Examination of a New Defense Mechanism: Honeywords

Ziya Alper Genç[1], Süleyman Kardaş[2],
Mehmet Sabır Kiraz[3]

[1] University of Luxembourg
[2] Batman University
[3] TÜBİTAK BİLGEM

Abstract. Past experiences show us that password breach is still one of the main methods of attackers to obtain personal or sensitive user data. Basically, assuming they have access to list of hashed passwords, they apply guessing attacks, *i.e.,* attempt to guess a password by trying a large number of possibilities. We certainly need to change our way of thinking and use a novel and creative approach in order to protect our passwords. In fact, there are already novel attempts to provide password protection. The Honeywords system of Juels and Rivest is one of them which provides a detection mechanism for password breaches. Roughly speaking, they propose a method for password-based authentication systems where fake passwords, *i.e.,* "honeywords" are added into a password file, in order to detect impersonation. Their solution includes an auxiliary secure server called "honeychecker" which can distinguish a user's real password among her honeywords and immediately sets off an alarm whenever a honeyword is used. However, they also pointed out that their system needs to be improved in various ways by highlighting some open problems. In this paper, after revisiting the security of their proposal, we specifically focus on and aim to solve a highlighted open problem, *i.e.,* active attacks where the adversary modifies the code running on either the login server or the honeychecker.

Keywords: passwords, cracking, honeywords, code modification

1 Introduction

Password based authentication is a widely used technique throughout the Internet due to its simplicity and efficiency. However, this mechanism brings the potential risk of user credentials' being stolen in a server compromise event. In fact, there have been many incidents that confirm the significance of this threat where an adversary were able to obtain the database, which contains the usernames and the corresponding password hashes [1,5,7]. Once the breach occurs, it is at the mercy of authentication authority to disclose the details of the incident unless the adversary publishes the credentials.

Authentication systems employ cryptographic measures to protect the user credentials, however, many users have tendency to choose weak passwords *i.e.,* common words that can be easily guessed by a dictionary attack [3, 4]. Due to the advancements in GPU technology, the hash value of a common password (*i.e.,* a word in a dictionary) can be cracked efficiently. These kinds of attacks may allow user credentials to be obtained by an adversary. Existing servers are capable of blocking any illegitimate login attempt when the authentication servers employ additional security mechanisms (*e.g.,* SMS that is used for 2 factor authentication) [10, 11]. Even though such multi-factor authentication solutions improve the security against any illegitimate login attempt, these solutions do not provide any detection of password breaches.

In order to detect whether the password file has been stolen or not, Juels and Rivest [6] proposed the use of "honeywords", that is, a set of fake passwords that are mixed with the user's real password and the hash values of these passwords (real password and honeywords) are stored in the password file. Suppose that this file is compromised and all hash values in the file are cracked, the adversary still does not know which one is the real password. Note that the user or the adversary sends LS identity and password in order to request login. Then, LS checks whether a submitted password is among a user's honeywords but even when this check succeeds, LS needs to consult another secure component, HC, to know whether the index of the retrieved honeyword is that corresponding to the user's real password. HC alerts the administrator otherwise, since having observed an honeyword signals that the password file might have been compromised.

Our contributions. In this paper, we first examine the Honeywords system of Juels and Rivest [6] and propose a practical improvement to solve a highlighted open problem. We enhanced the security of the Honeywords protocol against active code modification attacks where the adversary is assumed to modify the code running on either the login server or the honeychecker.

Roadmap. The outline of this paper is as follows. Section 2 describes the brute-force and dictionary attacks on passwords and their connection with the Honeywords system. Section 3 gives a detailed overview of Honeywords scheme. In Section 4, we present our improvements against active adversaries. Security analysis of our enhanced model is given in Section 5. Finally, Section 6 concludes the paper.

2 Offline Brute-force and Dictionary Attacks

Brute-force password cracking is one of the most popular attack types related to passwords. In a typical scenario, first, the adversary steals the password hash file. Next, a set which contains only the presumed characters that appear in a password is created. Then the adversary creates a combination of characters from this set, computes its hash and compares the hash value with the password hash. This process continues until a match is found [2].

There exist several techniques which increase the success rate of attackers while performing a brute-force attack. As an example, Weir *et al.* [12] developed a state of the art password cracking algorithm which uses probabilistic, context-free grammars. Kelley *et al.* [8] showed that using Weir's attack, one billion guess is enough to crack % 40.3 of the passwords that comply with the "basic8" policy, *i.e.,* a password must have at least 8 characters. In the meantime, parallel processing capabilities of GPUs have been increased dramatically. For example, using `hashcat`[4], an open source password recovery software, cracking speed of hashes has reached 101.3×10^9 passwords per second for SHA1 on a single high computing cluster[5] which is commercially available [9].

