The Weakest Link: On Breaking the Association between Usernames and Passwords in Authentication Systems

Eva Anastasiadi[1,2], Elias Athanasopoulos[3] and Evangelos Markatos[1,2]
[1]Computer Science Department, University of Crete, Greece
[2]Institute of Computer Science, Foundation for Research and Technology Hellas, Greece
[3]Computer Science Department, University of Cyprus, Cyprus

Keywords: Authentication, Passwords, Database Leaks.

Abstract: Over the last decade, we have seen a significant number of data breaches affecting hundreds of millions of users. Leaked password files / Databases that contain passwords in plaintext allow attackers to get immediate access to the credentials of all the accounts stored in those files. Nowadays most systems keep passwords in a hashed salted form, but using brute force techniques attackers are still able to crack a large percentage of those passwords. In this work, we present a novel approach to protect users' credentials from such leaks. We propose a new architecture for the password file that makes use of multiple servers. The approach is able to defend even against attackers that manage to compromise all servers - as long as they do not do it at the same time. Our prototype implementation and preliminary evaluation in the authentication system of WordPress suggests that this approach is not only easy to incorporate into existing systems, but it also has minimal overhead.

1 INTRODUCTION

User authentication is necessary in all modern systems and platforms that include personalization or store any form of private data. Getting access to such systems is very attractive to attackers, who systematically target the authentication system of web applications and other system services. Most systems use passwords to authenticate users. Traditional systems require username-password pairs for authentication, while others use them as basis and add supplementary techniques, such as mobile authenticators, to enhance security (multifactor authentication). Even single sign on schemes require a proper authentication from a main authority, that commonly makes use of passwords too. It is clear that, if attackers manage to leak usernames and passwords, all of those authentication schemes lose an important security factor.

There have been several cases where attackers successfully managed to steal password files or Databases even in some of the most popular companies (Thomas et al., 2017). Although it seems that most systems do not keep passwords in plaintext, which would turn all user accounts into easy targets, this is far from true. Indeed, in March 2019, Facebook revealed that they kept a log file containing 200 to 600 million plaintext passwords, accessible by more than 2000 developers (Krebs, 2019).

On the positive side, keeping passwords in plaintext is a diminishing practice. Indeed, most systems today keep password *hashes*. Since these hashes are produced by one-way hash functions, attackers can not easily retrieve the plaintext passwords. Mounting a dictionary attack might however reveal *some* of the passwords. To reduce the effectiveness of dictionary attacks, several authentication servers use *slow* hash functions and "salts" (Provos and Mazieres, 1999; Morris and Thompson, 2002). Although these approaches delay dictionary attacks, attackers are still able to find the passwords as long as they are willing to invest a lot of computing capacity (Shi et al., 2012).

To reduce the usefulness of leaked password files/ Databases, recent methodologies divide the responsibility of the authentication among multiple servers. This way, no single server has all the information required for a successful authentication and attackers need to compromise all servers to get a user's credentials. Since that is not likely, such methodologies protect from password file leaks. However, most of these approaches assume additional storage, complex systems and encryption mechanisms, or they require more user actions, which can be discouraging. This paper also proposes using two servers:

- The Authentication Server (AS for short), which stores the usernames but not the passwords.
- The Password Server (PS for short), which stores the passwords, but not the usernames.

Anastasiadi, E., Athanasopoulos, E. and Markatos, E.
The Weakest Link: On Breaking the Association between Usernames and Passwords in Authentication Systems.
DOI: 10.5220/0011276900003283
In *Proceedings of the 19th International Conference on Security and Cryptography (SECRYPT 2022)*, pages 560-567
ISBN: 978-989-758-590-6; ISSN: 2184-7711
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Contrary to previous approaches, our system is able to defend against attackers even if both servers are compromised - as long as this does not happen at the same time. Our contributions are the following:

- We propose a novel approach that protects against password file leaks by storing usernames and passwords into two different servers.
- We show that our approach is able to protect from such leaks even if both servers are compromised, as long as they are not concurrently compromised.
- We demonstrate that our approach can be easily integrated in existing systems by implementing it using the WordPress authentication system.
- We evaluate the performance of our prototype implementation and show that the performance overhead is minimal.

