Lab 2

David Herel

This is my notebook for lab2 - communication security class.⚾

Exercise 1: why we do all of this

> This is the easiest exercise, but also the most important one.

> Write in your report the following sentence:

```
I, <your name here>, understand that cryptography is easy to mess up, and
that I will not carelessly combine pieces of cryptographic ciphers to
encrypt my users' data. I will not write crypto code myself, but defer to
high-level libaries written by experts who took the right decisions for me,
like NaCL.
```

> That's it. You will indeed get points for writing this sentence in your report.

> Some similar resources if the idea is not clear enough:

- [Foot-shooting prevention agreement](http://www.moserware.com/assets/stick-figure-guide-to-advanced/aes_act_3_scene_02_agreement_1100.png) from A Stick Figure Guide to the AES
- Why is writing your own encryption discouraged?
- One does not simply write their own cipher
- Don't roll your own crypto.

> Here is a list of recommendations that you can read later if you want.

- Cryptographic right answers, 2018 Note how it just more or less says "do not code crypto yourself".
- NaCL and

libsodium what you should use if you use crypto in production

> Finally, remember to stay up to date, crypto that was good yesterday can be broken today.

> That's it for the "soft" stuff. For the rest of the lab we will code that does *not* follow these rules, precisely so you can break it and see what happens when one makes careless crypto decisions.

```
I, David Herel, understand that cryptography is easy to mess up, and
that I will not carelessly combine pieces of cryptographic ciphers to
encrypt my users' data. I will not write crypto code myself, but defer to
high-level libaries written by experts who took the right decisions for me,
like NaCL.
```

Exercise 2: encrypt single-block AES

> Write a function `encrypt_aes_block(x, key)` that takes an input `x` with a size of *exactly* 16 bytes, and a `key` also 16 bytes long, and returns the result of encrypting this block with AES.

> *It should encrypt only one block*; make it fail (exception, assert...) if the input is not 16 byte long.

> Do not write the AES algo yourself (it is somewhat complex and not in the scope of this lab). Just defer to a library implementation that will encrypt the block for you. Maybe your library will force you to choose a block mode, in this case choose ECB and make sure you get only one block of output.

> What is the ciphertext of encrypting the plaintext `90 miles an hour` with the key `CROSSTOWNTRAFFIC` ? Answer in hex.

```
!pip install Crypto==1.4.1
```

```
Requirement already satisfied: Crypto==1.4.1 in /root/venv/lib/python3.7/site-packages (1.4.1)
Requirement already satisfied: shellescape in /root/venv/lib/python3.7/site-packages (from Crypto==1.4.1) (3.8.1)
Requirement already satisfied: Naked in /root/venv/lib/python3.7/site-packages (from Crypto==1.4.1) (0.1.31)
```

```
Requirement already satisfied: pyyaml in /shared-libs/python3.7/py/lib/python3.7/site-packages (from Naked->Crypto==1.4.1) (6.0)
Requirement already satisfied: requests in /shared-libs/python3.7/py/lib/python3.7/site-packages (from Naked->Crypto==1.4.1) (2.26.0)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /shared-libs/python3.7/py/lib/python3.7/site-packages (from requests->Naked->Crypto==1.4.1) (1.26.7)
Requirement already satisfied: charset-normalizer~=2.0.0; python_version >= "3" in /shared-libs/python3.7/py-core/lib/python3.7/site-packages (from requests->
Requirement already satisfied: certifi>=2017.4.17 in /shared-libs/python3.7/py/lib/python3.7/site-packages (from requests->Naked->Crypto==1.4.1) (2021.10.8)
Requirement already satisfied: idna<4,>=2.5; python_version >= "3" in /shared-libs/python3.7/py-core/lib/python3.7/site-packages (from requests->Naked->Crypto
WARNING: You are using pip version 20.1.1; however, version 21.3.1 is available.
You should consider upgrading via the '/root/venv/bin/python -m pip install --upgrade pip' command.
```

```
from Crypto.Cipher import AES

#encrypt single aes block - x and key has to be exactly 16 bytes long - otherwise error is popped
def encrypt_aes_block(x, key):
    if (len(x.encode("utf8"))) != 16:
        raise Exception('Message has to be exactly 16 bytes long.')
    if (len(key.encode("utf8"))) != 16:
        raise Exception('Key has to be exactly 16 bytes long')
    cipher = AES.new(key, AES.MODE_ECB)
    message = cipher.encrypt(x)
    return message


answer = encrypt_aes_block("90 miles an hour", "CROSSTOWNTRAFFIC")
print(answer.hex())
```

```
092fb4b0aa77beddb5e55df37b73faaa
```

The answer in hex is: `092fb4b0aa77beddb5e55df37b73faaa` .

