a small domain) pepper p_C. At this point, it stores $h_C = F(pw_C||s_C||p_C||g_C)$ in the password file, where F is the chosen key derivation function. The salt s_C is stored along with h_C, while the pepper p_C and the ginger g_C are not stored by S.

4.2 Authentication Phase

To authenticate with S, C selects the current ginger g_C to use for the current authentication. In addition, it generates a new ginger $\bar{g}_C$ to use for the next authentication. Both the gingers g_C and $\bar{g}_C$ are provided, along with the password pw_C, to server S. Finally, the client removes the old ginger g_C from the local memory and maintains the new ginger $\bar{g}_C$.

At this point, S uses the ginger g_C and the password pw_C to compute $h_C = F(pw_C||s_C||p_C||g_C)$ and checks that this value is equal to the value of h_C currently stored. To do this, as explained in Section 3, the server computes h_C for all the possible values of p_C. If the check passes, the authentication performs successfully. Then, the value of h_C is updated with $\bar{h}_C = F(pw_C||s_C||\bar{p}_C||\bar{g}_C)$ where $\bar{p}_C$ is a new random pepper. The gingers g_C and $\bar{g}_C$, and the pepper $\bar{p}_C$ are discarded by S.

5 PRACTICAL ASPECTS

The protocol presented in Section 4 is discussed at an abstract level and does not include technological aspects. As usual for abstract definitions of authentication schemes, it considers a single client. In this section, we deal with some practical aspects of the authentication process, starting by considering that, nowadays, the user owns multiple devices (e.g., a smartphone, a tablet, a laptop, a PC).

As our solution is stateful, we have to face the problem of storing the safe state (i.e., the ginger useful for the next authentication) in *all* the devices of the user.

From a conceptual point of view, this is obviously not an issue, as the devices belong to the same entity (i.e., the user). However, from a practical point of view, we have to design a feasible and secure solution for the device state synchronization.

This can be done by simply considering the fact that, nowadays, the smartphone can be considered as an extension of our identity over the digital domain. The practical counterpart of this fact is that we are forced to keep the smartphone always close to us and/or under control.

Therefore, it is quite natural to consider the smartphone as the master storing point of the ginger. This means that, when the user wants to authenticate with another device, they must keep the smartphone in proximity, so that the ginger value can be synchronized manually or by any machine-to-machine protocol, such as Bluetooth (or other options, discussed below).

Clearly, possible apps or browser extensions could be installed in every device to implement the features of our protocol.

In more detail, we show how the two phases of Section 4 are instantiated to deal with this scenario.

5.1 Registration Phase

When U wants to register with S, preliminary, they register the domain of S on the smartphone app. Then, the app generates the ginger g_U and associates it with the domain of S.

At this point, g_U is transferred from the smartphone to the laptop. We consider three options according to the capabilities of the smartphone and the laptop.

Option 1: Typed Ginger. This is the basic and less usable option but compatible also with legacy devices. In this case, the ginger is generated on the smartphone as a string and typed by the user in a registration form.

Option 2: Camera-based Ginger. In this case, the ginger is encoded on the smartphone app through a QR code. This QR code is then scanned by the camera of the laptop and automatically provided to the server.

Option 3: Bluetooth-based Ginger. The last option is to use a compatible browser on the laptop that receives the ginger by the app through Bluetooth.

The ginger g_U along with the password pw_U typed by the user on the laptop are provided to the server.

At this point, the protocol performs exactly as in Section 4.1, i.e., S generates a random salt s_U and a random pepper p_U. Then, it stores $h_U = F(pw_U||s_U||p_U||g_U)$ and the salt s_U, and discards the pepper p_U and the ginger g_U.

5.2 Authentication Phase

The authentication phase follows the trace of the registration phase. First, U selects on the smartphone app the domain of S to obtain the current ginger g_U to use for this authentication. In addition, the app generates a new ginger $\bar{g}_U$ to use for the next authentication. Both the gingers g_U and $\bar{g}_U$ are sent to the laptop through one of the three options described in Section 5.1. Finally, the app removes the ginger g_U from the smartphone and maintains the new ginger $\bar{g}_U$.

