2DES Meet-in-the-Middle Attack

Introduction

The Data Encryption Standard (DES) is a symmetric key encryption algorithm developed in the early 1970s by IBM. DES uses a 64 bit key where 8 bits are parity bits, thus making the effective key size 56 bits. Unfortunately, 56 bits is not large enough to prevent brute force attacks, and DES is considered insecure. In 1999, the Electronic Frontier Foundation (EFF) were able to brute force a key in under 24 hours. They were a little lucky, as the machine they built would have taken a little over 9 days to try every possible DES key, but 9 days is still too short to be considered secure. In the intervening 20 years, the team behind crack.sh have built a machine that can try every DES key in just 26 hours. To brute force the same key as the EFF, it would take crack.sh's machine just a little under three hours.

The initial response to DES being considered insecure, was 2DES. 2DES simply consists of encrypting the data a second time, with (ideally) a second key. The idea was that the second encryption would have a multiplicative effect on the keyspace (the set of possible keys). This would mean that the total keyspace would be $2^{56} * 2^{56}$ for a total of 2^{112} keys. However, 2DES is vulnerable to what is called the Meet-in-the-Middle (MitM) attack. The MitM attack relies on having unencrypted data (plaintext) and the resultant encrypted data (cyphertext). This is sometimes not possible, but is frequently achievable due to common strings, file headers, and chosen text attacks. MitM involves encrypting the plaintext with all the possible keys and storing the results, followed by decrypting the cyphertext and then trying to find a match in the encryptions. You can see this in set notation in Figure 1.

$$SubCipher_1 = ENC_{f_1}(k_{f_1}, P),\ \forall k_{f_1} \in K:$$

and save each $SubCipher_1$ together with corresponding k_{f_1} in a set A

$$SubCipher_1 = DEC_{b_1}(k_{b_1}, C),\ \forall k_{b_1} \in K:$$

and compare each new $SubCipher_1$ with the set A

Figure 1

When a SubCipher found via decryption is found to also be in the set A, then k_{f1} and k_{b1} are considered candidates for being a key pair and are tested against other known pieces of plaintext and ciphertext. This reduces the effective keyspace from 2^{112} to $2^{56} + 2^{56}$ or 2^{57} which is only slightly better than 2^{56}.

In this tutorial, you will learn how to implement MitM in Python and use it to decrypt an image. This should take no more than an hour.

Prerequisites

- Python 3.6.x
  - The code presented will likely work on lower versions of Python 3.x.x, but use 3.6.x for the best experience
- pyDes 2.0.1
  - View on PyPI here: https://pypi.org/project/pyDes/
- A text editor
  - I recommend Vim or Sublime Text 3, but something like Notepad or Gedit will do
  - DON'T USE A WORD PROCESSOR LIKE MICROSOFT WORD
- A solid grasp of programming and Python 3
  - Although I will be explaining code and concepts as we go along, if you don't know how to program and or Python 3 you could get quite lost
- The "animal.png.2des" file from this tutorial repository (https://github.com/SirAeroWN/two_des_meet_in_the_middle)
- A PNG file, doesn't matter what of
- A hex viewer/editor
  - The Bless hex editor should work, as will xxd on linux systems

Implementing 2DES

The first step in being able to attack 2DES with MitM is to be able to encrypt and decrypt something with 2DES and a known key pair. At this point, make sure you have Python and pyDes installed on you machine. pyDes is the library we'll be using to perform regular DES encryption and decryption. Create a new python file, perhaps name it something like mitm.py. At the top, add `from pyDes import des`. This will let us encrypt and decrypt with regular DES. Normally, encryption with pyDes is a two step process: create a cipher object based on a key, then use that object to encrypt something. However, because we're programmers and don't like repeating ourselves, we'll write a pair of functions for encryption and decryption. Add the following code to your file.

```
# encrypt given plaintext with key
def encrypt(key, plain_text):
      cipher = des(key)
      return cipher.encrypt(plain_text)

# decrypt given plaintext with key
def decrypt(key, cipher_text):
      cipher = des(key)
     return cipher.decrypt(cipher_text)
```