2 RELATED WORK

One of the most well-known methods to detect password leaks using two servers was the Honeywords approach, proposed in 2013 on the ACM SIGSAC conference (Juels and Rivest, 2013). This approach suggests that multiple decoy passwords, the "Honeywords", are stored alongside the real password of each user. The system requires a second server, the Honeychecker, storing an index for each user record, that indicates the user's real password. Although this is an innovative approach, the key issue here is to find an algorithm to generate "realistic" honeywords.

Many approaches have been proposed on that matter, but most were found inefficient. In IJREAM 2016, it was proposed to produce the decoy passwords by modifying the real one(K. Naik, 2016). Unfortunately this does not provide protection for users that include easy to guess patterns (i.e. birth dates). Later it was proposed that, instead of the real password, the *hash* should be modified (Shi and Sun, 2021). However, similar hashes might have a very different plaintext password origin. If the attacker manages to crack a hash, then a given decoy password might look nothing like a real password. In 2021 (Dionysiou et al., 2021) a representation learning technique is proposed. It uses a Dataset of user passwords to generate the decoy passwords. On the downside, it requires a large amount of training data and ML infrastructure to produce realistic Honeywords. An analysis on the multiple aspects of Howneywords was presented in the NDSS symposium and reveals that all studied approaches in that paper fail to provide the expected level of protection (Wang et al., 2017).

Inspired by Honeywords, the *Amnesia* mechanism (Wang and Reiter, 2021) proposed using probability theory and only one server. Amnesia is on most occasions able to recognize the correct password among all the honeywords by tracking some indicators of what users usually enter in order to authenticate, without letting an attacker be aware of which is the correct user password. However, under specific circumstances, Amnesia allows attackers to login with decoy passwords, and it still requires storage capacity and realistic Honeyword generation.

In also 2013, SAuth paper (Kontaxis et al., 2013) proposed involving multiple services in the authentication process. Each service separately authenticates users using its own method and if they all agree on the users' identity they grant access to the requested service. Unfortunately, users need to remember their credentials to several services when authenticating, which might lead them to password reuse. Besides that, it is unsafe to rely on other systems due to the inability to control their security and availability.

In the spirit of dividing the responsibility of the authentication among different devices, Pythia-PRF service (Everspaugh et al., 2015) suggests using an external trusted server as a crypto-service that stores a specific key to encrypt users' passwords. Several other techniques also suggest keeping the password file entries encrypted, so that attackers need to find the key to be able to read them. Other approaches do not require additional server, but rely on a trusted hardware device (USB stick, PUF, etc) which is responsible for storing the encryption keys (Cvrcek, 2014; Mohammadinodoushan et al., 2019; Assiri et al., 2020). Unfortunately, such practices may fail if the hardware device used by the server is lost or broken.

In this paper, we explore a non encryption based approach. The main reason for this choice is that in order for encryption to be effective, it needs very careful key management. If, for example, the key is stolen, then it is not clear how effective would be the encryption on providing protection to the users. To make matters worse, if the key is lost, then data might become unusable.

Much like Honeywords, our approach makes use of two servers, but it is inspired by a different idea. In all cases studied so far, the password file contains one line for each user. This line has the username and the salted, hashed, or encrypted password. That way, once the attackers manage to crack a password, they are immediately aware of the password owner. In our approach we keep usernames and password in different servers, breaking this association. The details will be discussed in the following paragraphs, but let us first define the threat actors that we are going to face.

3 THREAT MODEL

As we mentioned, the AS (Authentication Server) stores the user names, while the PS (Password Server) stores the passwords. In this paper we assume two threat models:

- **Threat Model 1:** The attacker is able to read the permanent storage of any of the servers, but not both. This model has been widely adopted (Juels and Rivest, 2013; Kontaxis et al., 2013).
- **Threat Model 2:** The attacker can read the permanent storage of both servers but not concurrently. Consequently, the attacker can read the usernames (from the AS) and find information about passwords (from the PS), but these discoveries can not occur at the same time.

The main motivation behind those models is that the servers are probably implemented on top of different computers, running different Operating Systems and software. The AS is a customer-facing server, running a general-purpose OS, while the PS communicates only with AS and can be equipped with a stripped-down OS. Thus, vulnerabilities that might lead to exposure of data stored in one server's disk are probably not present in the other server.

