International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

Web Application:(with) HoneyWords and HoneyEncryption

Harish Reddy B[1], Beatrice Ssowmiya J[2]

M.Tech, Department of Information Technology, SRM University, Chennai, India

Abstract: ***It has become much easier for an attacker to steal hash passwords and enter into the account through legitimate user by cracking the hash passwords. So, for each user account, the legitimate password is stored with several honeywords in order to sense impersonation. If honeywords are selected properly, an adversary who steals a file of hashed passwords cannot be sure if it is the real password or a honeyword for any account. Moreover, entering with a honeyword to login will trigger an alarm notifying the administrator about a password file breach. Here I am implementing Honey Encryption for the protection of data stored by the user in a web application, that produces a cipher text, which, when decrypted with an incorrect key as guessed by the attacker, presents a plausible-looking yet incorrect plaintext password or encryption key.***

Keywords: Honeywords, Honey Encryption, Authentication, Security, Password

1.Introduction

For every web application, in the authentication process, password became the most important asset to login. But users choose weak passwords that can be predicted by the attacker using brute force, dictionary, rainbow table attacks etc…So it has become much easier to crack a password hash with the advancements in the graphical processing unit (GPU) technology. An adversary can recover a user's password using brute-force attack on password hash. Once the password has been recovered no server can detect any illegitimate user authentication.

So Honeywords are a defense against stolen password files. Specifically, they are bogus passwords placed in the password file of an authentication server to deceive attackers. Honeywords resemble ordinary, user-selected passwords. It's hard therefore for an attacker that steals a honeyword-laced password file to distinguish between honeywords and true user passwords.

An auxiliary service called a honeychecker checks whether a password submitted by a user on login is her true password or a honeyword. The password system itself stores a given user's password randomly along with honeywords. Together, the password and honeywords are called sweetwords. The user's password is placed within a sweetword list at a random position *c*, which is stored by the honeychecker. The password system itself doesn't know which sweetword in the list is actually the password. When here's an attempt to log into the user's account with a sweetword, the password system sends the position *j* of this sweetword in the stored list to the honeychecker. The honeychecker verifies that *j=c*, i.e., the true password has been submitted; otherwise it raises an alarm, as a honeyword has been submitted. If an incorrect password is submitted that is also not a sweetword, the system uses its ordinary policy for such cases.

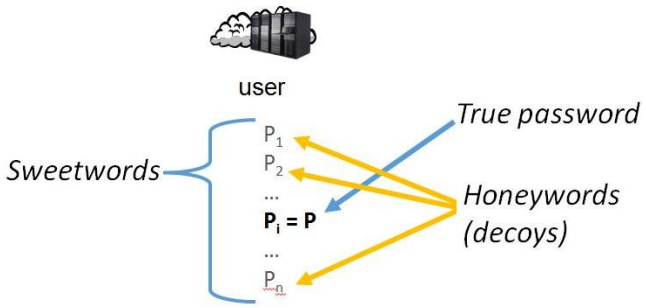

Figure: Sweetwords

A new encryption method is being developed that could frustrate hackers by giving them fake data while making it appear real. Honey Encryption turns every incorrect password guess made by a hacker into a confusing dead-end. When an application or user enters and sends a password key to access an encrypted database or file, as long as the password is correct, the data is decrypted and accessible in its original, and readable, format. If the password key is incorrect the data will continue to be unreadable and encrypted.

Hackers who steal databases of user logins and passwords only have to guess a single correct password in order to get access to the data. The way they know they have the correct password is when the database or file becomes readable. To speed up the process, hackers have access to sophisticated software that can send thousands of passwords each minute to applications in an attempt to decrypt the data. Using higher speed, multi-core processors also shortens the time it can take to break encryption.

2. Honeywords

In this section, I first briefly summarize the honeyword password model proposed by the Juels and Rivest.

Honeywords – enables detection of theft, prevents impersonation.

I propose salting a web-site's password database with lots of false passwords called "honeywords". Passwords in

password databases are typically hashed or scrambled to protect their secrecy. An adversary who steals a file of hashed passwords and inverts the hash function cannot tell if he has found the password or a honeyword.