Listing 1

This code reduces the multi step process of encryption and decryption to a single function call with a key and some text. At this point, you may be biting at the bit to do some actual encryption and decryption, but first we've got to figure out exactly how to represent a key. Now, we know that DES takes a 64 bit (or 8 byte) key with 8 parity bits. This is equivalent to 8 values between 0 and 127 inclusive concatenated together. That's all well and good, but we need a data structure! Well, pyDes wants the key to be a byte string. But before you go reaching for the Python 3 byte string, just know that, for some reason, pyDes doesn't like the Python 3 byte string. So what's the solution? The builtin array module and the tostring method. Import the array module in you file like so:

```
import array
```

Listing 2

And set your key with the following:

```
key = array.array('B', [0, 0, 0, 0, 0, 0, 1, 56]).tostring()
```

Listing 3

YOu don't have to use the exact same numbers I did, but they must be in the range 0-127. If you pick a string and call our functions like so:

```
ciphertext = encrypt(key, your_string)
print(ciphertext)
print(decrypt(key, ciphertext)
```

Listing 4

Then you'll probably get an error because pyDes wants the text to be in the special byte string form as well. To convert a regular old string to the particular byte string pyDes wants will require two functions. The first will convert each character into its hexcode equivalent, and the second will be similar too how we created our key. Add the first function to your file:

```
def string_to_hex(s):
      return ''.join([format(ord(c), '02x') for c in s])
```

Listing 5

Breaking this function down from the inside out, we have the ord() function getting the ASCII code for a character. This is then formatted with the format string "02x" which creates a hex string that is zero padded to a length of at least two (so 15 would be 0f). We do this for every character in the string s, which is our function's sole parameter. Lastly, we use the join command to concatenate all of the strings created by the previous steps.

The second function will take the previous function's output, and produce something that pyDes can work with.

```
# convert a regular string with hex pairs into a format pyDes can
play nice with
def convert_string_to_bytes(s):
    # string needs to be an even number in length
    if len(s) % 2 != 0:
        raise ValueError
    else:
        ls = [s[i:i + 2] for i in range(0, len(s), 2)]
        return array.array('B', [int(c, 16) for c in ls]).tostring()
```

Listing 6

This function first checks to make sure that the input is of the correct length, if using the output from the previous function, this should never be a problem. The next step is to divide the input string into smaller sections, each of length 2. We use another list comprehension to convert the string hex codes into integers and then pack it all into a byte string with the array module as before. The output from this function is ready for pyDes.

Now, you might be thinking “hey, couldn’t we save some steps here? This looks redundant.” And you’d be right, but we need one of these functions later, so it’s easier to do things this way for now.

Ok, so let’s try encrypting and decrypting again. One quick note first: make sure the string you use has a length that is a multiple of 8. DES is a block cipher based algorithm and uses a block size of 8. pyDes will be quite unhappy if you give it something else. I ran the following code and got the output “b'Hello, the world'”.

```
s = convert_string_to_bytes(string_to_hex("Hello, the world"))
key = array.array('B', [0, 0, 0, 0, 0, 0, 2, 56]).tostring()
ciphertext = encrypt(key, s)
print(ciphertext)
print(decrypt(key, ciphertext))
```

Listing 7

Now that we've got single DES down, it's time to move on to 2DES. Fortunately, the lead from single DES to 2DES is quite small and we only need two more functions:

```
# perform a full 2DES encryption (not used in cracking)
def two_des_encrypt(k1, k2, plain_text):
    return encrypt(k2, encrypt(k1, plain_text))


# perform a full 2DES decryption (not used in cracking)
def two_des_decrypt(k1, k2, cipher_text):
    return decrypt(k1, decrypt(k2, cipher_text))
```

Listing 8

That’s right! It’s just as simple as calling encrypt or decrypt a second time on the output from the first call to encrypt or decrypt, respectively. If we try this out similarly to the way we tested single DES we should get a similar output, don’t forget to create a second key.

```
s = convert_string_to_bytes(string_to_hex("Hello, the world"))
key1 = array.array('B', [0, 0, 0, 0, 0, 0, 2, 56]).tostring()
key2 = array.array('B', [0, 0, 0, 0, 0, 0, 34, 122]).tostring()
ciphertext = two_des_encrypt(key1, key2, s)
print(ciphertext)
print(two_des_decrypt(key1, key2, ciphertext))
```

Listing 9

Now that we know how to perform 2DES encryption and decryption, we can move on to the *really* fun part and break 2DES.