In addition, regarding the second model, circumstances that lead to server data leaks are short-lived and rather unlikely. The attacker may be able to exploit a vulnerability and leak information at some point, but this vulnerability will sooner or later be discovered and fixed. Materializing for both servers at very close time intervals is very unlikely, especially after assuming that they have different configurations.

Much like most of the previous work, we assume that the attackers *do not* have access to the main memory of the servers. Indeed, if attackers had such access, they would be able to see the plaintext username-password pairs of the users who were trying to log in or register at that time. In that case, attackers would not need to steal the password file and reverse engineer the hashed passwords.

We also take as granted that the communication between the clients and the AS uses some form of encryption much like the one used in HTTPS. Similarly, the AS - PS messages are sent using a secure channel. That way, usernames and passwords are protected from eavesdroppers on the network.

4 DESIGN

We will now describe the general architecture and design of our system, when used by traditional systems.

4.1 Breaking the Username-password Link

Traditional systems use a password file or Database, where both usernames and (hashed salted) passwords are stored. If the file is leaked, through a combination of dictionary and brute-force approaches, the attacker will eventually manage to crack some passwords. Once that happens, the attacker will immediately know the owner of each password, because the information is stored in the same line/record of the server. In this paper, we break this obvious association, by storing usernames and passwords in different servers. If the attacker manages to get a copy of the file stored in the AS, she will be able to have a list of all user names, but no information about their passwords, and vice versa.

4.2 What Will Each Server Store?

As we mentioned, the main idea of our system requires the AS storing the usernames and the PS holding the password-related information. Although this can confuse the attacker who will not be able to associate usernames and passwords after getting a copy of the one server's disk, we still have to explain

> How do ***we*** know which username corresponds to which password?

Figure 2 shows the contents of the AS and the PS. We can see that next to each username there is a "token". We can imagine the token as a "pointer" which connects the user name with its password. Indeed, we see that token1 is next to username1 in the AS, and next to password1 in the PS. This means that those credentials belong to the same user, since there are both connected with the same token.[1] Similarly, password2 connects to username2 via token2, etc.

In this way, both AS and PS have a common point of reference (i.e. the token) for the same user. They can use that for their communication during authentication. An attacker with access to only one of the two servers does not know the pieces of information stored in the other server. This attacker is therefore unable to use the token to find the missing part of the credentials.

An important factor for this approach to be effective is that the PS is responsible for only storing and checking a user's password. If the PS held additional entries related to the person's identity, such as the email addresses, that could be a strong hint for an

[1] For the purposes of this description, we omit the details about hashes and salts. Once the connection using tokens is made clear, we will add how salts and hashes get used.

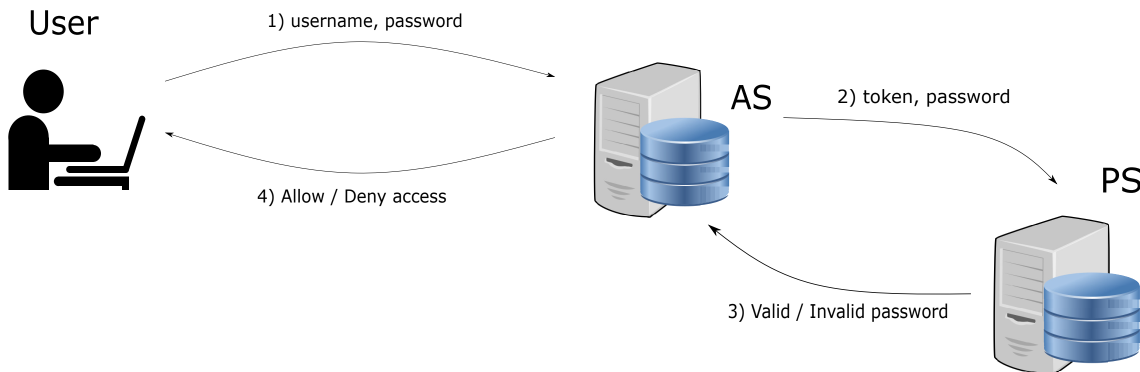

Figure 1: Message flow during user authentication between the User, the AS and the PS.