At this point, both the gingers are provided along with the password pw_U by the laptop to the server S through the login form. S uses the ginger g_U to compute $F(pw_U||s_U||p_U||g_U)$ and checks that this value is equal to the value of h_U currently stored.

If the check passes, the authentication performs successfully. Then, the value of h_U is updated with $\bar{h}_U = F(pw_U||s_U||\bar{p}_U||\bar{g}_U)$.

Due to the statefulness of our protocol, the other practical aspect we have to face is what happens if the user loses the current state. This may happen for example in case of crash of the smartphone. The solution to this problem is trivial. Indeed, in this case, the user is required to perform the registration phase again.

6 SECURITY ANALYSIS

Through this section, we evaluate the security of the proposed approach by comparing it with three state-of-the-art password-based authentication schemes. The three approaches we analyze represent the most widely adopted for authentication in web applications (Shadow, 2022). The results of our analysis are summarized in Table 1.

6.1 Standard Password-based Authentication

In the following, we compare our solution with a standard password-based authentication scheme.

For the latter, we assume that, for each user U, the server stores $F(pw_U||s_U||p_U)$, along with the salt s_U in the clear (as described in Section 3). On the other hand, in our solution, for each user U, the server stores $F(pw_U||s_U||p_U||g_U)$ (along with the salt s_U).

Concerning the password pw_U, being it a memorized secret, it is common to assume that it is not a random string (with high entropy (Taha et al., 2013)). Therefore, we assume that there is a non-negligible probability that pw_U can be found within the attacker's dictionary. As for the salt s_U, additional assumptions are not needed for our purposes. As for the pepper p_U and the ginger g_U, we consider $p_U \in \{0,1\}^{d_p}$ and $g_U \in \{0,1\}^{d_g}$, denoting with d_p and d_g the length of their bit-strings.

As highlighted in the previous sections, the pepper is neither stored server-side nor client-side, while the ginger is stored only client-side. Indeed, the pepper must be brute-forced by the server at each authentication, while the ginger is provided by the user to the server at each authentication. Therefore, for usability reasons, the pepper search space should be sufficiently small, while, in principle, no such constraints are needed for the ginger. Hence, it is fair to assume that $d_g \gg d_p$.

Given the above assumptions, we analyze the security of the two considered protocols in two cases: (i) against an attacker able to compromise the server and thus able to steal the list of the hashed passwords of all the users, (ii) against an attacker as powerful as the previous one, with the additional capability to compromise the personal device belonging to a user, whose password is in the stolen list.

Consider now case (i). Once an attacker has obtained a list of hashed passwords, they can mount a

Table 1: Search space dimension to guess the password.

Approach	Server-side	Server and Client-side (single client)
Our proposal	$n \cdot d_D \cdot d_p \cdot d_g$	Worst timeline attack: $d_D \cdot d_p$ Best timeline attack: $d_D \cdot d_p \cdot d_g$
Standard password-based authentication	$n \cdot d_D \cdot d_p$	$d_D \cdot d_p$
Authentication with password manager (attack on master password)	$n \cdot d_D \cdot d_p$	$d_D \cdot d_p$
OTP-based authentication	$n \cdot d_D \cdot d_p$	$d_D \cdot d_p$

dictionary attack to attemp to obtain the memorized passwords. We denote by n the length of the above list, and by d_D the length of the dictionary used by the attacker. In case the server implements a standard password-based authentication scheme, the dimension of the search space is given by $n \cdot d_D \cdot d_p$. Instead, with our approach, the dimension of the search space is given by $n \cdot d_D \cdot d_p \cdot d_g$. Thus the computational effort made by the attacker is increased by a multiplicative factor d_g. Moreover, the ginger dimension d_g can be chosen in such a way that a dictionary attack is not feasible.