Breaking 2DES

As discussed in the introduction, we'll be attacking 2DES using a Meet-in-the-Middle attack. This essentially consists of two brute force attacks: one encrypting the plaintext with every possible key into a set of intermediate ciphertexts, and one decrypting the ciphertext with every possible key into a set of intermediate plaintexts. This is followed by looking for any intermediate ciphertexts that match with an intermediate plaintext.

While the MitM attack is much faster than a simple brute force attack, it still has room for improvement. Fortunately for us, there are a few implementation details that we can use to speed up our attack. The first detail is that if we generate one of the intermediate sets before starting on the second, then when we generate the second, we can stop immediately upon finding a match and don't have to generate the entirety of the second intermediate set. The second is that the set generation process for the first set is easily parallelizable.

With these improvements in mind, let's run through our program's structure. First, we'll generate all of our intermediate ciphertexts in a parallel way. We'll be sure to store each intermediate cipher text with its associated key. Next, we'll generate our intermediate plaintexts. We'll do this in a serial way, checking each intermediate plaintext to see if it matches any of our intermediate cipher texts. If we get a match, we'll record the keys and declare victory.

Before we can do a whole load of encryptions and decryptions, we need a way to generate all of our keys. You could just hardcode all of the keys, but while 2^{56} may not be a large number for computers, it *is* a large number for humans. And trust me, you want a function to do the key generation for you. To do this in an efficient way, we'll use a python generator. Generators behave like a list when you're just iterating through values, but have the advantage of being lazy. That is, they don't generate a number until you ask for it. This greatly reduces the impact on memory requirement.

```
def generate_keys(num):
    for n in range(num):
        arr = [0] * 8
        powers = [7, 6, 5, 4, 3, 2, 1, 0]
        for i, p in enumerate(powers):
            arr[i] = n // (128 ** p)
            n = n - (arr[i] * (128 ** p)) if n >= (128 ** p) else n
        yield array.array('B', arr).tostring()
```

Listing 10

This function takes a number of keys to generate, and for each value from 0 to that number, creates a key. The "yield" keyword indicates that we are dealing with a generator. The

inner loop essentially converts the current key number into a base 128 number stored as a list. We again use the array module to create a byte string that pyDes will play nice with.

With that our of the way, we can finally implement our MitM function. First, we'll write a version that doesn't take advantage of parallelization so that we can be sure to understand how it works, then revise it to be parallel.

```
def mitm(nkeys, plain_text, cipher_text):
    table = {}
    # generate all the encryption
    for k in generate_keys(nkeys):
        c = encrypt(k, plain_text)
        table[c] = k

    # iterate each decryption and quit if we find a match
    for k in generate_keys(nkeys):
        p = decrypt(k, cipher_text)
        if p in table.keys():
            k1 = table[p]
            k2 = k
            print('found keys: (k1:{}, k2:{})'.format(k1, k2))
            print('apply k2 then k1 to decrypt')
            return (k1, k2)

    # if here then we didn't find anything
    print('did not find keys')
```

Listing 11

The ideas in this function should not be unfamiliar by this point, but let's walk through it anyhow. We start by creating a dictionary to hold our intermediate ciphertexts. Then we iterate through our keys which we are generating with our generate_keys function from earlier. THe generate_keys function takes a number of keys to generate, we'll use the nkeys parameter passed to the mitm function. For each key we create an intermediate ciphertext and store it in our dictionary with the key used to create it. We are storing the ciphertext as the dictionary key and the encryption key as the dictionary value in order to make lookups easier later. Once we've exhausted all our keys, we iterate through them again. This time, for each key we decrypt our ciphertext to generate an intermediate plaintext. We then try to find this intermediate plaintext in the dictionary that we created earlier. If we find a match, then we fetch the key used to generate the matching intermediate ciphertext and report that key along with the one we used to generate the intermediate plaintext. We then return from the function because we are done. If we don't get a match, then we continue on to the next key.