A password database salted with honeywords would be plugged into a server dedicated exclusively to distinguishing between valid passwords and honeywords. When it detects a honeyword being used to log into an account, it will alert a site administrator of the event, who can lock the account down. Using honeywords won't prevent hackers from breaching your website and stealing your passwords, but it will alert the operators of the website that a breach may have occurred.

While the passwords are basically stored in a hash form like,

1. *P* = Alice's password
2. System stores mapping "Alice" ➔*h (P)* in database, for a suitable hash function *h*.
3. When someone (perhaps Alice) tries to log in as Alice, system computes *h (P')* of submitted password *P'* and compares it to *h (P)*. If equal, login is allowed.
4. Hash function *h* should be easy to compute, hard to invert. Such "one-wayness" makes a stolen hash not so useful to adversary.

To defeat attacks like precomputation attack, a per-user "salt" value *s* is used: system stores mapping "Alice"➔(s,h(s,P)). Hash h(s,P') computed for submitted password *P'* and compared. Hashing with salting forces adversary who steals hashes and salts to find passwords by brute-force offline search: adversary repeatedly guesses *P'* until a *P'* is found such that h(s,P') = h(s,P).

Attacker game is like compromising system ephemerally, steals password hashes. Attacker cracks hash, finding *P*. Impersonate user(s) and logs in. Attacker almost always succeeds, and is often undetected. When an attacker gets the password list, he/she recovers many password candidates for each account and she cannot be sure about which word is genuine. Hence, cracked password files can be detected by system administrator if a login attempt is done with a honeyword by the adversary. Here I use the notations and definitions to simplify the description of honeyword scheme.

H()	Cryptographic hash function used to compute hash of passwords
Ui	Username for the ith user.
Pi	Password of ith user
Wi	List of potential passwords for ui
K	Number of elements in Wi
Ci	Index of correct password in list Wi
Gen(k)	Procedure used to generate Wi of length k of sweetwords
Sweetword: Each element of Wi	
Sugarword: Correct password in Wi	
Honeyword: Fake passwords in Wi	

The honeyword mechanism works simply as follows: For each user ui, the sweetword list Wi is generated using the honeyword generation algorithm Gen(k). This procedure takes input k as the number of sweetwords and outputs both the password list Wi = (wi,1, wi,2, . . . , wi,k) and ci, where ci is the index of the correct password (sugarword). The username and hashes of the sweetwords as < ui, (vi,1, vi,2, . . . , vi,k) > tuple is kept in database of the main server, whereas ci is stored in another server called as honeychecker. By diversifying the secret information in the system – storing password hashes in one server and ci in the honeychecker – makes it harder to compromise the system as a whole, i.e. provides a basic form of distributed security. Notice that in traditional password technique < ui , H (pi) > pair is stored for each account, while for this system < ui, Vi > tuple is kept in database, where Vi = (vi,1, vi,2, . . . , vi,k).

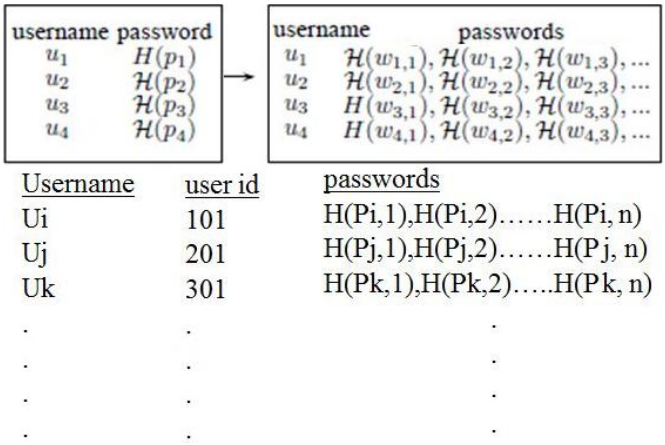

Figure : Password list

The login procedure of the scheme is summarized below:

- User ui enters a password g to login to the system.
- Server firstly checks whether or not H(g) is in list Vi. If not, then login is denied.
- Otherwise system checks to verify if it is a honeyword or the correct password.
- Let v(i, j) = H(g). Then j value is delivered to honeychecker in an authenticated secure communication.
- Honeychecker checks j = ci. If the equality holds, it returns a TRUE value, otherwise it responses FALSE and may raise an alarm depending on security policy of the system.

