Stronger Encryption Using a Weak Password

Oleg Mazonka[1], *2016*

***Abstract*—This paper presents an idea how to use a weak password for encryption while the encryption retains a stronger level of security. This method is based on time spent by a computational device required to encrypt and decrypt.**

I. Introduction

Suppose Alice wants to encrypt a secret *message* "hello" and pass the *ciphertext* (the encrypted message) to Bob who decrypts and reads the original message. Alice and Bob use a shared password "john" to encrypt and decrypt. Alice and Bob do not want Eva to read the message, but they are aware that Eva knows how to decrypt the message and learns the ciphertext. What Eva does not know is the password "john". However Eva has a hackers dictionary of a million passwords including "john"; and she can try each password from the dictionary in a hope to decrypt the message. If decryption is fast, then trying all million passwords is not a big job. Can Alice and Bob make sure that Eva is not able to decrypt and read their secret message?

There are two possible solutions. One is to use a complex password e.g. "john34xrvq968b". But Alice and Bob already have to remember so many different passwords that remembering another complex password is too difficult for them. So let us ignore this case. The second solution is to make decryption expensive. In other words every time Eva tries a new password, the program spends a fair amount of time to decrypt and to give the result whether the password matched or not. If one decryption takes 30 seconds, then Eva has to wait a year to try all the passwords from the dictionary. Of course Eva can use a supercomputer with parallel computation to reduce that time, but in any case a lot of *computational work* has to be done to break the encryption.

II. Strengthening Password Using Time

The most obvious solution is to increase the complexity of the decryption by extra computational work. For example, the original password can be digested by a hash function in a number of cycles before applying to encryption. However, the problem here is – how much of the 'extra work' is enough? With a fixed number of cycles the number must be known that adds a parameter to the algorithm. If you add too little, then it does not make it too hard for Eva to break the decryption. If you add too much, then the decryption for Bob can be unusable on a weak computational device.

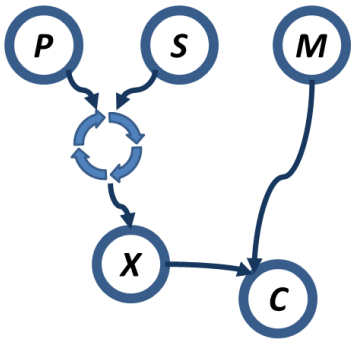
A solution is to let the algorithm increase the decryption complexity by Alice giving a certain amount of time when encrypting. From Alice's point of view – she starts the encryption process and then decides when to stop. The amount of time when she waits defines the complexity of the future decryption: the longer period, the more time-consuming decryption is.

In this case the decryption process must solve a corresponding computational problem. On a similar computational device this would take as much time as Alice waited. For Eva, every attempt to match the password requires the same computational effort, i.e. spent time.

III. A Cipher Example

Let us consider a cipher presented in the **Diagram**. To encrypt message M Alice uses a one-way function to generate the *cipher pattern* X out of the password P and the *salt* S. Salt is a random string added to the password to make the cipher pattern different for every encryption. Obviously the salt has to be passed together with the encrypted message, so Bob is able to generate the same cipher pattern. For Eva, however, the salt prevents pre-digestion of the dictionary to speed up cracking the password.

Diagram



Instead of using the cipher pattern immediately Alice lets the algorithm to digest it again and again in a cycle until Alice presses stop. The algorithm selects the best pattern according to a predefined rule; for example, the smallest or the greatest. This rule ensures that the generation is logarithmic in time – the number of new best found patterns is $O(\log n)$. The found cipher pattern X is used to encrypt the message M producing the ciphertext C. To get C out of M and X any strong cipher algorithm can be used.

The decryption process works in a similar way. However this time every new best found cipher pattern is tried to decrypt the message. This process continues until the message is decrypted. The decryption diagram is the same with the exception that the message M and the cipher text C are swapped.

[1)]Hasq Technology Pty Ltd, Email: om@hasq.org

An alternative to cyclic digestion of cipher pattern can be produced by an enumerated suffix appended to the password, such as $1, 2, 3, ...$ changing the password to "john1", "john2", etc. If Alice encrypts the message for herself to decrypt later, she may remember a part of the suffix that can make decryption faster. For example, while encrypting, the algorithm has found the combination "$salt$+john5663058" producing the smallest X and reported to Alice the suffix 5663058. If Alice remembers the first digit 5 and uses "john5" to decrypt, the decryption will take 10 times faster because it has to run only up to 663058 instead of 5663058. The suffix can be constructed of any symbols. The only condition is that it is selected from a reproducible sequence.

IV. A Numerical Example

Let us chose as an example the following:

- a short one-way function - MD5 hash;
- a suffix option with decimal numbers;
- a message represented in a 36-digit alpha-numeric number base: 0, 1, 2, ..., 8, 9, a, b, c, ..., y, z, 10, 11, ..., 19, 1a, ..., 1z, 20, etc;
- `XOR` function as a cipher;
- salt as a string of 4 digits – "2016"; and
- salt as the message prefix to detect successful decryption.