To be fair, we have to point out that, even adopting a standard password-based authentication scheme, a dictionary attack may be made unfeasible by forcing each user to change their password periodically. This, in principle, limits the time available to the attacker to carry out the attack. However, in practice, this is never the case. In fact, when users are forced to change their passwords, they tend not to change them radically but to keep the old ones, changing at most a few characters in a predictable manner (Zhang et al., 2010). Hence an attacker can easily find out the new password from the old one, just by doing a few online attempts.

Consider now case (ii). As argued in the previous case, if our approach is adopted, for an attacker to be able to succeed in a dictionary attack, it is necessary to brute-force g_U for each of the n users in the list. Alternatively, an attacker could compromise the personal devices belonging to n users in order to steal their gingers. However, this is still an unfeasible attack. Hence the probability for an attacker to succeed in a massive dictionary attack is negligible.

Nevertheless, it is interesting to analyze the security of our solution compared to a standard password-based approach against an attacker at least as powerful as the attacker in case (i), but with the additional capability of compromising a user's personal device. Different from the previous case, here we assume the attacker is interested in discovering a single user's password. Hence the search space dimension for both the approaches is the same as in the previous case, except for a multiplicative factor n.

Considering the standard password-based approach, an attacker cannot gain any additional information compromising the client and thus the search space dimension remains unchanged (i.e. $d_D \cdot d_p$). On the contrary, considering our approach, two cases may occur. Suppose the attacker is able to compromise the server in the instant $T1$ and subsequently, the attacker is able to compromise the user's personal device at the instant $T2$, where $T1 < T2$. We denote by g_U^s, the ginger used server-side at the instant $T1$, and by g_U^c the ginger stored client-side at the instant $T2$. Suppose the first user authentication, after $T1$, takes place at the instant $T3$. As explained in Section 4, a new authentication implies that the user's personal device stores a new ginger in place of the old one.

If $T1 < T3 < T2$, then the ginger g_U^c stolen by the attacker at the instant $T2$ differs from the ginger g_U^s, since the victim has authenticated in between $T1$ and $T2$. Hence, the search space dimension for a dictionary attack remains equal to $d_D \cdot d_p \cdot d_g$.

If $T1 < T2 < T3$, then the ginger g_U^c stolen by the attacker at the instant $T2$ coincides with the ginger g_U^s, since the victim did not authenticate in between $T1$ and $T2$. Therefore, the search space dimension for a dictionary attack is narrowed down to $d_D \cdot d_p$. Observe that, in this case, we obtain the same search space dimension the attacker would have by attacking a standard password-based approach.

The above reasoning holds for $T2 > T1$. However, it is easy to realize that it applies also when $T1 > T2$ with the same results.

6.2 Authentication with Password Manager

The second approach we consider to be compared with our solution is based on password managers. A password manager is a repository in which users store, in encrypted form, the passwords of their accounts of different websites. All these passwords are encrypted and decrypted locally through a master password. In this way, users have to remember just a single password to authenticate with all the websites. Furthermore, modern password managers auto-fill the

login form. Then, since users do not need to manually type their passwords, they can be chosen randomly with high entropy. This makes dictionary attacks ineffective even if the servers of the websites are compromised. However, this is not enough since the effort of the attacker moves from compromising the website servers to compromising the password manager. Indeed, in case the encrypted passwords (stored in the password manager) are stolen by the attacker, a dictionary attack can be performed on the master password. We observe that several data breaches on commercial password managers occurred in recent years (Last Pass, 2022).

Observe that, since the encryptions/decryptions are performed client-side, typically, in real-life applications, no salt or pepper mechanism is introduced to avoid an excessive burden. However, in favor of security, we assume that they are enabled. Then, by considering a server-side attack (case (i) of Section 6.1), the size of the search space is the size of the dictionary $n \cdot d_D \cdot d_P$. No further advantage is obtained by compromising also the client (case (ii) of Section 6.1), so that the size of the search space is $d_D \cdot d_P$ (by considering the compromise of a single client). Therefore, our approach outperforms the authentication with password manager in both cases (i) and (ii).

