ResearchCoders' For Programmers Series Honeywords: Making Password-Cracking Detectable V.1

Mohammed Q. Hussain
mqh@reseachcoders.dev

December 2018

1 Introduction

The work in [1] focuses on the scenario where an attacker of a specific system steals a file (or data on database) that contains the hashed passwords of system's user. The authors present the idea of *honeywords* to make such situation detectable. The paper suggests that instead of associating just one password to users accounts, more than one password is associated to all users accounts of the system, just one of these passwords is correct, others are just honeywords. The presented mechanism in this work lies under the umbrella of deception defense mechanisms.

1.1 Deception Defense

In computer security, *deception defense* is a way of protecting systems from attacker but instead of traditional way which aims to prevent the attacks, in deception defence the attacker will be *deceived.* We all know that perfect security of any given system is unattainable. Therefore, multiple ways other than completely-preventing-the-attacks has been presented in the scientific literature. Deception defense one of those ways, the other one is Moving Target Defense.

The work in [1] is one of deception defense techniques. Another example of deception attacks is *honeypot*, which is a normal machine (or virtual machine) in a network which appears to provide some services, for example serving a website, but in reality, the honeypot tries to appeal the attackers to intrude it. Once the attacker take the bait and attack it, the honeypot is going to log detailed information about the attacker in order to be analyzed later by the system administrator. Of course, the honeypot has no valuable data for the attacker, so by attacking it, the attacker just wasted her time and has been exposed for the system administrator.

1.2 Hashed Passwords

It is a common practice in current system to store the passwords of users as hashed passwords instead of plaintext. When we say a hash password we mean that the original password has been passed to some hash function, and the returned result of this function is what we call *hashed password* and it is what will be stored in the system, the hashed password of course will be different than the original password. To authenticate a user when only hashed password is stored in the system, the user first has to enter her password into the system, then the system is going to use the hash function on the password the user just entered, if the hash of the entered password matches the hashed password which is stored in the system, then the entered password is correct, otherwise, the entered password isn't correct.

A hash function takes an input, and converts this input into something else. In this case, the input is the original password, and the output is the hashed password. In the realm of passwords, a special kind of hash functions are used, they are called *cryptographic hash functions*. These cryptographic hash functions has an important property which makes them useful for hashing the passwords. They are *one-way functions*, which means that computing the hashed password (output) from the original password (input) is easy, but if we only have the hashed password and we would like to compute the original password from it, that will be hard. A popular example of *cryptographic hash functions* is MD5 which can be used easily in multiple programming languages such as PHP [1] and Python [2].

As an example, let's say that we have a PHP web application that stores users' passwords as hashed passwords by using MD5. Let's say one of user's password is: 123. If we use MD5 to hash this password the output [3] will be 202cb962ac59075b964b07152d234b70 which will be stored in the database. If some attacker manages to steal the list of users and their passwords they will get this type of passwords. To invert these hashed passwords to the original ones, multiple time/space consuming techniques are available, such as brute-force attacks and rainbow tables.

1.2.1 Salts

As an enhancement for hashed passwords technique, sometimes *salts* are used. Salt is a randomly generated data which is associated to a password. This password is concatenated with its salt and the whole string is passed as an input to

[1] http://php.net/manual/en/function.md5.php

[2] https://docs.python.org/2/library/md5.html

[3] The output of hash functions in general is called *digest*.

the hash function, the salt will be stored in the system as well and the users know nothing about them. For example, let the password of the user is 123 and its salt is 539, then the input to the hash function (MD5 for example) will be 123539, which gives the following hashed password: 3e4281ab57e6c6e3395d90e348f926a7. Salts are useful against rainbow tables, brute-force and dictionary attacks, they make them harder and more time consuming.

2 Honeywords Technique

As mentioned before, this paper [1] assumes that some attacker has obtained a file the contains the usernames, their hashed passwords and salts. In normal situations, where honeywords technique is not employed, in each entry of the file there will be one unique username and the associated hashed password. When honeywords are used, for each username there will be multiple entries, each with different hashed password, only one of them is the real password and the others are honeywords. For example, let's assume there is a username called u_1 with five different hashed passwords associated with it, in some way, the attacker managed to crack one of those five hashed passwords, but this cracked password, for the bad luck of the attacker, is a honeyword and not the real password of u_1. When the attacker tries to login by using this honeyword the system is going to detect this action and send an alert for the system administrator that there is a possible passwords disclosure.

2.1 Honeychecker

From the previous discussion until now, there is an obvious question that pop up, how they system is going to differentiate between the types of stored passwords of a specific user? On other words, the user u_1 which mentioned previously has five hashed passwords, how the system that employs honeywords technique can tell which of these passwords is the real one and which are honeywords?

The naive solution for this problem is to store this information in the same place, for example, if our system uses database to maintain the list of users and their passwords in a database table called $users$, a new column called *honeyword* can be defined in the $users$ table, and if its value is Y then the hashed password of corresponding row is a honeyword, otherwise it's not. The problem of this solution is obvious, since we assumed that the attacker is able to compromise this server and obtain the list of username and hashed passwords then it is logical to assume that the attacker is also able to obtain the data of *honeyword* column, that makes the whole honeyword technique useless. Therefore, we need a better and more secure solution that doesn't store more secrets on the same server. For the preceding discussion, we call the server that contains the main web application S_1.

To solve this problem, *Honeychecker* is presented in [1] which is another secure server that has a secure communication channel with S_1 and stores the information about honeywords of each specific account. So, in this way we will have two servers, the first one is S_1 which has the main application (e.g. web application) and stores the hash passwords (real passwords and honeywords) of the users. The second server is the Honeychecker. With these two servers that have different roles to secure the system, we have got what is called a *distributed security system*. When a user tries to login, S_1 is going to consult Honeychecker which is capable of detecting any anomaly (e.g. using honeyword to login), when such anomaly is detected, and alarm can be sent to the system administrator. Honeychecker has been designed in a way that when the it is compromised by itself, it doesn't cause users' account to be compromised.