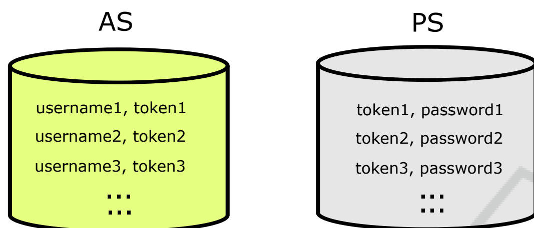

Figure 2: Contents overview for the AS and the PS.

attacker accessing the PS in guessing the corresponding username. Therefore, we keep this information in the AS. Of course, the PS must be able to receive the password and a token and decide whether it should allow or deny access. Consequently, if another password technique is used and requires storing more information, such as storing a salt for each entry, then this information is also kept in the PS.

4.3 User Authentication

Figure 1 shows the steps in the process of user authentication:

1) The AS receives the authentication request from the user containing the credentials (step 1).
2) The AS searches its Database to find the entry with the given username. If this entry exists, it sends the newly received password to the PS alongside the token (pointer) found in that entry of the AS (step 2).
3) The PS searches for the entry with the same token in the password Database. If this token exists, it checks whether the given password matches the one stored. Then, the PS responds back to the AS with the suitable message: i.e. “accept” if the passwords match, “reject” otherwise (step 3).
4) The AS allows/denies the user depending on the response it received on step 3 above (step 4).

As mentioned, we used gRPC to enable the TLS-protected communication needed between AS and PS.

4.4 Registration

There are several popular system techniques for interaction with a new user during registration. The simplest registration model consists in collecting both username and password by just asking the user to provide those values. Other approaches, as the one used in the default authentication of WordPress, first verify the email of the user. Additionally, systems can enforce their password policies during registration so that users are not allowed to use very common and easy-to-guess passwords. However, this has to be done sparingly because of the trade-off between usability and security (Sahar, 2013), which can turn strong security policies into weak security.

Independently from the variations in the registration process, the system must finally have a pair consisting of the new user’s username and password and then make sure that those credentials get stored in the appropriate positions. A high-level overview of the registration process contains the following steps:

1. AS receives the username - password pair from the user.
2. AS generates a random string (the token).
3. AS stores the given username alongside this token in its Database.
4. AS sends the password and token to the PS.
5. PS stores this pair in the password file / Database.

4.5 Password Change

Having covered the initialization phase, we will now describe what happens when users forget their passwords. If users forget their passwords, the AS first needs to establish their identity. This usually happens

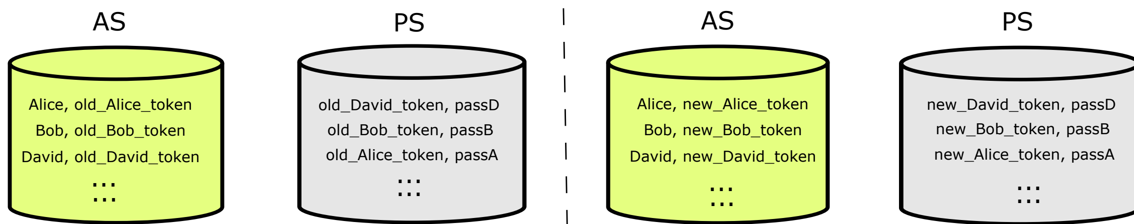

Figure 3: User fields stored in AS and PS before (left half of the figure) and after (right half of the figure) a token update.

by sending a token to the user's mobile phone or email address. The actual way with which this establishment happens is beyond the scope of the paper. In all cases, once the system verifies the identity of the user who made the request, it prompts the user for the new one. The steps that follow this procedure are similar to the registration phase.

4.6 Token Update

So far, the description of our approach covered **Thread Model 1**: i.e. the attacker managed to leak the data of one server only. Now let us try to address the second model: The attacker is able to leak the data from both servers but not at very close time intervals. To achieve this, we capitalize on the use of tokens. Up to this point, we seem to have implied that token1 is an entity that remains static and does not change. Actually, this is not the case. To be able to defend against **Threat Model 2** we frequently change the token. For example, although the initial information might be "(username1, token1) (token1, password1)", after a token update, it will be "(username1, token2) (token2, password1)". After another token update occurs, this information will change to "(username1, token3) (token3, password1)".