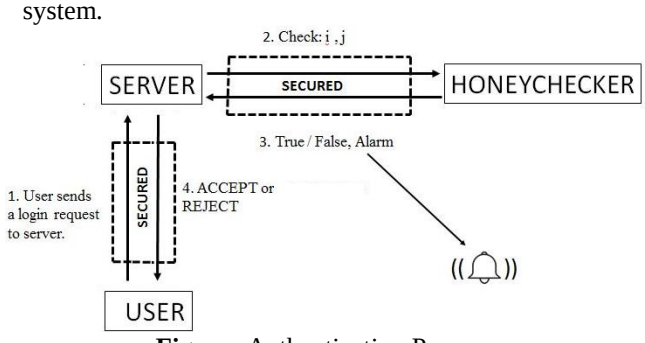

Figure: Authentication Process

In this work, I analyze the honeyword system according to both functionality and the security perspective by implementing web application and try to prevent the legitimate user account from attacker.

3.Honey Encryption

Honey Encryption was created by Ari Juels, former chief scientist of the RSA, and Thomas Ristenpart from the University of Wisconsin. At the time of this writing, Honey Encryption is best-suited for constructions in which encrypted data is derived from passwords. Honey Encryption is a security tool that makes it difficult for an attacker who is carrying out a brute force attack to know if he has correctly guessed a password or encryption key. Typically, an attacker will know he's guessed wrong because the decrypted results will be unintelligible. If Honey Encryption has been used, however, the wrong guess will generate phony results that appear to be genuine. Because each incorrect guess generates a plausible result, it will be difficult for the attacker to know when he has guessed correctly. For an example, if an attacker makes 1000 attempts to guess a credit card number, then for every 1000 attempts he will receive 1000 fake credit card numbers. "Each decryption is going to look plausible. The attacker has no way to distinguish a priori which is correct."

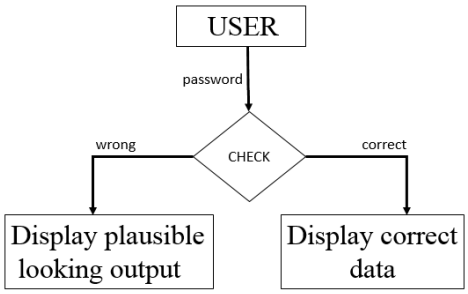

Figure: basic model of Honey Encryption

I think, this technique can be extremely beneficial in credit card and data breaches. Recently, there have been major data breaches at Adobe and Target in which millions of credit card and customer data was compromised. Even though the credit and debit card information was encrypted, the hackers managed to access the data. It is definitely important to use strong encryption standards to protect sensitive data, and adding an extra layer of protection to it will ensure better security and privacy of sensitive customer information. By implementing Honey Encryption, businesses can establish a better defense mechanism against such kind of data breaches.

Encryption process:

Encryption is the most effective way to achieve data security. To read an encrypted file, you must have access to a secret key or password that enables you to decrypt it. Unencrypted data is called plain text; encrypted data is referred to as cipher text.

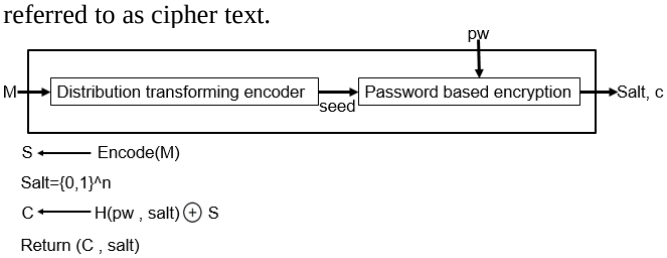

Figure: Honey Encryption (encryption process)

Decryption process:

Decryption is the process of converting cipher text back to plaintext. To encrypt more than a small amount of data, symmetric encryption is used. A symmetric key is used during both the encryption and decryption processes.