2.1.1 The List of Sweetwords

For each user, the Honeychecker maintains a list of *sweetwords*, To make our discussion easier, let's give this list a name, say W. A sweetword, which is a member of the list W, can be a honeyword (a chaff, which is a fake password) or a *sugarword* (which is the original password). The size of the list W can be a system-wide parameter k for each user, the authors recommend the size 20 for each user's list of sweetwords (W), more that 20 can also be used of course. Beside the list W for each user, the Honeychecker also stores the index of the sugarword (the original password) in W. The following pseudocode shows an example of how the the information related to the user u_1 are stored in Honeychecker where $k = 5$:

```
// sweetWordsOfU1 Also known as "List W of U1"

sweetWordsOfU1[ 0 ] = "5737034557ef5b8c02c0e46513b98f90";
sweetWordsOfU1[ 1 ] = "73e4fad7c9a93fbec445b96159ee5f78";
sweetWordsOfU1[ 2 ] = "a2596ad2908727814da1e8732a848589";
sweetWordsOfU1[ 3 ] = "202cb962ac59075b964b07152d234b70";
sweetWordsOfU1[ 4 ] = "caf1a3dfb505ffed0d024130f58c5cfa";

indexOfU1Sugarword = 3;
```

The Honeychecker is also responsible of generating the honeywords that should be in a specific user's sweetwords list W. The generation of honeywords should be random and generated honeyword can be a *tough nut* which is a very strong password that its hash cannot be cracked by the attacker.

2.1.2 What Honeychecker Can Do If a Honeyword is Used?

The authors suggested many multiple actions to be done when someone uses a honeyword to login, (1) an alert can be sent to the system administrator that shows a possible passwords disclosure, (2) letting login proceed to a *honeypot* that gathers forensic information about the logged user, (3) shutting down the account of the user until the user reset his password, (4) shutting down all the accounts and force the users to change their passwords. However, the programmer of the system is free to choose any of these options or to define a new one.

2.2 Honeyword Generation

The proposed methods of generating honeywords are divided to two groups: (1) those methods that don't require modifications on change password user interface (*legacy-UI*) and (2) the methods that require some modifications (*modified-UI*).

Chaffing is the process of generating honeywords, placing them in a random order beside the sugarword. This process should generate honeywords in a way that makes the honeywords hard to be distinguished from the sugarword (the original password) by the attacker.

2.2.1 Legacy-UI Methods

In legacy user interface of change password, the system request from the user to enter her new password.

Chaffing by Tweaking In this method, in order to generate a new honeyword, the characters in some selected positions of the sugarword (original password) are replaced with randomly-chosen character of the same type, for example, digits are replaced with digits, letters are replaced with letters and special characters are replace with special characters. The number of positions that should be changed can be a system-wide parameter, this parameter is called t.

A derived method of chaffing by tweaking method is called *chaffing by tail tweaking* where always the last t positions of sugarword are selected and replaced. For example, given that the system changes three positions in the sugarword (which means $t = 3$), and the sugarword itself is $BG + 7y45$, then one of possible honeyword to be generated by using chaffing by tail tweaking is $BG + 7q03$.

Another derived method is *chaffing by tweaking digits*, which selects the lat t digits of the sugarword and replaces them. For example, if the sugarword is $42 * flavors$ and $t = 2$ then one possible honeyword to be generated is $18 * flavors$.

Chaffing with a Password Model This method uses a list of thousands or millions of real passwords to generate honeywords from, though, the authors note that using a publicly available passwords list is not a good idea since the attacker may have an access to it and can use it to differentiate between real passwords and honeywords.

Another method under this umbrella is *modelling syntax* which uses same way as compiler's lexical analyzer uses. The sugarword will be analyzed to extract *tokens* out of it. By using these tokens the components of the sugarword are replaced with other components. For example, let the sugarword is $mice3blind$, tokens extraction is going to give us the following result about this password $W_4|D_1|W_5$, which means that this password contains a word of 4 letters (W_4) then there is a digit (D_1) and finally a word of 5 letters (W_5). The replacements of both W_4 and W_5 can be from a dictionary, W_4 will be replaced by another word with 4 letters, for example "gold", and W_5 will be replaced with a word with 5 letters, for example "rings", finally, D_1 can be replaced with any other digit. So a possible generated honeyword by using this method is $gold5rings$.

Chaffing with Tough Nuts As mentioned before, *tough nuts* are hashes that really hard to crack by the attacker, in this way, multiple (but not all the list of honeywords) tough nuts are generated and used in the list W of the user, these tough nuts can be generated easily by for example hashing a really long and random text. When the attacker cannot crack some passwords in the list, he maybe pause the attack and don't try to use the cracked passwords.

2.2.2 Modified-UI Method

In this method, the authors propose to change the interface of changing password by showing for the user that their entered password will be appended by some randomly generated digits. Say the user entered the password "RedEye2" and the system showed for her that this password will be appended with the digits "413", then the real password of the user will be "RedEye2413". After that, the aforementioned method *chaffing by tail tweaking* is used.

3 Epilogue

I tried my best in this document to give a general picture of the work presented in [1] in an easy way for programmers, yet, I don't believe that this document is enough for an enthusiastic programmer that would like to read all the details. Therefore, I encourage you to take a look at the original work to read about other details not presented in this document.

References

[1] Ari Juels and Ronald L Rivest. Honeywords: Making password-cracking detectable. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 145–160. ACM, 2013.