Alice wants to encrypt "hello". She adds the prefix and converts it to a hexadecimal number using 36-digit base:

$$\texttt{2016hello} \rightarrow \texttt{522479efddc}$$

Now Alice runs the algorithm generating cipher patterns from salt, the password, and suffix:

$$md_5(\texttt{2016john1}) = \texttt{ccf6fb7c2640169b4425254034b4efee}$$
$$md_5(\texttt{2016john2}) = \texttt{9cc8ee22976970815c4cbd6e8343cc6c}$$
$$md_5(\texttt{2016john3}) = \texttt{dc9057dbe33fa62218643f3f8f85262b}$$
$$md_5(\texttt{2016john4}) = \texttt{3fb1ebff42fb505f3c422e0841b183a5}$$
$$md_5(\texttt{2016john5}) = \texttt{f82afcf06ea74b413dc994652b192957}$$
$$...$$

The algorithm selects the smallest found cipher pattern. If Alice stops the algorithm after suffix 3, then the selected suffix is 2 because `9cc8`... is less than `ccf6`... and `dc90`.... After waiting for a few seconds Alice stops the algorithm producing

$$md_5(\texttt{2016john5663058}) =$$
$$= \texttt{000000413c3a1a14c0f67686bf6f2875}$$

Alice digests it once more to make the cipher pattern bits random:

$$md_5(\texttt{000000413c3a1a14c0f67686bf6f2875}) =$$
$$= \texttt{f02df44b30f511a9880788ade933dd00}$$

and `XOR`s it with the original message in hex format:

$$\texttt{522479efddc} \oplus \texttt{f02df44b30f511a9880788ade933dd00} =$$
$$= \texttt{f02df44b30f511a988078d8faead20dc}$$

The encryption process is finished. Alice sends this result together with the salt to Bob:

$$\texttt{2016} + \texttt{f02df44b30f511a988078D8FAEAD20DC} \rightarrow \text{Bob}$$

Bob, knowing the salt and the password, starts generating cipher patterns in the same way as Alice did. However, this time once the algorithm finds the best cipher pattern, it tries to decrypt the ciphertext. The first cipher pattern tried is

$$md_5(md_5(\texttt{2016john1})) =$$
$$= md_5(\texttt{ccf6fb7c2640169b4425254034b4efee}) =$$
$$= \texttt{83320c66c4d83694ad674f3bc43d3cf7}$$
$$\texttt{8332...} \oplus \texttt{f02d...} = \texttt{731ff82df42d273d2560c2b46a901c2b}$$

The lowest 64 bits convert to a message:

$$\texttt{731ff82df42d273d2560c2b46a901c2b} \rightarrow \text{kgnysp118tgr}$$

This attempt to decrypt fails because Bob expects the text of the message to start with 2016. The algorithm keeps running:

$$\text{2016john2} \rightarrow \text{1de6mpx8lcfsg}$$
$$\text{2016john4} \rightarrow \text{34ngdfaxtpbb0}$$
$$\text{2016john6} \rightarrow \text{28rrdspptjoz7}$$
$$\text{2016john15} \rightarrow \text{2f2zuvcip0zr}$$
$$\text{2016john24} \rightarrow \text{1y0tc1zxgpa8c}$$
$$...$$
$$\text{2016john2933482} \rightarrow \text{1gwi9xz9bw0x8}$$
$$\text{2016john5663058} \rightarrow \text{2016hello}$$

Finally at some point the message decrypts into "2016hello" that stops the algorithm. This stop condition is weak because the chance that random decryption would start with 2016 is too high. In real life this condition can be stronger; for example, simply using a longer sequence.

Bob spends the same time on decryption as Alice on encryption if they use similar computational devices. A small overhead for Bob related to trying a cipher pattern to decrypt and to test for successful decryption is negligible because the number of tries grows only logarithmically with time.

V. Security Considerations

Eva knows the salt and the ciphertext. She needs to run all the passwords from her hacker's dictionary. With the suffix used in Alice's encryption 5663058. Eva has to run the hash function at least this number of times for each entry in the dictionary. The best strategy for Eva is to select a parameter – the maximal number for the suffix – and run decryption up to that number. If it does not work, then increase the parameter and run again. If Eva had chosen 6000000 and run each password from the dictionary up to this number, she would break the encryption.

VI. Conclusion

Using short and simple passwords is often insecure. If the adversary has the access to hashes of passwords, the passwords can be broken by brute force. In this paper a simple method that leverages encryption time to make decryption harder, hence, improving the security, is presented. Its nice feature is that the hardness of decryption is tied to the encryption time (determined by the user) and not to some fixed parameter.