To explain it further, let us consider the following chronological sequence of events:

1. Mallory, an attacker, got access to the Database of the AS, by either an attack or a Database leakage. The last snapshot of the AS contents she was able to get is shown in the first Database of figure 3. Mallory did not have access to the PS at that time.
2. A token update resulted to the change of the tokens and, therefore, the servers' contents. The Databases now contain the information shown on the right half of Figure 3.
3. Mallory now manages to leak the Database of the PS. She has the contents of the first and the last (fourth) Database shown in Figure 3 available. However, although she is aware of all the stored usernames and passwords, she is unable to make the connections and construct the username-password pairs. Indeed, in the eyes of Mallory, Alice might own any of the passwords existing in the PS (passD, passB, passA, ...), because all of the tokens seem unfamiliar for her.

It is easy to imagine that the result would be the same if an attacker first leaked the Database of the PS and then, after the token update, the Database of the AS. Concluding, we saw how systems can get protected from an attacker with the ability to leak information from both servers at different time intervals. This attacker is able to know the usernames and password-related fields of all the registered users but, thanks to the token update, is unable to join them and get a user's credentials without starting testing the passwords found in the PS online. Such an attack would probably trigger an Intrusion Detection Mechanism from the first few online guesses.

4.7 Adding Honeypots

Assuming that Mallory manages to compromise both the AS and the PS she will have a set of usernames and a set of passwords. Although she will not be aware of the connection between them, she will be aware that all the usernames and passwords in her possession are *real* ones.

To confuse the attacker, we add several "fake" user names in the AS and several "fake" passwords in the PS (many more than the ones created by the actual users). To generate fake usernames and passwords we can use several techniques that have been studied for Honeytoken / Honeyword generation (Bercovitch et al., 2011; Dionysiou et al., 2021).

These fake usernames and passwords are stored alongside tokens that do not match with any password, or username respectively. If the attacker uses any of these usernames or passwords, the system will have the ability to immediately detect this abuse as their associated tokens will not match with anything. By adding several fake usernames and passwords we can, with high probability, make sure that if Mallory launches an online attack and tries to use any username or password she has leaked she will get detected from the first few tries.

5 IMPLEMENTATION

We adjusted the basic WordPress authentication system to use our design, performing some necessary changes when needed, but keeping the original user interface for all the authentication-related procedures. The implementation described below shows a practical use of our method providing a supplementary security mechanism to an already existing authentication infrastructure. We present an overview of the default WordPress authentication system and, at the same time, we mention the required changes to make this authentication system follow our design.

We should note that this approach can be combined with other authentication techniques, such as MFA and SSO. For example, if we added biometric characteristics as an additional authentication factor (2FA), the scheme would work in the same way, but it would also require the biometrics to match with the ones provided by the user. However, we present the traditional systems' method to make it more clear and easy to measure.

5.1 Server Roles

The default WordPress system uses only one server. All of the system's procedures take place there, and all the data are stored in its Database. However, our approach uses an additional server. In order to make minimal changes to the system, we decided that this unique server is the AS in our design and that the PS is another server whose sole responsibility is to perform actions related only to users' passwords. Every functionality that does not involve the user password or the hash of this password is performed by the AS exclusively. Therefore, there are no further modifications needed for those processes.

5.2 Database Contents

The default mechanism of WordPress has a Database that stores several tables. All those tables are now stored in the AS without any changes, except from the User table that needs some modifications. The fields of the User Table used to include an ID, a username, an email, an activation code, some metadata, the salt, and the hash of the "salted" password. The activation code field was used to identify the user while registering or changing the password (we will discuss those procedures in detail).

In our modification, PS holds the password-related information (the salt and the hash of the salted password), plus the random token that connects each entry with the one in the AS corresponding to the same user. AS also now needs to hold the token, but discards the hash and the salt. Therefore, we can find the new configuration in figure 4.[2]

AS user table
username
email
metadata
activation code
token

PS password table
token
salt
hash (password, salt)

Figure 4: Fields stored in the AS and the PS for the WordPress implementation.

The hash function we used is the PBKDF2 HMAC SHA-512 with 120,000 iterations (OWASP, 2021). This choice was decided based on its extensive use and effectiveness against brute force attacks. However, as we mentioned, a wide variety of password storing approaches could be used.

5.3 Server Communication

In our implementation, all the message and data transfers between the user, the AS and the PS need to take place using secure communication channels. We have kept all the default communication procedures between the user and the AS intact. We only have to ensure that information exchanged between AS and PS can be read by only those servers. To do that, we used the gRPC protocol with TLS-protected connections. The functionalities requiring such communication are described in the remaining paragraphs of the section.