Brute-force attacks sometimes can be applied efficiently depending on the password policy of the system. Assume that a user creates a password consisting of letters in the English alphabet which complies with the basic8 policy. The required time to span the password space of this password can be computed as follows:

$$\mathsf{Time} = \frac{\mathsf{Password\ space\ length}}{\mathsf{Cracking\ speed}}$$

Considering the worst case and applying the above formula, we find that using the previously mentioned cluster, an adversary can crack the SHA-1 hash of that password in

$$\frac{(26)^8 \mathsf{\ passwords}}{101.3 \times 10^9 \mathsf{\ passwords\ per\ second}} \approx 2.06 \mathsf{\ seconds}.$$

Offline dictionary attack is similar to brute-force, with one difference. In this type of attack, an adversary computes the hash of words (possibly with salt) from a list that consists of strings which are typically derived from a dictionary. The adversary compares these hashes with the password hashes. The intention is to try dictionary words (which are more likely to be a user's password) rather than sequences of random characters. Most users unfortunately do not choose strong passwords for the sake of easy memorization. On the contrary, they choose weak passwords that are simple concatenations of dictionary words, common names, birthdays, city/street names, or easily guessable phrases.

3 Review of Honeywords System

Juels and Rivest proposed a novel authentication scheme called Honeywords system [6]. The central idea behind the Honeywords system is to change the structure of the password storage in such a way that each user is associated with a password and a set of fake passwords. The fake passwords are called honeywords. The union of all honeywords and the password is called sweetwords. As soon as a honeyword is submitted during the login process, it is automatically detected that the password database has been stolen. Hence, unlike conventional systems, honeywords based solutions can easily detect password database breaches.

---

[4] https://hashcat.net/hashcat/
[5] https://gist.github.com/epixoip, Retrieved on June 22, 2017.

User ID	Username	Hashes
1	u_1	$H\left(sw_{1,1}\right), \cdots, H\left(sw_{1,n}\right)$
2	u_2	$H\left(sw_{2,1}\right), \cdots, H\left(sw_{2,n}\right)$
$\cdots$	$\cdots$	$\cdots$
m	u_m	$H\left(sw_{m,1}\right), \cdots, H\left(sw_{m,n}\right)$

Fig. 1. Credentials database of a LS in the Honeywords system

User ID	Password Index
1	c_1
2	c_2
$\cdots$	$\cdots$
m	c_m

Fig. 2. Data stored on a HC

The Honeywords system works as follows. As in the many conventional systems, users choose a *username* and a *password* during the registration phase. Next, the Login Server (LS) generates honeywords for the password and creates a record in credentials database. In each record, the ordering of the sweetwords is randomly chosen by the LS. Furthermore, LS sends the corresponding user ID and the index of the real password to Honeychecker (HC), which is an auxiliary server designed to store the index of the password. Let u_i and $H()$ denote the username of user i and the hash function used in the system, respectively. $H(sw_{i,j})$ denotes the hash of jth sweetword of user i. A typical example of credentials table is demonstrated in Fig. 1.

HC stores the user IDs and the index of the passwords among the honeywords. Neither username nor password itself is submitted to HC during the authentication. Moreover, HC is designed as an *hardened* server which can only be accessed by LS. A typical structure of the data stored in HC is demonstrated in Fig. 2.

Note that HC accepts only two types of messages: Check and Set.

- Check(i, j) means to confirm whether $j = c_i$. If $j = c_i$, HC returns True, otherwise it returns False and triggers the alarm.
- Set(i, j) means to set $c_i = j$.

During the authentication phase, user submits her username and password. LS tries to find the corresponding record for that username in the credentials database. If a record exists, LS computes the hash of submitted password and tries to find a match in the hashes of sweetwords. If there is no match, then the submitted password is wrong and the access is denied. If there is a match, LS sends the corresponding user ID and the matching index to HC. Next, HC finds the record which corresponds to the user ID and compares the received index value with the one stored in its database. If the result is true, then the access is granted. Otherwise, the HC returns false and follows the system policy, *e.g.,* creates an alert and notifies the administrators. Authentication phase of the Honeywords system is depicted in Fig. 3.