Exercise 3: decrypt single-block AES

> Write a function `decrypt_aes_block(y, key)` that takes an encrypted block `y` (exactly 16 bytes) and decrypts it with a 16 byte `key` . As above, this should only operate on 16 byte blocks, and should defer to a library implementation.

> What is the decryption of the 16 byte block `fad2b9a02d4f9c850f3828751e8d1565` with the key `VALLEYSOFNEPTUNE` ?

```
import binascii

#decrypt single aes block - x and key has to be exactly 16 bytes long - otherwise error is popped
def decrypt_aes_block(y, key):
    if (len(y)) != 16:
        raise Exception('Message has to be exactly 16 bytes long.')
    if (len(key.encode("utf8"))) != 16:
        raise Exception('Key has to be exactly 16 bytes long')
    cipher = AES.new(key, AES.MODE_ECB)
    message = cipher.decrypt(y)
    return message


def hex2bin(my_input):
    return binascii.unhexlify(my_input)

answer = decrypt_aes_block(hex2bin("fad2b9a02d4f9c850f3828751e8d1565"), "VALLEYSOFNEPTUNE")
print(answer)
```

```
b'I feel the ocean'
```

We got the answer: `I feel the ocean`

Exercise 4: implement PKCS#7 padding

> Write a function `pad(x)` that takes input data with arbitrary size and returns the same data (or a copy of it) with some bytes appended to it, in this manner:

> - if the length (in bytes) is a multiple of 16 minus 1, append a single byte of value 1 (hex `01`).
> - if the length is a multiple of 16 minus 2, append two bytes of value 2 (hex `0202`).
> - if the length is a multiple of 16 minus 3, append three bytes of value 3 (hex `030303`).
> - ...
> - if the length is a multiple of 16 minus 15, append fifteen bytes of value 15 (hex `0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f`).
> - if the length is exactly a multiple of 16, append sixteen bytes of value 16 (hex `10101010101010101010101010101010`).

> Your output should have, therefore, a length always a multiple of 16 bytes.

> What is the output of `pad("hello")` ?

```
def pad(x):
    left_overs = len(x.encode("utf8")) % 16
    byte_to_copy = 16 - left_overs
    text_append = byte_to_copy.to_bytes(1, 'big')
    text = x.encode() + (text_append * byte_to_copy)
    return text

answer = pad("hello")
print(answer)
```

```
b'hello\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b'
```

Output of `pad("hello")` is `b'hello\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b'` .

Exercise 5: implement PKCS#7 unpadding

> Write a function `unpad(y)` that takes an input data `y` padded with the scheme above, and returns the same data without the padding.

> What is the output of `unpad("hello\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b")` ?

```
def unpad(y):
    left_overs = int.from_bytes(y[len(y)-1].encode(), 'big')
    if left_overs >= 17:
        return y
    return y[0:len(y)-left_overs]

print(unpad("hello\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b"))
```

```
hello
```

We have got correct output: `hello` .

Exercise 6: implement ECB encryption

> Write a function `encrypt_aes_ecb(x, key)` that:

> - takes an arbitrary sized input `x` ,
> - pads it so the length is a multiple of 16 byte,
> - cuts it to consecutive blocks of 16 bytes,
> - encrypts every block independently with `encrypt_aes_block()` from Ex. 2
> - concatenates the result

> Implement ECB mode yourself by following the steps above; you are not allowed to defer this task to your AES library of choice (but the library can still encrypt individual blocks through `encrypt_aes_ecb()` , of course).