5.4 Register a New User

When users try to register in the WordPress system, they first submit a username and an email address. An activation code is generated by the AS and is being sent to the given email address, in the form of a confirmation link. By visiting this link, users can provide a password of their choice.

Until a password is provided, the system has to store the email, the username and the activation code of the new user. In order to do that, the AS generates a random string, which is used as a (dummy) password. Consequently, the registration can then proceed as described in the 4.4 section. A main difference is that instead of storing the plaintext password, the PS

[2]If the WordPress system required more factors to authenticate the user, let us for simplicity imagine that the data related to that scheme would also be stored in the AS, and the AS would also check for this metric before or after checking the password using our scheme.

generates a salt and then stores the hash of the salted password and the salt. Also, AS stores the activation code, the metadata and the email additionally to the fields mentioned in 4.4.

When the user follows the activation link and submits the password, the activation code is erased from the user table and the ”Change user’s password” procedure is followed. The ”dummy” password is then considered as the old one, and gets replaced with the user’s input.

5.5 Change User’s Password

In order to change password, users follow the same procedure with registering. The only difference is that, instead of a completely new entry, servers have to search for the old user’s entry and replace the salt, the hash and the token. To be able to identify the user’s entry, the PS receives the password, the new *and* the old token.

5.6 Login User

The WordPress system follows the exact same steps with the ones mentioned in section 4.3. Figure 5 visually represents this process.

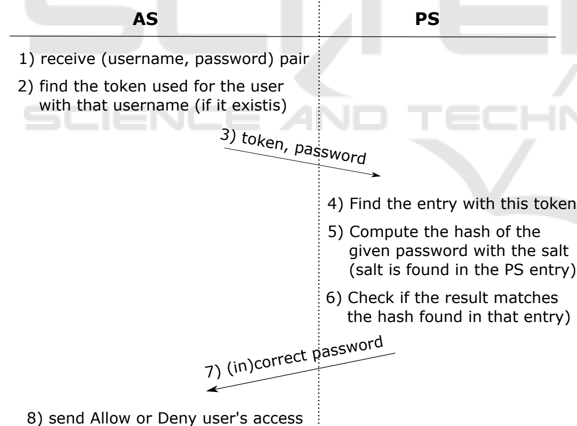

Figure 5: Steps needed to authenticate user (WordPress modification).

The only difference with section 4.3 is that instead of comparing the password with the one stored in the PS, now the PS has to first combine the salt stored in its Database with the one provided by the AS and compute the hash. Then, it can compare the two hashes and send the allow/deny access response.

5.7 Update Password Tokens

Token updates get invoked hourly, and run as a background process. We should however note that tokens can be updated more frequently as needed, depending on the system, but we leave studying the optimal frequencies with a more realistic implementation for future work. The process examines one entry of the AS at a time, finds the old token, and generates a new one. It sends the old and the new token to the PS which responds indicating whether the token update was successful. In that case, AS also renews the corresponding token. In order to avoid race conditions that might occur during an update and an authentication request we use synchronization techniques (such as locks). Finally, to make the probability of collisions exponentially small (two users concurrently using the same token) we let tokens be long random integers.

Table 1: Average overhead of basic authentication operations (in seconds). We use the PBKDF2 HMAC SHA-512 hash function with 120,000 iterations and we test it on 100 users.

Time per operation (sec)	Vanilla WordPress	Our approach
Login	0.0338	0.1679
Register	0.0705	0.1879
Change Password	0.0366	0.1598

6 MEASUREMENTS

We will now analyze the time overhead of our approach. In this work, we provide a preliminary evaluation of our implementation. Our goal is showing a simple prototype that, using very basic tools, performs authentication-related tasks efficiently. For the AS’s storage we use a Database (following the WordPress default implementation), while for the PS we use a simple text file handled by Python code. The measurements were taken using two Dell workstations (one for each server), connected and communicating with one another using 100 Mb/s Internet.

Table 1 presents the results of our measurements, after testing the first column’s scenarios for 100 user accounts. We measure three authentication operations: (i) login, (ii) register, and (iii) change password. Each operation is performed for 100 users, in a system with 150 already registered users. We report the average time needed per user for each procedure. We see that a simple “login” operation lasts 0.1679 seconds, which is practically unnoticeable for most users. The rest of the operations (i.e. “register” and “change password”) similarly last 0.18 and 0.15 seconds.