The Honeywords system is originally designed with the assumption that the adversary can steal the hashed passwords and can invert the hashes to obtain the passwords. Also, it is assumed that the adversary cannot compromise both LS and HC in the same time period. Under this assumption, the Honeywords system protect passwords against brute-force and dictionary attacks described in Section 2. The Honeywords system aims at detecting password database breaches
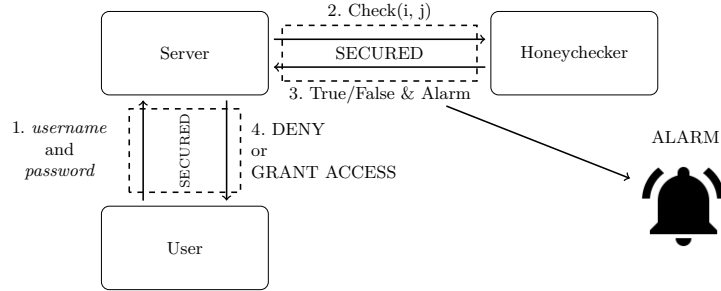
Fig. 3. Login scheme of a system using honeywords

and helps deterring only offline dictionary attacks where it is assumed that the adversary has stolen the password hashes and left the system. As also pointed out by Juels and Rivest, there are multiple open problems to solve in order to withstand active attacks.

4 Our Proposed Solutions

In this section, we focus on the following open problem which is highlighted in the original paper [6]:

> How can a honeyword system best be designed to withstand active attacks, e.g., malicious messages issued by a compromised computer system or code modification of the computer system (or the honeychecker)?

For this scenario, we deal with the active attacks in which the adversary makes code modifications on the Honeywords system where the adversary basically executes a malicious code on LS and set the index of the password to a new value that corresponds to a recovered honeyword. In the lights of these circumstances, the Honeywords system needs to be improved in order to withstand these advanced types of attacks.

4.1 Defending Against Malicious Code Modifications

An adversary may gain privileges to modify the running code on components of the Honeywords system. We only consider the cases where an adversary corrupts the component of LS that performs Set and Check commands. In that case, the parameters of these commands can be altered by the adversary. Similarly, HC can also be corrupted and send maliciously modified responses to LS. To mitigate these attacks, a reliable auditing mechanism is needed to check and verify the correctness of LS and HC. We classify the attack scenarios into two cases as follows.

Preventing malicious modifications on Set and Check Commands: A malicious adversary may target on modifying Set and Check commands that are run on the HC in order to gain more advantages for her attacks. He could call these command from the corrupted LS. In this context, in order to detect a malicious activity of a corrupted or legal LS, HC can verify whether the user requests a password change. If confirmed, HC will process Set command. Otherwise, the request would not be valid which detects the malicious activity by LS. In order to validate the origin of the Set request, HC may use some helper data. Following the design principles of the original protocol and keep the amount of data at HC at a minimum level it is possible to ask a security question or send a validation code to the mobile phone of the user. We follow the latter approach since it is in wide spread use and a practical way of adding another factor of authentication. The system roughly works as follows: during the registration, LS asks the user to enter the registration information including mobile number. The mobile number will be stored by HC. In order to accomplish this task, we overload the Set function as follows:

- Set(i, j)
- Set(i, j, phn)

where phn denotes the phone number. Note that Set(i, j) function is the same as the Set function in the original Honeywords system. Set(i, j, phn) is invoked whenever a user registers to the system. Set(i, j) is invoked whenever a user changes her password. While the login procedure depicted in Fig. 3 did not change, the sign up of our enhanced Honeywords system is depicted in Fig. 4.

If the password change request is received, LS will generate new honeywords and randomly permute the sweetwords. Next, LS will send Set(i, c_i) to HC. HC generates a one time pad *nonce* and using *phn* sends it to user via SMS. The user submits *nonce* to LS which forwards it to HC for validation. The password change scheme is depicted in Fig. 5.

Preventing malicious code modification on HC: In this scenario, an adversary can modify HC and can send illegitimate responses to LS. In order to
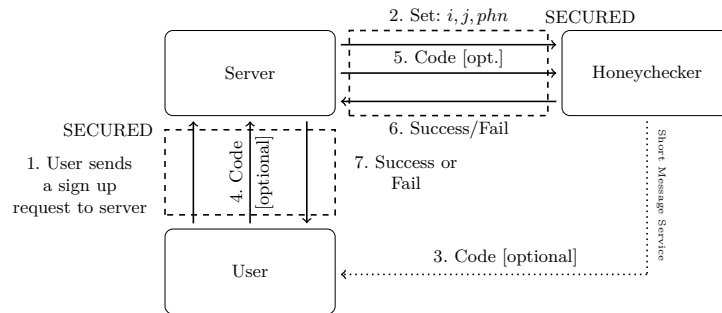

