Security Lab – Secret Key Cryptography

VMware

- This lab can be solved using any Java IDE. No VM is needed.

Introduction

This lab highlights some of the properties of a secret key's work factor. Recall that the work factor is the average number of tries you need to make in order to find the correct key, ad the work factor in bits is the binary logarithm of the work factor.

This lab simulates the effects of ransomware. This is malware that infects your computer, encrypts a few important files, sends the keys off to a command-and-control server and then deletes the original files. You will thus be left with a bunch of ciphertext files, but without the corresponding plaintext files or the key that was used to encrypt the file. You will then be contacted by the authors of the ransomware and asked to pay them money. If you do pay the money, you will sometimes get instructions how to decrypt the files. Sometimes, even the ransomware authors won't know the key, or won't care enough to actually provide you with it, even though you have paid the ransom. The bastards.

In the case of real-life ransomware, if you don't get the key somehow, your files would now be gone: this is often professionally-written software that takes great care to ensure that you cannot recover the key except from the authors. And the encryption algorithms used are normally secure so that there is no known way to break the encryption to get your files back. But perhaps the program that you are about to encounter is not so professionally written, and you can obtain the plaintext somehow?

Preparation

Downloading and Running the Code

Step 1. Start a shell. In a directory of your choice, type

```
git clone https://github.zhaw.ch/neut/itsec-secret-key-crypto.git
```

This should give you a directory called `itsec-secret-key-crypto`, a Java project containing our simulated ransomware. You should be able to import this project into any Java IDE; we have been using IntelliJ, but Eclipse, VS Code, and indeed any IDE should work.

Step 2. Build the project in your IDE or with maven. This should leave you either with a bunch of class files or (if you used maven to build the project) with an executable Jar file. If you want to use maven, type

```
mvn package
```

Step 3. Now create a file called "plain" in the itsec-secret-key-crypto directory, containing some contents of your choice. Warning: This file will be the target for the simulated ransomware, and will therefore be removed by the program, so do not use any data here that you want to keep and of which you have no backups!

Here, we create a small text file and verify that it is there:

```
$ echo "Hi, I am a quite small plaintext but not that small" > plain
$ ls -l
total 256
-rw-r--r--  1 neut  staff      80 Oct 12 15:00 itsec-secret-key-crypto.iml
-rw-r--r--  1 neut  staff  118752 Oct 15 14:19 mystery
-rw-r--r--  1 neut  staff      52 Oct 15 15:46 plain
-rw-r--r--  1 neut  staff    2156 Oct 15 10:50 pom.xml
drwxr-xr-x  4 neut  staff     128 Oct 12 15:00 src/
drwxr-xr-x  9 neut  staff     288 Oct 12 16:15 target/
```

Step 4. Now we run the ransomware (called Ransom) with the command-line arguments "`plain`" and "`cipher`". For example, if you have built the program with maven, you should type:

```
java -cp target/secretkey-1.0-SNAPSHOT.jar ch/zhaw/its/lab/secretkey/Ransom -ransom
plain cipher
```

You should see something like this:

```
MUAHAHAHAHAHAHA! The original file "plain" is gone,
the key is also gone. The encryption algorithm is AES.
Now you must pay $$$ to get the files back! Resistance is futile!
```

We can verify that the file "`plain`" no longer exists, but that there is now a file called "`cipher`":

```
$ ls -l
total 256
-rw-r--r--  1 neut  staff      80 Oct 15 15:48 cipher
-rw-r--r--  1 neut  staff      80 Oct 12 15:00 itsec-secret-key-crypto.iml
-rw-r--r--  1 neut  staff  118752 Oct 15 14:19 mystery
-rw-r--r--  1 neut  staff    2156 Oct 15 10:50 pom.xml
drwxr-xr-x  4 neut  staff     128 Oct 12 15:00 src/
drwxr-xr-x  9 neut  staff     288 Oct 12 16:15 target/
```

That file certainly looks encrypted:

```
$ cat cipher
\???.a?^w??!?e?L1Cц}AB'M?\f??lQ?
```

(This output will look different on your terminal.)

