Technische Universität München
Lehrstuhl Informatik VIII
Prof. Dr.-Ing. Georg Carle
Dr. Heiko Niedermayer
Cornelius Diekmann, M.Sc.
Dr. Ralph Holz
Stefan Liebald, B.Sc.

Network Security WS14/15
Challenge 05

Task 1 Implementing the Meet-in-the-middle attack on 2DES

The keylength of DES is too short for today's computers. A naive extension would be to encrypt twice with two different keys. Let p be the plain text, and c the cipher text. Let Enc be the encryption operation in DES, and Dec the decryption operation[1]. 2DES is then: $c = Enc_{k_2}(Enc_{k_1}(p))$. Surprisingly, the strength of this encryption is not what one might expect. This is due to the following attack, called Meet-in-the-middle. This attack trades space vs. time. We are going to implement this attack, plus ask some theoretical questions.

Meet-in-the-middle The attacker needs to know one plaintext and one corresponding ciphertext (result of 2DES), and then does the following.

1. He computes and stores all $Enc_{k_i}(p)$ for all possible k_i on the plaintext p.
2. He computes $Dec_{k_j}(c)$ for all possible k_j on the ciphertext c.
3. He compares each $Dec_{k_j}(c)$: if there is a match $Enc_{k_i}(p) = Dec_{k_j}(c)$, then $Enc_{k_j}(Enc_{k_i}(p)) = c$ must hold. He knows $k = (k_1, k_2) = (k_i, k_j)$.
4. If there is more than one match, he needs more plaintext-ciphertext pairs to eliminate the wrong candidates. This will not be needed in our implementation.

Key lengths To save you computational effort, our DES keys are stupid. Usually a DES key has a length of 64 bit (56 bit effective key length), which in our implementation corresponds to a string of 8 characters (1 Byte per character). We use simple numbers as key, somewhere between "00000000" and "00999999" (leading zeros are important for correct key lengths!). Thus, the effective 2DES key length that you have to crack is severely reduced and it is more fun to work with. When first connecting to our server, you can send us a plaintext which we will encrypt using 2DES in CBC Mode and two different, randomly chosen keys. The ciphertext then will be sent back to your client.

General instructions

- You can find the client framework to use on the lecture's home page (`netsec05_client.py`).
- If you haven't done already for the previous challenge, download the `pyCrypto` library from https://pypi.python.org/pypi/pycrypto and install it following the instrucions on the bottom of the page. You can find the API of pyCrypto here, including some examples: https://www.dlitz.net/software/pycrypto/api/current/

[1] Note that decryption is just using the keys from each DES round in reverse order.

- You can test your (2)DES Implementation using the example given in the client.
- Since DES requires a block size of 8 Byte, we use simple zero padding http://en.wikipedia.org/wiki/Padding_%28cryptography%29#Zero_padding in case our plaintext isn't a multiple of that length (example again in client).

Desired protocol flow

- Client ⇒ Server: Plaintext
- Client ⇐ Server: iv, Ciphertext (both base64 encoded)
- Client ⇒ Server: Key1, Key2
- Client ⇐ Server: Result (keys correct or not)
- Client ⇒ Server: Send your tag for the database (if keys were correct)
- Whenever an error occurs the server will send you the error message instead of the usually expected next message.

New communication protocol In the old challenges we used a simple line-based protocol for exchanging messages between client and server. From this task on we switch to using dictionary to hold the values we want to send and JSON to send these dictionarys over the internet. So for example in step 3 in our protocol we would put the keys in a dictionary as follows:

dict={"Key1":"00000042","Key2":"00001337"}

We can then send this dictionary using the already given **send_json(socket,obj)** function:

send_json(socket,dict)

This will convert the dictonary to a JSON string and send it to the server, using the given socket. The server then will convert it back to a dictionary and simply read the keys:

Key1=dict["Key1"]
Key2=dict["Key2"]

So the only thing you have to do when sending the data is creating a dictionary containing the key value pairs you want to send and call **send_json(socket,obj)**.
Receiving Data works similar, simply call **recv_json(socket)**.

a) Implement the attack. Some hints:

- The plaintext must have a size of at least 10 characters
- Make sure that your keys in the dictionaries follow exactly the format of our skeleton code (e.g. "Key1" for the first key, not "key1", nor "1key").
- Make sure to use the correct IV (the server will send it in his first message along with the ciphertext, both base64 encoded).
- Only change code in the main function or in functions created by your own. Changing the code of other functions may result in errors when sending or receiving messages.

For your reference, this is our 2DES implementation at the server:

```
key1=str(random.randint(1,1000000)).zfill(8)
key2=str(random.randint(1,1000000)).zfill(8)

#add padding
if(len(plaintext) % DES.block_size!=0):
    plaintext += "0"*(DES.block_size-len(plaintext) % DES.block_size)

#compute ciphertext
iv = Random.new().read(DES.block_size)
cipher = DES.new(key1, DES.MODE_CBC, iv)
ciphertext = cipher.encrypt(plaintext)#1st DES run
cipher = DES.new(key2, DES.MODE_CBC, iv)
ciphertext = cipher.encrypt(ciphertext)#2nd DES run

#encode and send to client
message={"iv":base64.b64encode(iv).decode(),"ciphertext":base64.b64encode(ciphertext).
    decode()}
```

b) We conclude with a few theoretical questions. These will be discussed in the same exercise hour as the practical part.

1. *Derive* the *average run time* of the algorithm as described above. Assume a keylength n.
2. Assume we had used full keylengths and a message length of 64 bit. What would be the *space requirement* in the *worst case*? Give your answer in GB, rounded to the next GB. Show how you arrived at this answer!
3. Explain: would this attack also work on AES – and why?