> What is the encryption of the following plaintext? Use the key `vdchldslghtrturn` . Answer in hex.

>
> ```
> Well, I stand up next to a mountain and I chop it down with the edge of my hand
> ```

```
def encrypt_aes_ecb(x, key):
    padded = pad(x)
    for i in range(0,len(padded),16):
        temp = encrypt_aes_block(padded[i:i+16].decode("utf8"), key)
        padded = padded[:i]+temp+padded[i+16:]
    return padded

answer = encrypt_aes_ecb("Well, I stand up next to a mountain and I chop it down with the edge of my hand", "vdchldslghtrturn")
print(answer.hex())
```

```
883319258b745592ef20db9dda39b076a84f4955a48ba9caecd1583641cf3acac86acd5e5795de7895fab54481e9d8c3afc179c39412282eb8445ea2450e763df7282998a74baf19887c843b658f88
```

The answer is:
`883319258b745592ef20db9dda39b076a84f4955a48ba9caecd1583641cf3acac86acd5e5795de7895fab54481e9d8c3afc179c39412282eb8445ea2450e763df7282998a74baf19887c843b658f8891` .

Exercise 7: implement ECB decryption

> Write a function `decrypt_aes_ecb(y, key)` that decrypts ECB by reversing the steps of the previous exercise: slice in 16 byte blocks, decrypt blocks independently, concatenate, unpad. Again, do not defer to your AES library for these steps, except for individual block decryption through `decrypt_aes_block()`.

> What is the decryption of the following ECB encoded ciphertext? Use the key `If the mountains`.

```
792c2e2ec4e18e9d3a82f6724cf53848
abb28d529a85790923c94b5c5abc34f5
0929a03550e678949542035cd669d4c6
6da25e59a5519689b3b4e11a870e7cea
```

```
def decrypt_aes_ecb(y, key):
    for i in range(0,len(y),16):
        temp = decrypt_aes_block(y[i:i+16], key)
        y = y[:i]+temp+y[i+16:]
    return unpad(y.decode("utf8"))

answer = decrypt_aes_ecb(hex2bin("792c2e2ec4e18e9d3a82f6724cf53848abb28d529a85790923c94b5c5abc34f50929a03550e678949542035cd669d4c66da25e
print(answer)
```

```
If the mountains fell in the sea / Let it be, it ain't me
```

The answer is: `If the mountains fell in the sea / Let it be, it ain't me`.

Exercise 8: ECB ciphertext manipulation (cut and paste 1)

> The file `text1.hex` contains lyrics of a song, where each line is exactly 32 bytes long (31 letters + a newline character). In other words, the first line is in blocks 0-1; the second line is in block 2-3, etc.

> However, there is a small mistake! The first line and the third line have been unfortunately swapped, so the song is not correct anymore.

> 1. Have a quick look at the hex file. Can you quickly spot some obvious

> patterns? What fact can you deduce about the song lyrics?

> 1. At the end of the file, you can see a 16 byte block "alone"; however, all

> lines of the song lyrics are really 32 bytes long. Can you explain the presence of this last 16 byte block? Can you guess the plaintext of this block?

> 1. Restore the correct order in the ciphertext. In other words, swap the first

> line and the third line. You must do this operation by manipulating the ciphertext only, without decrypting. (The point is to show you that you can manipulate encrypted text without knowing the key!)