6.3 OTP-based Authentication

Finally, we compare our solution with the OTP-based authentication. An OTP (One-Time password) is a secret value used as a second factor during the authentication phase. We consider two versions of OTP, which are the main ones adopted in real life: SMS-based or app-based. In the first version, the user performs a first authentication with the server through a standard password. The server generates a random code and sends it to the user through an SMS. The user provides this code to the server and the authentication concludes. This second factor proves the ownership of a given phone number provided by the user to the server during a registration phase. In the second version, during the registration phase, the server provides a seed (secret) to the user (typically, through a QR code that is scanned by the smartphone) to be stored in a dedicated app. Then, through this seed, the user generates pseudo-random codes to be provided, along with the password, during the authentication. Since the server owns the same seed, the codes can be verified.

To be fair, in our comparison, we assume that along with the OTP, the server applies the standard salt and pepper approaches as in Section 6.1.

The problem with the OTP, in both versions, is that it does not protect the password. Indeed, by considering the case (i) of Section 6.1, if the attacker compromises the server, the size of the search space is again $n \cdot d_D \cdot d_p$ and does not depend on the OTP itself. Even though the attacker is not able to authenticate without the OTP, this represents a critical vulnerability since, commonly, passwords are re-used for different services (Das et al., 2014).

Considering the case (ii) of Section 6.1 (i.e., a compromise both client and server side), the attacker is also able to authenticate with the server with the same effort of case (i) by stealing the OTP. By considering a compromise of a single client (as in Sections 6.1 and 6.2), the size of the search space is $d_D \cdot d_p$.

To enrich the discussion, we report some attacks that can be performed client-side to steal the OTP.

Concerning the app-based version, the attacks can be performed through malware installed on the smartphone (as with our solution). On the other hand, regarding the SMS-version, several SIM-swapping attacks (Jover, 2020), in which the adversary is able to obtain the control of the phone number of a victim, are shown to be effective.

7 CONCLUSIONS

Despite new promising attempts (Chakraborty and Bugiel, 2019) to introduce password-less authentication, password-based authentication schemes remain the most widely used authentication method. However, the growth of data breaches combined with the fact that users do not follow the best security practices when selecting their passwords, makes password-based authentication vulnerable to dictionary attacks. To mitigate the problem, traditional solutions consist in the introduction of salts and peppers possibly combined with key derivation functions. However, this is not conclusive since by employing dedicated hardware, it is still possible to discover users' passwords. Then, in this paper we propose a new authentication approach (i.e., the introduction of a ginger) to strengthen password-based authentication schemes that can be used in combination with salts and peppers. An important point is that our solution does not introduce a relevant overhead neither client-side nor server-side. From the security point of view, we observed, in Section 6, that we outperform state-of-the-art authentication schemes in case of server-side compromise. Furthermore, if the client, along with the server, is compromised, we obtain an improvement with respect to traditional schemes in the best case and we have the same robustness as other schemes

in the worst case. Currently, we are implementing a prototype of our solution by relying on the Web Bluetooth API supported by Google Chrome.

REFERENCES

Andrews, R., Hahn, D. A., and Bardas, A. G. (2020). Measuring the prevalence of the password authentication vulnerability in ssh. In *ICC 2020 - 2020 IEEE International Conf. on Communications (ICC)*, pages 1–7.

Boehm, O., Caumanns, J., Franke, M., and Pfaff, O. (2008). Federated authentication and authorization: A case study. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 356–362. IEEE.

Boneh, D. and Shoup, V. (2020). A graduate course in applied cryptography. *Draft 0.5*.

Bonneau, J., Herley, C., Oorschot, P. C. v., and Stajano, F. (2012). The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pages 553–567.

Buccafurri, F., De Angelis, V., and Nardone, R. (2020). Securing mqtt by blockchain-based otp authentication. *Sensors*, 20(7):2002.

Chakraborty, D. and Bugiel, S. (2019). Simfido: Fido2 user authentication with simtpm. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2569–2571.