If you run this function on something you've encrypted with 2DES, you'll notice that the format of the printed keys is rather unhelpful. This is because Python is trying to convert the byte string that pyDes uses into something human readable. To make the keys *actually* human readable, we'll need another function:

```
# convert a bytes string into something more human readable
def int_formating(bs):
    return [int(b) for b in bs]
```

Listing 12

This function simply replaces each byte in the byte string with the corresponding integer. Call it on the keys found in the mitm function like so to get readable keys.

```
def mitm(nkeys, plain_text, cipher_text):
    table = {}
    # generate all the encryption
    for k in generate_keys(nkeys):
        c = encrypt(k, plain_text)
        table[c] = k

    # iterate each decryption and quit if we find a match
    for k in generate_keys(nkeys):
        p = decrypt(k, cipher_text)
        if p in table.keys():
            k1 = int_formating(table[p])
            k2 = int_formating(k)
            print('found keys: (k1:{}, k2:{})'.format(k1, k2))
            print('apply k2 then k1 to decrypt')
            print('use a tool such as this one
(http://des.online-domain-tools.com/) to decrypt the whole file with
the keys')
            return (k1, k2)

    # if here then we didn't find anything
    print('did not find keys')
```

Listing 13

Go ahead and try out this function. Try to keep the number of keys on the lower end so you don't have to wait too long. Here's how I ran the code:

```
s = convert_string_to_bytes(string_to_hex("Hello, the world"))
    key1 = array.array('B', [0, 0, 0, 0, 0, 0, 2, 56]).tostring()
```

```
    key2 = array.array('B', [0, 0, 0, 0, 0, 0, 34, 122]).tostring()
    ciphertext = two_des_encrypt(key1, key2, s)
    keys = mitm(127**2, s, ciphertext)
    print(two_des_decrypt(to_byte_string(keys[0]),
to_byte_string(keys[1]), ciphertext))
```

Listing 14

Be sure to test your keys! One of my keys was different, but still worked just fine:

```
found keys: (k1:[0, 0, 0, 0, 0, 0, 3, 57], k2:[0, 0, 0, 0, 0, 0, 34,
122])
apply k2 then k1 to decrypt
b'Hello, the world'
```

Listing 15

And now you've got a working Meet-in-the-Middle attack! But it's a little slow, I don't know about you, but I don't like waiting, so let's get some parallelization up in here and make our attack faster.

Parallelization

To parallelize our attack, we'll use two builtin libraries: multiprocessing and functools. Multiprocessing is the de facto parallelization library, and functools will help us make things a bit more convenient. To start, add the following imports at the top of your file:

```
from multiprocessing import Pool
from functools import partial
```

Listing 16

Next, we'll need a helper function to make the parallelization easier. Our helper function calls the encrypt function and generates a tuple for us.

```
def composed_encrypt(plain_text, key):
    return (encrypt(key, plain_text), key)
```

Listing 17

Next, we'll make some changes to our mitm function:

```
def mitm(nkeys, plain_text, cipher_text, pool=None):
    # build table in serial if we didn't get the multiprocessing pool
    if pool is None:
        table = {}
        # generate all the encryption
        for k in generate_keys(nkeys):
            c = encrypt(k, plain_text)
```

```
            table[c] = k
    else:
        table = dict(pool.map(partial(composed_encrypt, plain_text),
generate_keys(nkeys)))  # partial composes a function with standard args

    # iterate each decryption and quit if we find a match
    for k in generate_keys(nkeys):
        p = decrypt(k, cipher_text)
        if p in table.keys():
            k1 = int_formating(table[p])
            k2 = int_formating(k)
            print('found keys: (k1:{}, k2:{})'.format(k1, k2))
            print('apply k2 then k1 to decrypt')
            print('use a tool such as this one
(http://des.online-domain-tools.com/) to decrypt the whole file with the
keys')
            return (k1, k2)

    # if here then we didn't find anything
    print('did not find keys')
```