Luckily for you, the attackers left their source code behind. Perhaps they made a mistake that enables you to recover the key and hence your original files?

Exercises

Theoretical Groundwork

Exercise 1. The work factor of a problem is defined as the average number of tries one needs to do in order to solve that problem by trial and error. If you have a problem in which you need 1 try with probability 1/2, 2 tries with probability 1/3, and 3 tries with probability 1/6, what is the work factor of this problem? Give the solution as a fraction (e.g., "1/9"), not in decimals (e.g., "0.11111…").

Exercise 2. You want to find the key for a ciphertext by trying out the various possible keys. There are N keys.

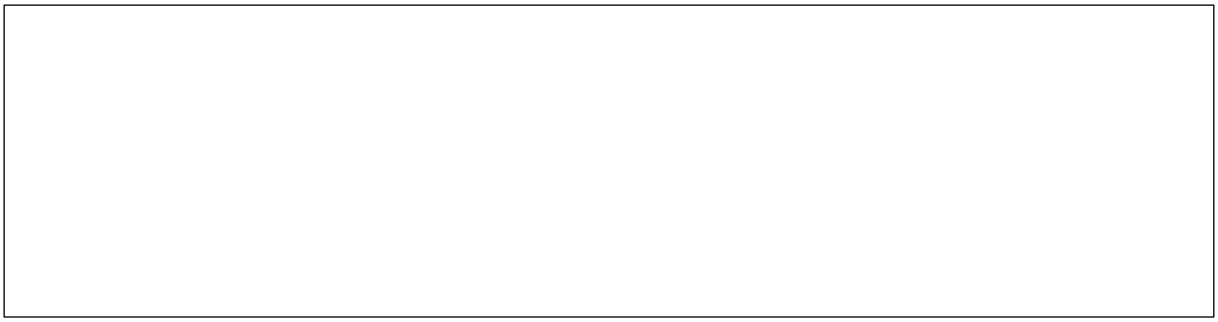
a) If those keys are *uniformly distributed* (each key is equally probable), compute the work factor. (The lecture slides already gave the answer, (N+1)/2, but I want to see the computation.)
b) Argue that if N is very large, it doesn't matter numerically whether we use (N+1)/2 or N/2.
c) If those keys are *not* uniformly distributed, then some keys are more probable than others. Assuming you want to spend as little time as possible on the problem, suggest a strategy with which the work factor is less than (N+1)/2. Explain!

Exercise 3. The cipher that the attackers used is simply "AES" (with no extensions to the name). Look up AES's default key length. Assuming the key was chosen uniformly at random, what is that key's work factor? Also give the work factor in bits. Show and explain your work.

Exercise 4. Make a *reasonable assumption* about how fast your computer can try a single key on a file of 512 bytes. You should do this by looking up your computer's clock speed and then looking up approximate values for "cycles per byte" for AES on your CPU. Cycles per byte is a standard performance figure for cryptographic speed. Then compute how long that computer would need to exhaust the work factor. Can you get it done by Friday? How about next Friday? **Important note:** If you can't find the cycles per byte for your particular computer, use the numbers for *any* modern CPU. If you're stuck, one reasonable answer can be found on Wikipedia. Even then, *absolute precision is not essential, order-of-magnitude estimates will suffice*. If you still don't know which of the numbers to choose, choose the one that means that your CPU is faster (higher cycles per second, lower cycles per byte).

Exercise 5. As the previous calculation shows, you cannot hope to break the key by brute force alone. What assumptions would have to change so that you have a good chance of recovering the key nevertheless?

Exercise 6. Compute the maximum work factor that the key could have so that your computer (under the assumptions from above) could break the key in 1 minute.