> 1. Then, decrypt the text with key `TLKNGBTMYGNRTION`. The first line should

> start with "People" -- what is the rest of this line?

```
text1 = "f55dd7f3f8a6ab401f534e3bd14e17e5a9e6a812719593e07cd2ae73c9233ce43130c68457ca3d0783e5a5beec8965b1ce2b204963fd41209775362f9db531e
#swapped first and third line
text1 = "5dd0fb1c4956cf54cd2d3189072347e63bc53a7b56501a157f44894c27bfc0f93130c68457ca3d0783e5a5beec8965b1ce2b204963fd41209775362f9db531e

answer = decrypt_aes_ecb(hex2bin(text1),"TLKNGBTMYGNRTION")
print(answer)
```

```
My, my, my, my generation
People try to put us d-down
(Talkin' 'bout my generation)
Just because we g-g-get around
(Talkin' 'bout my generation)
Things they do look awful cold
(Talkin' 'bout my generation)
I hope I die before I get old
(Talkin' 'bout my generation)
My generation
This is my generation, baby
My, my, my, my generation
My, my, my, my generation
Talkin' 'bout my generation
(My generation)
```

```
Talkin' 'bout my generation
(My generation)
Talkin' 'bout my generation
(Is my generation baby)
Talkin' 'bout my generation
(This is my generation)
Talkin' 'bout my generation
(This is my generation)
Talkin' 'bout my generation
(This is my generation)
Talkin' 'bout my generation
(This is my generation)
Talkin' 'bout my generation
(This is my generation)
```

Answer to the 1 question:

Yeah there are lines like this `3130c68457ca3d0783e5a5beec8965b1ce2b204963fd41209775362f9db531ef` , which are repeating periodically, so this has to be pop song with some catchy phrase.

1. question

It was probably padded

1. and 4. done via code above.

`try to put us d-down`

Exercise 9: ECB message crafting (cut and paste 2)

> 1. Write a function `welcome(name)` that
>   - first, concatenates three strings:
>   - `"Your name is "` (13 bytes)
>   - `name` (the input)
>   - `" and you are a user"` (19 bytes)
>   - then, encrypts the resulting string with the key `RIDERSONTHESTORM`
>   - finally, returns the ciphertext

> For questions 2-5, only operate through the function `welcome()` ; do not use the key directly. (Pretend you're an outside attacker, and therefore, you don't know the key.)

> 1. What is the ciphertext of `welcome("Jim")` ? Answer in hex.

> 1. Obtain the ciphertext of a block whose plaintext is 16 times the byte 16:

> `1010...10` .

> 1. Obtain the ciphertext of a block whose plaintext is "you are an admin".

> 1. Use these blocks (and perhaps other calls to `welcome`) to craft an encrypted

> message whose plaintext starts by `"Your name is "` and finishes with `" and you are an admin"` . In your report, write down your crafted ciphertext.

> For question 6, you may use the key directly:

> 1. Decrypt your encrypted message to make sure it is correct. In your report,

> write the decrypted message.

> 1. Could you quickly describe a real-world scenario where this could be a

> security issue?

```python
def welcome(name):
    #13 before and 19 after
    text = "Your name is " + name + " and you are a user"
    ret = encrypt_aes_ecb(text, "RIDERSONTHESTORM")
    return ret

answer = welcome ("Jim")
print(answer.hex())

#because of pad func. 10..10 is added so just take last 16 bytes
point3 = welcome("")
point3 = point3[len(point3)-16:]
```

```
print(point3.hex())

point4 = welcome("   you are an admin")
point4 = point4[16:32]
print(point4.hex())

point5 = welcome("David Herel   ")
point5 = point5[:32]+point4+point3
print(point5.hex())
print(decrypt_aes_ecb(point5, "RIDERSONTHESTORM"))
```

```
d4d7730a2d4255c88dead80a2ad924f2b114fddb898d7ef8abdfefef30d552863f62b0605102e0186402df7666edcec7
4e9eb1df207c25bebdcfc57385251689
7edb62ceff6a92e3a59029a06e5e622b
de24280e189ec5c1e880ca3ef5668ea54b9c6694fdc82189315fe5009c55ec047edb62ceff6a92e3a59029a06e5e622b4e9eb1df207c25bebdcfc57385251689
Your name is David Herel    and you are an admin
```

1. and 2. done

Cipther text of welcome("Jim") is `d4d7730a2d4255c88dead80a2ad924f2b114fddb898d7ef8abdfefef30d552863f62b0605102e0186402df7666edcec7` .

1. `4e9eb1df207c25bebdcfc57385251689`
2. `7edb62ceff6a92e3a59029a06e5e622b`
3. `de24280e189ec5c1e880ca3ef5668ea54b9c6694fdc82189315fe5009c55ec047edb62ceff6a92e3a59029a06e5e622b4e9eb1df207c25bebdcfc57385251689`
4. Decrypted message is `Your name is David Herel and you are an admin

` and it is correct

1. We can modify any information which was encrypted this way - we could insert ours.

Exercise 10: ECB decryption (cracking) byte-at-a-time

> In this exercise, you will decrypt the >content of a ciphertext without knowing the key!

