Crynko

Trust No One? Ink it!

Introduction

It has become a prevalent issue where companies and organizations are reluctant to add proper security measures to their websites. Such mistakes range from storing users' passwords plain text to using outdated or easily-cracked encryption methods. The consequences result in user data being easily revealed and exposed to the Internet and, in some cases, causing widespread panic as users use this one password for all websites that they venture upon and use. While it is repeated that users shouldn't be using the same password for different websites, why should users be expected to fix a security issue that is caused by the company themselves?

Crynko is an attempt at fixing this issue. Crynko is a method of putting the control of the password in the hands of the user rather than the company hosting the accounts these users use. This method of security ensures that if a company is negligent in user security and cuts corners that will leave user data vulnerable, all other accounts that the user holds on other websites will be safe from malicious intent.

Crynko uses something called INKing, which uses an imitation of two-factor Authentication. The process of INKing will ensure further security by encasing the password in both a salt and the other factor of authentication(i.e. security questions pertaining to personal information).

What is INKing?

INKing is the process of taking the password and hiding it in a bunch of stuff that will imitate two-factor authentication. The stuff that the password will be hidden under is the salt, which will be used for hashing the password, and the two-factor authentication string. These components will make up the entire string *R*, which will be run through the *SHA256* function. The equations assume that + means 'append.'

$$R = STR_{salt} + STR_{pass} + (STR_{2AuthBits \otimes salt})$$
$$Sha256(R)$$

The salt will be a numerical value that is in between 2^0 and 2^k, where *k* is a value determined by the developer or user. It is recommended that k is in between 16 and 64 for better security. A higher value of *k* will be slower, but more secure.

An example would be if the salt is 2^{64}, the password is some password, the 2Auth string, or the second part of the authentication, is a car name, and the URL of the website a user visits is *www.google.com*. One of the problems that could arise is the user's device could be infiltrated and data stolen. This would mean that the system is broken and not viable. One reason for this is the necessary data for the system to work would be the salt value and the url. The following represents the relation for the storage.

$$(URL,\ Salt)$$

If a malicious user obtains your information about the salt and the URL that this salt is used on would be easier since the salt and the URL are already known now. To solve for this issue, the URL can be encrypted with the password and the factor authentication string, and then hash the result. The following equation represents this process.

$$Hash_{URL} = (STR_{pass} \otimes STR_{2FAuth}) \otimes URL$$
$$Sha256(Hash_{URL})$$

The method ensures that if the malicious user somehow obtains the salt there will be other information required, which is not easily obtainable, to find the original url. All the other information required for the process will also be very difficult to obtain using this process. The hash of the URL and other content will then be stored in local storage along with the salt value.

Retrieving the URL and Salt

The salt will be retrieved via the URL as the URL will be the primary key. When the user initiates a login to the website, the password will be retrieved by Crynko. The software will also retrieve the 2FAuth string from the user. These values will be XOR'd against each other, which then will be XOR'd with the URL string. Crynko will search through the list of local storage URLs, which then, if found, will retrieve the salt for the pseudo-password. If not found, there will be an error.

Algorithm for process of password development

Registration

1. When user sends a GET or POST command to server, retrieve password from box that has 'password' type

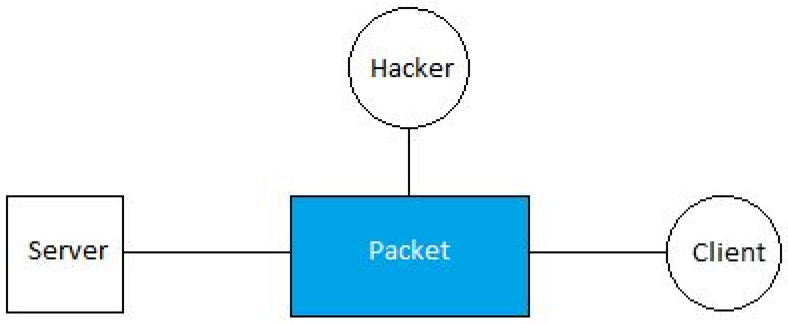
2. Ask for answer to question(2FAuth STR)
3. Run method *AuthPassW2Auth* method.
4. Set value of password textbox to return value of *AuthPassW2Auth*
5. Store 2FAuth STR and salt in local database

Login

1. When user sends a GET or POST command to server, retrieve password from box that has 'password' type
2. Ask for answer to question(2FAuth STR)
3. Run method *AuthPassW2Auth* method.
4. Set value of password textbox to return value of *AuthPassW2Auth*

Problems to Address

While ideally secure, there are some setbacks that exist for Crynko and the INKing method. Not only do these setbacks exist within the inner-workings of the concept, but of the problems with consistencies of websites and how they run. One of these issues is the issue of sending/receiving passwords across a network rather than developing the hash on the server. Figure 1 shows this particular problem.



In the figure, the client sides a packet of data to the server. However, the hacker could, through some method, steal these packets and obtain the data the client used to login into their account. At that very moment, the hacker has obtained the information for the client's username and password. Unlike the method of hashing the password on the server and preventing this, the hacker now has access to the plain text pseudo-password, which would be used on the server. While this is the case, this problem would also exist with a regular password that is plain text and transmitted in a packet to the server. This makes the problem moot.

Another problem to address, which is one of the major potential problems, is the problem of websites having consistent password requirements. One website may require that '!' or capital letters be required within the password. Others also have a maximum length. This would hinder Crynko because Sha256 hashes are 64 character hashes and if a website limits password length to 16 characters there would be an issue. There a8887766re two approaches to this problem. One approach is to simply reduce the size of the hash to the size of the required password length of the website. The second approach is to promote a consistency of no maximum password length. The latter approach is the most effective because of the probabilistic implications of a more difficult process of brute-forcing passwords.

License