Analysing the Code

If we look at the code, we find that the same program that so dastardly encrypted and deleted our files can also run in a decryption mode where we enter the key on the command line:

```
java -cp target/secretkey-1.0-SNAPSHOT.jar ch.zhaw.its.lab.secretkey.Ransom -pay
cipher decrypted 00112233445566778899aabbccddeeff
```

This attempts to decrypt the file `cipher` into the file `decrypted` using the key `001122…eeff`. If we could find the key, we could decrypt our file, even without paying the ransom! If we use this very key on the file `cipher`, we unfortunately get an exception:

```
Trying with key 00112233445566778899aabbccddeeff
Exception in thread "main" javax.crypto.BadPaddingException: Given final block not
properly padded. Such issues can arise if a bad key is used during decryption.
```

The program has started to decrypt the file `cipher` into `decrypted` but didn't finish the last block because it sensed that something wasn't right. This is one indication that this is not the right key. The program has left us with a partial decrypt in `decrypted`, which we can look at with, e.g., hexdump:

```
$ hexdump -C decrypted
00000000  d6 0f bb 59 d6 2a 1f 0a  eb 79 93 ab d7 7e ea 69  |...Y.*...y...~.i|
00000010  31 f3 b4 db 4e d3 f1 56  ac 34 20 e5 81 a5 9b de  |1...N..V.4 .....|
00000020  57 dd c9 e0 68 9c b4 16  bd 4c f1 35 c8 13 70 89  |W...h....L.5..p.|
00000030
```

This looks quite random and is definitely not our original file. This is not surprising, since the key we used for the decryption, `001122…eeff`, was in all likelihood not the one that was used for encryption. When using AES, decrypting a ciphertext with the wrong key will yield random-looking plaintext. This is another indication that we haven't found the right key.

Estimating Randomness

One way to distinguish the right key from a wrong key is the observation that *using a wrong key will lead to a random output*, and the *right key will lead to the decrypted original file*. If we could distinguish random files form less random files, we could decrypt the mystery file with a trial key and see if the decryption was random. If it is random, the trial key is not the right one and we try another key.

How do we distinguish random files from others? One (imperfect) way to do that is to observe that in a random file, all bytes will occur with approximately equal frequency, whereas in a nonrandom file, some bytes will occur much more often than others. It turns out that one can measure how uneven this frequency distribution is. If in a file F of length n bytes, byte k (0 <= k < 256) appears n[k] times, we define f[k] as the relative frequency of byte k, i.e., f[k] = n[k]/n. Then the randomness H(F) of the file F is defined as

$$H(F) = -\sum_{k=0}^{255} f_k \log_2 f_k.$$

Should some f[k] be zero, we don't include that term in the sum (which otherwise would be infinite).

Here is what you need to know: *If the file is random, H(F) will be close to 8, but if the file is not random, H(F) will be much lower.*

Exercise 7. If a file F of length n consists only of `0x00` bytes, what is H(F)? Show your work.

Exercise 8. If a file F of length n (for large n) consists of uniform random bytes, what is H(F)? Show your work.

Exercise 9. Write a program *in a language of your choice* that takes on its command line a list of file names and that outputs H(F) for each file. For example, for the file `mystery`, a file that was part of the repository, and `FileEncrypter.java`, one of the source files of the ransomware, your program should produce the following output:

```
mystery: 7.998687714859995
src/main/java/ch/zhaw/its/lab/secretkey/FileEncrypter.java: 4.82373026044958
```

Exercise 10. From looking at the output of this program, which of the two files do you think contains "more random" data? Assuming that both files had been produced by the ransomware in decryption mode, which one is more likely to contain actual plaintext? Explain.

Exercise 11. From the preceding exercises, you should now have a good idea how to distinguish correctly decrypted text from wrongly decrypted text, provided that the original plaintext is in a natural language. Write a program *in Java* that takes as input a file and that outputs `true` if the input is likely to be a natural-language text, and `false` otherwise. (You can write the program in another language if you want, but you will have to use what you wrote in the next exercise.) Download some examples of natural-language text from the Internet and test your program. Also create some files with random data and test your program with those as well. On Unix-like systems, the special file `/dev/urandom` provides you with a convenient source of random numbers on the command line.