> Let `SECRET` be the following >constant: "this should stay secret".

> Write a function `hide_secret(x)` that

> - concatenates two pieces of data:
>   - `x` (the input, which can >contain non-ASCII characters)
>   - `SECRET`
> - encrypts the result with the key > `COOL T MAGIC KEY`
> - returns the ciphertext

> To test your function, make sure > `hide_secret("just listen find the >magic key")` returns the following ciphertext:

> 45a306391112e09639cc44fa4d53c79e c90162749b6055bbc3d0811c0da6bd9b df3dccce5ff98e742ffdc33a1c8e84b9 d47e0182d8fa07c9291b25d8dab01199

> Now, perform the next questions by >pretending you don't know the key: only operate through `hide_secret()` .

> Write a function that discovers the >first character of the secret. Here is >how you can proceed:

> 1. Call `hide_secret()` with an input >containing fifteen times (16-1) the

> character `A` : "AAAAAAAAAAAAAAA". >In the first block to be encrypted, >what will be the last plaintext byte?

> 1. Call `hide_secret()` with the >following 16-byte inputs: "AAA...>AA\x00",

> "AAA...AA\x01"... In other words, >iterate through all possibilities for >the last byte of the plaintext, and >check the resulting ciphertext against >the first ciphertext block of question >1 above. If it matches, it means you >have discovered the first character!

> Now generalize this technique to >obtain all subsequent characters:

> 1. Say you know some bytes at the >beginning of the secret. To obtain the >next

> byte, decrement the number of A's. >When calling `hide_secret("AAA..AA")` >the plaintext of the first block will >look like:

> AAA..AAxxxxy ^^^^----- you know this >already ^---- crack this byte, >you know how to do it!

> Automate this in a loop. When you run out of A's, realize you can start back at 15 and attack the second block (and later, the third, and so on)

> How to find out when you have the complete secret (i.e. the termination condition for your loop):

> - You could continue cracking new bytes, but they will always have the value 1. Why?

> - The length of the response to your initial call `hide_secret("AAA..AA")` will be exactly `a + s + 1` where `a` is the number of A's (between 0 and 15) and `s` is how many secret bytes >you know so far. Why is this true only when the secret is complete?

```
def hide_secret(x):
    text = x + "this should stay secret"
    ret = encrypt_aes_ecb(text, "COOL T MAGIC KEY")
    return ret
answer = hide_secret("just listen find the >magic key")
print(answer.hex())

import string

def find_secret():
    printable = string.printable
    found = False
    secret = b''
    part = 1
    aaa = "AAAAAAAAAAAAAA"
    while not found:
        found_counter = 0
        temp_cip = hide_secret(aaa)
        temp_cip = temp_cip[:16*part]
        for i in printable + str(1):
            b_i = str.encode(i)
            temp_guess = str.encode(aaa) + secret + b_i
            #print("temp_guess")
```

David Herel / KBE Published at Dec 15, 2021 Unlisted

```
            #print("guess_ret")
            #print(guess_ret)
            #print("temp_cip")
            #print(temp_cip)
            if temp_cip == guess_ret:
                found_counter +=1
                if (not found):
                    secret = secret + b_i
                    if (len(secret)%16 == 0 and len(secret)!=0):
                        aaa ="AAAAAAAAAAAAAA"
                        part +=1
                        #print("secret: " + str(secret))
                    else:
                        aaa = aaa[:len(aaa)-1]
                        #print(aaa)
                        #print("secret: " + str(secret))
        if (found_counter <= 0):
            found = True
    return secret

print(find_secret())
```

```
45a306391112e09639cc44fa4d53c79ef86f5d44f45cc3f3f0f06e0bc4880847d38e3ebfa8b489d3e2fb3819bcb4786e04d50c322b3bf872c7d37d6f127f3a06
b'this should stay secret'
```

That's it I have succesfully decrypted ciphertext without knowing the key.