Written by: Adam Berent
Edited by: Paul LaCivita

ABI SOFTWARE
DEVELOPMENT

Key Iterations & Cryptographic Salts

V.1.2

Preface

The following document discusses the use of key iterations and cryptographic salts to stop dictionary attacks in password based encryption (symmetric cryptography).

Introduction

One of the most powerful attacks one can mount on encrypted data is a *Dictionary Attack.* A dictionary attack is a form of a brute force attack, which simply tries every single combination of a key against encrypted data. However, in most cases, this is not needed. User pass-phrases are unfortunately sometimes based on real words, dates, names, etc. We can eliminate most of the pass-phrase combinations by simply testing for most probable 30,000 words. An English dictionary is a good place to start, hence the term *Dictionary Attack*. This means that a key with a 128 bit key space, which has 3.4 x 10^38 possible combinations, has just been reduced to just over 30,000 (somewhere close to 15 bits). A computer that can process just 1 pass-phrase per second can run through the dictionary in just over 8 hours. Here are the basic steps in a dictionary attack:

1. Build a *Dictionary* containing a list of possible words or word combinations that are likely to be used as passwords. This part is easy since there are many of these pre built dictionaries already available on the internet.
2. Intercept encrypted data.
3. Attempt to decrypt data consecutively using every single pass-phrase in the dictionary.

Dictionary attacks are very powerful since:

1. You can't stop them because they rely on the ability to decrypt files (something legitimate users must be able to do).
2. You can't entirely stop users from choosing stupid passwords. Users will always want passwords that are easy to remember. Making users add numbers and upper case letters to their password will not stop dictionary attacks. It will just make them a bit more time consuming.

Since we can never entirely stop the attacker from attempting to execute a dictionary attack, we have to make it so time consuming that it becomes impractical.

This is where Cryptographic Salts and Key iterations are used to protect the encrypted file from dictionary attacks. This is achieved by:

1. Making it more time consuming.
2. Making it impossible to pre-compute passwords before having the actual encrypted file.

Key Iteration Count

First, we never use the user provided pass-phrase as is. We have to convert it to a usable key. The simplest way is to create a hash of the key using a message digest algorithm such as MD5 or SHA-1. The iteration count is the amount of times we perform this function to derive the final encryption key. In each step we take the previous hash result and hash it again. This is done to simply make it more time consuming for the attacker. An end user might not notice a 10 second delay during decryption however it could complicate things when you need to perform this 10 second function 30,000 times in a row. Since the attacker cannot perform the decryption until he has the actual key used to decrypt the file he has to endure the delay for every single key in the dictionary.

Random Salt

A salt complicates things even further for the attacker. As is, it would still be possible for an attacker or group of attackers to pre-compute a key from every pass-phrase over a period of time. They could then use this pre-computed dictionary file to run the attack on the encrypted data.

A salt is 64 bits of random data that is added to the key before the pass-phrase. This means that for every pass-phrase, a user creates, there are additional 64 bits of random data that cannot be predicted. An attacker would have to pre-compute one dictionary entry for every combination of the 64-bit salt in the key. This is something that would practically take too long.

The salt is usually stored in the beginning of the header in the encrypted file. This way the user that is attempting to decrypt the file can combine their pass-phrase with the random salt and produce the decryption key. An attacker will also have access to the salt since it is not encrypted. However it does not matter since the salt only prevents the attacker from pre-computing the dictionary before the file is created.

Salts are generally equal to the block space of the encryption algorithms. In most cases 64 bits. Salts also have to be computed using a pseudo random number generator such as Yarrow. The Rand() function built into compilers, is not random enough for cryptographic purposes.

S2K (String to Key)

S2K stands for string to key. It is a description used to define the function used to take a user type password and convert it to a salted and iterated key that can be used for encryption.

The following is a description of the S2K function used in the Open PGP standard:

The key components of the S2K function are:

Message Digest Algorithm (ex. MD5)
Pseudo Random Number Generator (ex. Yarrow)

The main goal of the S2K function is to create a hash from the user chosen pass phrase and random salt to produce the encryption key.

Usually hash functions (message digest algorithms) have a block size that is less then the required length of the encryption key. This is why in most cases we have to repeat the hashing procedure several times and concatenate the output to form the final key. In doing this, we also want the next portion of the key to be different from previous, yet still dependent on the pass phrase. This is why after the first time we hash the key we preload the hash function with bytes of 0s equal to the current loop count -1.   The hash phases are as follows:

1st hash no preloading
2nd hash 1 byte of zeros
3rd hash 2 bytes of zeros
4th hash 3 bytes of zeros
And so on until we have enough key material to stop.

Most implementations of hash algorithms have an update function so that you can preload (update) the function with the appropriate amount of zeros.

For example to produce a 256-bit key using the 128-bit MD5 hash algorithm we would have to:

1. Clear the hash function.
2. Update the hash function with the salt.
3. Update the hash function with the pass phrase
4. Get hash and store it as the first part of the key.
5. Clear the hash function.
6. Update the hash function with one byte of 0s.
7. Update the hash function with the salt.
8. Update the hash function with the pass phrase
9. Get hash and store it as the second part of the key.

In a case where the hash algorithm output is greater then required by the key, we use the leftmost bytes of the hash output.

The function above is repeated as many times equaling to the key iteration count. The difference being that we would assign the key result of the previous function to the pass phrase input of the next function.

For example:

Iteration count 1
        1. Perform above steps 1-9
        2. Assign result to pass-phrase
Iteration count 2
        1. Perform above steps 1-9
        2. Assign result to pass-phrase
This continues until we complete the amount of iterations specified by the iteration count.

Conclusion

Following the above steps will make it much harder for an attacker to execute a dictionary attack against an encrypted file. However, it does not make it impossible.

For example, if we set the iteration count high enough, so that it takes 10 seconds for an attacker to complete the S2K function, then it would take close to 83 hours for an attacker to search through a dictionary of 30,000 possible pass phrases. This makes it still possible to break a weak key. If the attacker has a very fast PC, the amount of time can severely decrease, since it would be easier for the attacker to complete the iterations.

We can always increase the iteration count so that it takes longer for the attacker to complete the S2K function. However, the longer the iteration count is, the longer a legitimate user has to wait to decrypt a file.

The unfortunate truth is that if the users pass phrase is weak, it will still be possible to break it using a dictionary attack.   When building encryption software it is always a good idea to prevent users from choosing weak passwords. It is a good idea to quickly search trough a small dictionary of English words for the users pass phrase, and if found suggest that they change it. Since at this point we have the original pass phrase and we do not have to derive it through the S2K function, this search can be fairly quick.

Another good idea is to include a random number generator in the software so that a user has the ability to select a random pass-phrase. This has its own problems since it's extremely hard for users to memorize such pass phrases.

The last and most important advice is not to make it easy for an attacker to retrieve the encrypted data. Without having local access to the file there is no way to mount a dictionary attack in the first place.

This document was written by Adam Berent and can be distributed without copyright as long as proper credit is given. If you would like to contact me feel free to do so at aberent@abisoft.net or visit www.abisoft.net.