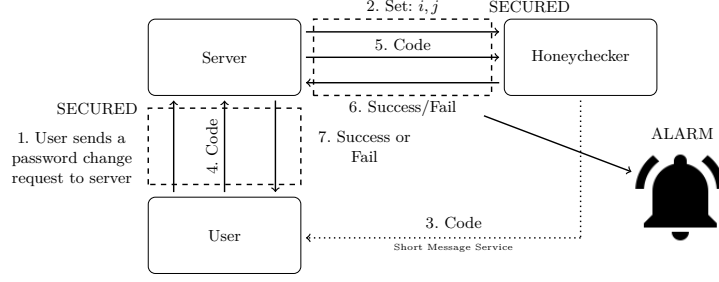**Fig. 4.** Sign up scheme to mitigate against malicious code modification of LS.

Fig. 5. Password Change scheme to mitigate against malicious code modification of HC.

verify the correctness of HC, we propose an efficient probabilistic method which audits HC periodically. More concretely, there will be built-in user accounts in LS whose passwords will be known to LS. Since LS knows the result of these Check messages by itself, it can easily verify them with the ones coming from HC. More formally, let b be the user ID of a built-in account and c_b be the index of the password which is known to LS. A scheduled service at LS takes the following actions to test the correctness of HC.

Algorithm 1 Test Correctness of Honeychecker

1: **function** IsCorrupted(bID, c_{bID})   ▷ bID: built-in user ID, c_{bID}: built-in index
2:     $response \leftarrow$ **false**
3:     $isCorrupt \leftarrow$ **true**
4:     $response \leftarrow$ Check(bID, c_{bID})
5:     **if** $response =$ **true then**:
6:         $isCorrupt =$ **false**
7:     **return** $isCorrupt$

This test will be repeated periodically for all built-in accounts to increase the probability of detection. For simplicity, let the probability of receiving a legitimate response from a compromised HC be fixed (p). After t number of tests, the probability of detecting a malicious activity P is computed as

$$P = 1 - \prod_{i=1}^{t} p_i$$

where t denotes the number of tests and p_i denotes the probability of receiving an legitimate response from the malicious adversary and $\forall i, p_i = p$.

If HC responds uniformly at random, then the equality yields

$$P = 1 - \prod_{i}^{t} p = 1 - \prod_{i}^{t} \frac{1}{n} = 1 - \frac{1}{n^t}$$

where n denotes the number of honeywords per user. Given $n = 20$ as suggested by [6], only two test is enough to achieve the correctness check with probability $P > 0.997$.

5 Security Analysis

In this section, we analyze the security of our enhanced system. We follow the case by case approach to perform the security analysis. We begin with the case that the adversary has compromised the LS.

Theorem 1. *Under the assumption that the honeywords are indistinguishable from passwords, if the LS is compromised, then the enhanced system depicted in Fig. 4 detects illegitimate login attempts and the incident of password breaches with probability $1 - \frac{1}{n}$ where n is the number of sweetwords.*

Proof. Assume that the adversary compromised the LS and obtained the user IDs and sweetwords, *i.e.,* $sw_1, sw_2, \ldots, sw_n$. Since the index of the password, c_i is stored securely in the HC, the adversary will try to guess the password among honeywords. There are $n-1$ honeywords per user and the probability of guessing the password is $\frac{1}{n}$. In other words, the adversary will fail and the system will detect the credentials breach with probability $1 - \frac{1}{n}$. $\qquad\square$

Next we continue with the case that the adversary steals the information from HC.

Theorem 2. *Our enhanced system does not disclose any information about the passwords if an adversary steals only HC database.*

Proof. Assume that an adversary steals HC database. According to the design principles of the Honeywords system LS does not send any information about passwords to HC. Therefore, the adversary cannot not obtain any information about the passwords as only i, c_i and *phn* records are stored in HC. $\qquad\square$

Next, we prove the security of our enhanced system in the case that the adversary modifies the code running on LS. We assume that any abnormal deviation from the protocol would be noticed by the system administrators (*e.g.,* by auditing network logs), and therefore, adversary would have to restrict itself to modify existing functions for attacking the system. Thus, we consider abuse of only existing functions of LS, *i.e.,* Set and Check commands.