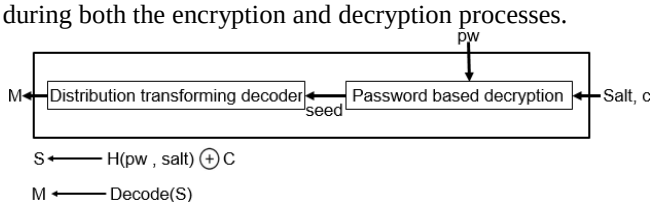

Figure: Honey Encryption (decryption process)

The data which we store inside the web application is stored safely by Honey Encryption. To access the data, user should provide another password which is based on Honey Encryption. Whenever user stores data, the data will be encoded by using Huffman encoder, which will produce seed ({0,1}^n) . The password is combined with salt and hash the total password, the hash and seed will combine with exclusive_or (XOR) to get a cipher text which will be stored in our database with used salt. Decryption process takes place vice versa.

Finally, the design of a conventional cryptographic system makes it easy to know when [guessing a password] is correct or not: the wrong key produces a garbled mess, not recognizable by a piece of raw data. 'With Honey Encryption,' [The hacker] gets something that looks like real context." What the hacker sees "resembles actual data to the point that attackers won't be able to tell what is and isn't real.

4.Solution Architecture

Here, I am creating a web application. When user registers into my account, my system will check the password and generate honeywords with honeyword algorithm and store in the database with user name. When user login, my system will check the password in the database containing real password and honeywords list associated with the username. If it is in the list, then index of the password and user id will be sent to honey checker called server. Honey checker contains only user id and real password, if it matches then user will login and other wise alert message should be sent to email id which user provided at the time of registration. And if the password is not in the database, then login should be denied.

After user login into web application, he creates a profile password for his account. Account contains entries which can be added by the user. Entries are the user's website details like username and password etc... Whenever users want to edit or add entry, he should provide profile password. The data stored by the user will be encrypted by honey encryption. If profile password is correct then correct data should be given to user otherwise plausible looking output.

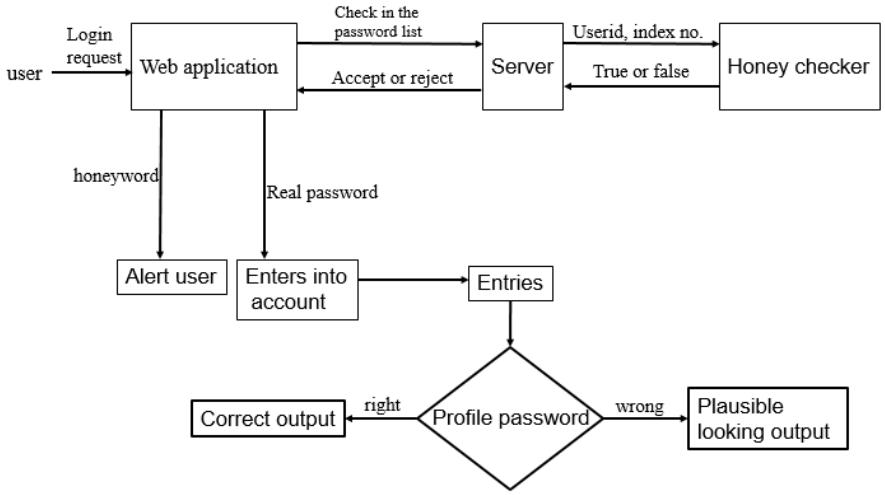

Figure: Proposed architecture

In this work, I suggest some possible improvements in developing a secure web application. The developer should perfectly generate the number of honeywords of a user and the user should give the password which is impossible to guess by the attacker which will help in active or passive attacks problem. I hope my contribution will improve in designing more secure web application with the help of Honeywords and Honey Encryption.

References

[1] A. Juels and R. L. Rivest. Honeywords: Making password cracking detectable.
[2] Ari Juels, Thomas Ristenpart. Honey Encryption: Security beyond the Brute-Force Bound
[3] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas,L. Bauer, N. Christin, L. F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In IEEE Symposium on Security and Privacy, pages 523–537, 2012.
[4] J. Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In IEEE Sump. Security and Privacy, 2012.
[5] A. Conklin, G. Dietrich, and D. Walz. 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, HICSS '04, pages 70170.2–, Washington, DC, USA, 2004. IEEE Computer Society.
[6] D. Elser and M. Pekrul. Inside the password-stealing business: the who and how of identity theft, 2009.
[7] A. Shamir,―How to share a secret,‖ Commun.ACM, vol. 22, no. 11, pp.612–613, 1979.
[8] B.M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside Attackers Using Decoy Documents, pages 51–70. 2009.