Concluding, our approach introduces some overhead for the server communication compared to the “Vanilla WordPress”, but still requires less than 0.2

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

seconds for each operation.

We leave an optimized commercial implementation for future work. With such an implementation we could study how this approach serves large-scale systems and measure the performance degradation as the number of users increases (hundreds, thousands, millions of users). Also, it would be interesting to study how the overhead that this approach introduces affects the user experience, and what is the trade-off with the extra security layer that is offered.

7 CONCLUSION

In this paper, we proposed a new decentralized design for user authentication that protects against password file leaks. To demonstrate the applicability of our approach, we embed it to the WordPress authentication system, showing that it can be easily integrated to existing systems. Future studies with a commercial implementation could measure the scalability and parallelization of the approach, the trade-off with the user experience, and factors that affect performance and security of the system (token update frequency, etc).

ACKNOWLEDGEMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830929 (CyberSec4Europe) and from Marie Skłodowska-Curie grant agreement No. 101007673 (RESPECT). This work reflects only the author's view. The Commission is not responsible for any use that may be made of the information it contains.

REFERENCES

Assiri, S., Cambou, B., Booher, D. D., and Mohammadinodoushan, M. (2020). Software implementation of a sram puf-based password manager. In *Science and Information Conference*, pages 361–379. Springer.

Bercovitch, M., Renford, M., Hasson, L., Shabtai, A., Rokach, L., and Elovici, Y. (2011). Honeygen: An automated honeytokens generator. pages 131–136.

Cvrcek, D. (2014). Hardware scrambling-no more password leaks.

Dionysiou, A., Vassiliades, V., and Athanasopoulos, E. (2021). Honeygen: Generating honeywords using representation learning. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 265–279, New York, NY, USA. Association for Computing Machinery.

Everspaugh, A., Chatterjee, R., Scott, S., Juels, A., and Ristenpart, T. (2015). The pythia prf service. *IACR Cryptology ePrint Archive*, 2015:644.

Juels, A. and Rivest, R. L. (2013). Honeywords: Making password-cracking detectable. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 145–160, Berlin, Germany. ACM.

K. Naik, V. Bhosale, V. D. S. (2016). Generating honeywords from real passwords with decoy mechanism. *International Journal for Research in Engineering Application & Management (IJREAM)*, 02.

Kontaxis, G., Athanasopoulos, E., Portokalidis, G., and Keromytis, A. D. (2013). Sauth: Protecting user accounts from password database leaks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 187–198, Berlin, Germany. ACM.

Krebs, B. (2019). Facebook stored hundreds of millions of user passwords in plain text for years. https://krebsonsecurity.com/2019/03/facebook-stored-hundreds-of-millions-of-user-passwords-in-plain-text-for-years/.

Mohammadinodoushan, M., Cambou, B., Philabaum, C., Hely, D., and Booher, D. D. (2019). Implementation of password management system using ternary addressable puf generator. In *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 1–8. IEEE.

Morris, R. and Thompson, K. (2002). Password security: A case history. *Communications of the ACM*, 22.

OWASP (2021). Password storage cheat sheet.

Provos, N. and Mazieres, D. (1999). A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, Monterey, California, USA. USENIX Association.

Sahar, F. (2013). Tradeoffs between Usability and Security. *IACSIT International Journal of Engineering and Technology*, 5(4).

Shi, C. and Sun, H. (2021). *HoneyHash: Honeyword Generation Based on Transformed Hashes*, pages 161–173. Springer International Publishing, Virtual.

Shi, Z., Yang, C., and Wu, Q. (2012). Scalable md5 crypt cracker on petascale supercomputer. *Advanced Materials Research*, 532-533:1080–1084.

Thomas, K., Li, F., Zand, A., Barrett, J., Ranieri, J., Invernizzi, L., Markov, Y., Comanescu, O., Eranti, V., Moscicki, A., et al. (2017). Data breaches, phishing, or malware? understanding the risks of stolen credentials. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1421–1434, Dallas, USA. ACM.

Wang, D., Cheng, H., Wang, P., Yan, J., and Huang, X. (2017). A security analysis of honeywords.

Wang, K. C. and Reiter, M. K. (2021). Using amnesia to detect credential database breaches. In *30th USENIX Security Symposium (USENIX Security 21)*.