Chiasson, S., Stobert, E., Forget, A., Biddle, R., and Van Oorschot, P. C. (2011). Persuasive cued clickpoints: Design, implementation, and evaluation of a knowledge-based authentication mechanism. *IEEE Transactions on Dependable and Secure Computing*, 9(2):222–235.

Das, A., Bonneau, J., Caesar, M., Borisov, N., and Wang, X. (2014). The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26.

Eldefrawy, M. H., Alghathbar, K., and Khan, M. K. (2011). Otp-based two-factor authentication using mobile phones. In *2011 eighth international conference on information technology: new generations*, pages 327–331. IEEE.

Jens 'atom' Steube, Gabriele 'matrix' Gristina (2015). Hashcat advanced password recovery. https://hashcat.net/hashcat/. Last checked on Aug 01, 2022.

Jover, R. P. (2020). Security analysis of sms as a second factor of authentication. *Communications of the ACM*, 63(12):46–52.

Kumar, H., Kumar, S., Joseph, R., Kumar, D., Singh, S. K. S., Kumar, A., and Kumar, P. (2013). Rainbow table to crack password using md5 hashing algorithm. In *2013 IEEE Conference on Information & Communication Technologies*, pages 433–439. IEEE.

Last Pass (2022). Lastpass security history. https://www.lastpass.com/it/security/what-if-lastpass-gets-hacked/. Last checked on Aug 01, 2022.

Lenz, T. and Zwattendorfer, B. (2015). Enhancing the modularity and flexibility of identity management architectures for national and cross-border eid applications. In *11th International Conference on Web Information Systems and Technologies*, pages 123–143. Springer.

Li, Z., He, W., Akhawe, D., and Song, D. (2014). The {Emperor's} new password manager: Security analysis of web-based password managers. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 465–479.

Manber, U. (1996). A simple scheme to make passwords based on one-way functions much harder to crack. *Computers & Security*, 15(2):171–176.

Mohammed, S., Ramkumar, L., and Rajasekar, V. (2017). Password-based authentication in computer security: Why is it still there. *SIJ Trans. Comput. Sci. Eng. Its Appl*, 5:33–36.

NordPass (2022). Top 200 most common passwords. https://nordpass.com/it/most-common-passwords-list/. Last checked on Aug 01, 2022.

OpenWall (1996). John the ripper. https://www.openwall.com/john/. Last checked on Aug 01, 2022.

Pasquini, D., Cianfriglia, M., Ateniese, G., and Bernaschi, M. (2021). Reducing bias in modeling real-world password strength via deep learning and dynamic dictionaries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 821–838.

Shadow, D. (2022). Account Takeover in 2022. Technical report.

Shirvanian, M., Jareckiy, S., Krawczykz, H., and Saxena, N. (2017). Sphinx: A password store that perfectly hides passwords from itself. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1094–1104. IEEE.

Shirvanian, M., Price, C. R., Jubur, M., Saxena, N., Jarecki, S., and Krawczyk, H. (2021). A hidden-password online password manager. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1683–1686.

Taha, M. M., Alhaj, T. A., Moktar, A. E., Salim, A. H., and Abdullah, S. M. (2013). On password strength measurements: Password entropy and password quality. In *2013 International Conference on Computing, Electrical and Electronic Engineering (ICCEEE)*, pages 497–501. IEEE.

Tao, H. (2006). *Pass-Go, a new graphical password scheme*. PhD thesis, University of Ottawa (Canada).

Yao, F. F. and Yin, Y. L. (2005). Design and analysis of password-based key derivation functions. In *Cryptographers' Track at the RSA Conference*, pages 245–261. Springer.

Zhang, Y., Monrose, F., and Reiter, M. K. (2010). The security of modern password expiration: An algorithmic framework and empirical analysis. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 176–186.

Zhang, Z., Liu, P., Wang, W., Li, S., Wang, P., and Jiang, Y. (2020). High-performance password recovery hardware going from gpu to hybrid cpu-fpga platform. *IEEE Consumer Electronics Magazine*, 11(1):80–87.