Exercise 12. Extend your program so that you use (by copy-and-paste) the `decrypt()` routine and all needed helper routines from the `FileEncrypter` class in the ransomware to do the following:

- It accepts as input a `byte[]` containing ciphertext, and a `byte[]` containing a key
- It attempts to decrypt the ciphertext
- It returns `true` if it thinks that the decrypted output is in some natural language

Now you have the basics for a program that allows you to automatically try keys until you have found one that makes the input decrypt to a natural-language plaintext.

Cryptanalysis of the Ransomware

We have found out in the earlier sections that if the AES key is chosen randomly, we have no practical chance to recover the key by brute-force, i.e., by trying all keys until one fits. Therefore, there are only two possible ways of getting our files back: either the programmer made some security-relevant mistake while coding up the crypto in the program, or the keys aren't as random as they should be.

We can tell you now that the implementation of the crypto is solid, there are no security-relevant mistakes to be found there. The only chance therefore is that the keys are perhaps not random enough.

As the code in `FileEncrypter` class reveals, the ransomware program uses AES in a mode known as CBC (we will discuss this in the lecture later). For this variant of AES, CBC mode splits the plaintext up into blocks of 128 bits, or 16 bytes: P[0], P[1], … But because of the way it is constructed, CBC needs an artificial zeroth block, P[0]. This block is known as the *initialization vector*, or IV. This IV is provided by the program (*FileEncrypter.java*, lines 46—50). As you can see, it is generated by a random number generator, *just like the key*, and is put in front of the ciphertext. So *if the key is random, the IV and therefore the first 16 bytes of the file should also be random*. Let's check that with hexdump:

```
$ hexdump -C mystery | head -3
00000000  98 0b ab 77 66 01 00 00  00 00 00 00 00 00 00 00  |...wf...........|
00000010  7e 1c 58 a0 d8 9a a5 bc  47 57 41 9b 82 e4 08 50  |~.X.....GWA....P|
00000020  ec 6e 79 8c 11 e6 db 48  6a fd 44 35 3a 16 b1 43  |.ny....Hj.D5:..C|
```

Even if it only concerns the IV, which is not secret, *that* is very interesting indeed! Instead of 16 random bytes in positions 0 to 15, fully ten of those 16 bytes are zero! Something is rotten in the state of Denmark!

Exercise 13. Look at the code and find out how the IV is generated, exactly. How did it happen that ten out of the sixteen bytes in the IV are zero?

Well, now we've found out why the IV wasn't as random as perhaps it should have been. Unfortunately, this doesn't mean anything, since we only have the one mystery file, and as long as IVs aren't repeated, nonrandom IVs do not affect security (in this use case).

Exercise 14. Or do they? Look at the code again and find how they *key* is generated. Is this related to how the *IV* is generated? (Yes) Are therefore key and IV related? (Yes) If you know the IV, can you say something about they key? (Yes) This should give you an idea of how to extend the program you wrote above so that you can now indeed find the key, given only the ciphertext file. Explain the flaw and then write the program.

Exercise 15. Given your new knowledge, estimate the work factor of the key *in bits*. An estimate suffices, absolute precision is not required.

This should make it clear that the work factor of messages (or key material) depends on what we know about that message (or key material). Here, our knowledge about the key material reduced the work factor from 178 bits, which is impossible to crack, to something even a student's laptop can handle easily.

Putting it All Together

Exercise 16. Find the key that was used to encrypt the file mystery and decrypt it. What is the key? What is the text? Who wrote it? In what work does it appear, and where?

Points

In this lab, you can get up to **4 points**, by showing your instructor the program you wrote and how it hangs together, with a demonstration that it actually finds the key for the `mystery` file and and correctly decrypts it.