Theorem 3. *Our enhanced system does not disclose the passwords with probability $1 - \frac{1}{n}$ where n is the number of sweetwords if an adversary maliciously modifies the running code on LS. Also, the probability of unauthorized password change is negligible.*

Proof. Assume that an adversary maliciously modifies the running code on LS. A corrupted LS would send only Set or Check messages to HC (other abnormal behaviors would be detected by the administrators). However, a malicious Check message would be detected and the alarm would be set off by HC with probability $1 - \frac{1}{k}$ where k is the number of honeywords. Similarly, in the case that adversary sends malicious Set messages, HC will ask a validation code which is sent to the user's mobile phone. However, the probability of sending a valid combination without possessing the mobile phone is negligible. Thus, the adversary will also fail to send a malicious Set request. □

Finally, we analyze the case that the adversary modifies the code running on HC.

Theorem 4. *Our enhanced system does not disclose any information about the passwords if an adversary maliciously modifies the running code on HC.*

Proof. Consider that the adversary modifies the code running on HC. The TestHoneychecker routine will check the correctness of the Check messages and detect malicious responses with probability $P = 1 - \prod_{i=1}^{t} p_i$, where t denotes the number of tests and p_i denotes the probability of receiving an illegitimate response. The HC does not contain any useful information that would help to adversary developing a meaningful attack strategy. Hence, if the HC maliciously responds to the Check messages, LS will detect with probability P. □

Hence, our scheme has all the security properties of the the primitive system designed in [6], plus it is more robust to code modification attacks.

6 Conclusion

Juels and Rivest [6] propose an interesting defense mechanism under a common attack scenario where an adversary steals the file of password hashes and inverts most or many of the hashes. The Honeywords system provides a powerful defense against this attack. Namely, even if the adversary recovers all of the hashes in the password file, he cannot try to login to the system without a high risk of being detected. On the other hand, the original Honeywords system is not a complete solution for the password management problem. The scenarios in which an adversary modifies running code on LS or HC is left as open problem. In this work, we review the original Honeywords system and focused on solving that problem. Namely, we enhanced the Honeywords system through adding additional security checks. Our additions are inexpensive and practical, and can be easily integrated into the primitive scheme. Moreover, we discussed the security of our enhanced protocol and showed that it is robust against code modification attacks.

Acknowledgments

References

1. Burgess, M.: How to check if your linkedin account was hacked (May 2016), http://www.wired.co.uk/article/linkedin-data-breach-find-out-included
2. Conklin, A., Dietrich, G., Walz, D.: Password-based authentication: A system perspective. In: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 7 - Volume 7. pp. 70170.2–. HICSS '04, IEEE Computer Society, Washington, DC, USA (2004)
3. Florencio, D., Herley, C.: A large-scale study of web password habits. In: Proceedings of the 16th international conference on the World Wide Web. Association for Computing Machinery Inc. (2007)
4. Furnell, S., Dowland, P., Illingworth, H., Reynolds, P.: Authentication and supervision: A survey of user attitudes. Computers & Security (2000)
5. Gallagher, S.: Yahoo admits it's been hacked again, and 1 billion accounts were exposed (Dec 2016), https://arstechnica.com/security/2016/12/yahoo-reveals-1-billion-more-accounts-exposed-and-some-code-may-have-been-stolen/
6. Juels, A., Rivest, R.L.: Honeywords: Making password-cracking detectable. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. pp. 145–160. CCS '13, ACM (2013)
7. Keane, J.: Security researcher dumps 427 million hacked myspace passwords online (Jul 2016), https://www.digitaltrends.com/social-media/myspace-hack-password-dump/
8. Kelley, P.G., Komanduri, S., Mazurek, M.L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L.F., Lopez, J.: Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In: IEEE Symposium on Security and Privacy. pp. 523–537 (2012)
9. Sagitta: Brutalis - GPU Compute Nodes, https://sagitta.pw/hardware/gpu-compute-nodes/brutalis/
10. Wang, D., Wang, P.: Two birds with one stone: Two-factor authentication with security beyond conventional bound. IEEE Transactions on Dependable and Secure Computing PP(99), 1–1 (2017)
11. Wang, D., Gu, Q., Cheng, H., Wang, P.: The request for better measurement: A comparative evaluation of two-factor authentication schemes. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. pp. 475–486. ASIA CCS '16, ACM, New York, NY, USA (2016)
12. Weir, M., Aggarwal, S., Medeiros, B.d., Glodek, B.: Password cracking using probabilistic context-free grammars. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy. pp. 391–405. SP '09, IEEE Computer Society, Washington, DC, USA (2009)