Listing 18

The first change is that we've added a keyword argument to the function parameters. The pool parameter refers to a Pool multiprocessing object which handles running functions in parallel. Next, we have an if statement which runs the mitm function as before if it doesn't get a Pool object. Lastly, we have this nice long line:

```
table = dict(pool.map(partial(composed_encrypt, plain_text),
generate_keys(nkeys)))
```

Listing 19

Let's break this down from right to left. On the far right we have our generate_keys function doing the voodoo that it does (generating keys, in case you forgot). Next, we have `partial(composed_encrypt, plain_text)`, this uses the partial function from functools to create a new function that is equivalent to the composed_encrypt function, but with the plain_text parameter already set. As we continue left, we reach `pool.map` this behaves just like the regular map function, accept it uses the power of the multiprocessing library to run in parallel. `pool.map` applies the function created by `partial(composed_encrypt, plain_text)` to each of the keys generated by generate_keys. The result is a great big iterable of tuple pairs representing intermediate ciphertexts and their associated keys. Calling dict on this iterable transforms it into a dictionary, producing the same output that we got when running the function in serial, but faster. We need to make one change to our mitm call so that we will use our parallelization:

```
with Pool() as p:
    keys = mitm(127**2, s, ciphertext, pool=p)
```

Listing 20

The `with` keyword will automatically handle cleaning up the Pool object when our code finishes.

Applying to Files

Being able to crack 2DES on strings that you've come up with is all fine and dandy, but you're far more likely to come across a file that you didn't create that's been encrypted with 2DES than a string that you did. Fortunately, if we know something about what the file is supposed to be, such as what its file type is, we can use the same technique.

For example, open up the PNG file that you found when gathering the prerequisites with a hex editor (or run `$ xxd -l 32 animal.png.2des` if you chose to use xxd). Notice that it starts with the hex values 8950 4E47 0D0A 1A0A? That's the header for PNG files, so they all start with the exact same hex values. Now open up the animal.png.2des file provided in this repository with a hex editor (or run `$ xxd -l 32 animal.png.2des` if you chose to use xxd). This is a PNG file too, but it's encrypted and as such you'll notice that the first 8 bytes aren't the same as the unencrypted bytes we saw earlier. Because it's a PNG file though, we *know* that the first 8 bytes *must* be 8950 4E47 0D0A 1A0A when unencrypted. Thus, we have our plaintext (the unencrypted header) and our corresponding ciphertext (the encrypted header).

Since our plaintext and ciphertext are already in hexcode format we just need to call a single function on them to get a format that pyDes will work with:

```
PLAIN_TEXT = '89504E470D0A1A0A'
CIPHER_TEXT = '89A3F4E3A99337A4'
plain = convert_string_to_bytes(PLAIN_TEXT)
cipher = convert_string_to_bytes(CIPHER_TEXT)
```

Listing 21

Passing plain and cipher into our parallel mitm function will give us the keys:

```
with Pool() as p:
    keys = mitm(127**2, plain, cipher, p)
```

Listing 22

Great! We have keys, how do we decrypt the entire file? Well, rather than dealing with trying to get our files into a format that pyDes likes, we can use an online tool. First, we need to get our keys into a format that the tool likes with a short conversion function:

```
def hex_formating(bs):
    return ''.join([format(b, '02x') for b in bs])
```

Listing 23

Print the results of calling this function on our keys by adding the following line after our mitm function call:

```
print(f"k1:{hex_formating(keys[0])}, k2{hex_formating(keys[1])}")
```

Listing 24

Then, head over to http://des.online-domain-tools.com/ and choose “file” for input type, upload animal.png.2des using the browse button, make sure that DES is selected as the function, the mode should be ECB, paste k1 in the key field, select the hex radio button, and click the “> Decrypt!” button. This will apply the first round of DES decryption to our file. You can download the output by clicking the link in the red box shown in Figure 2.

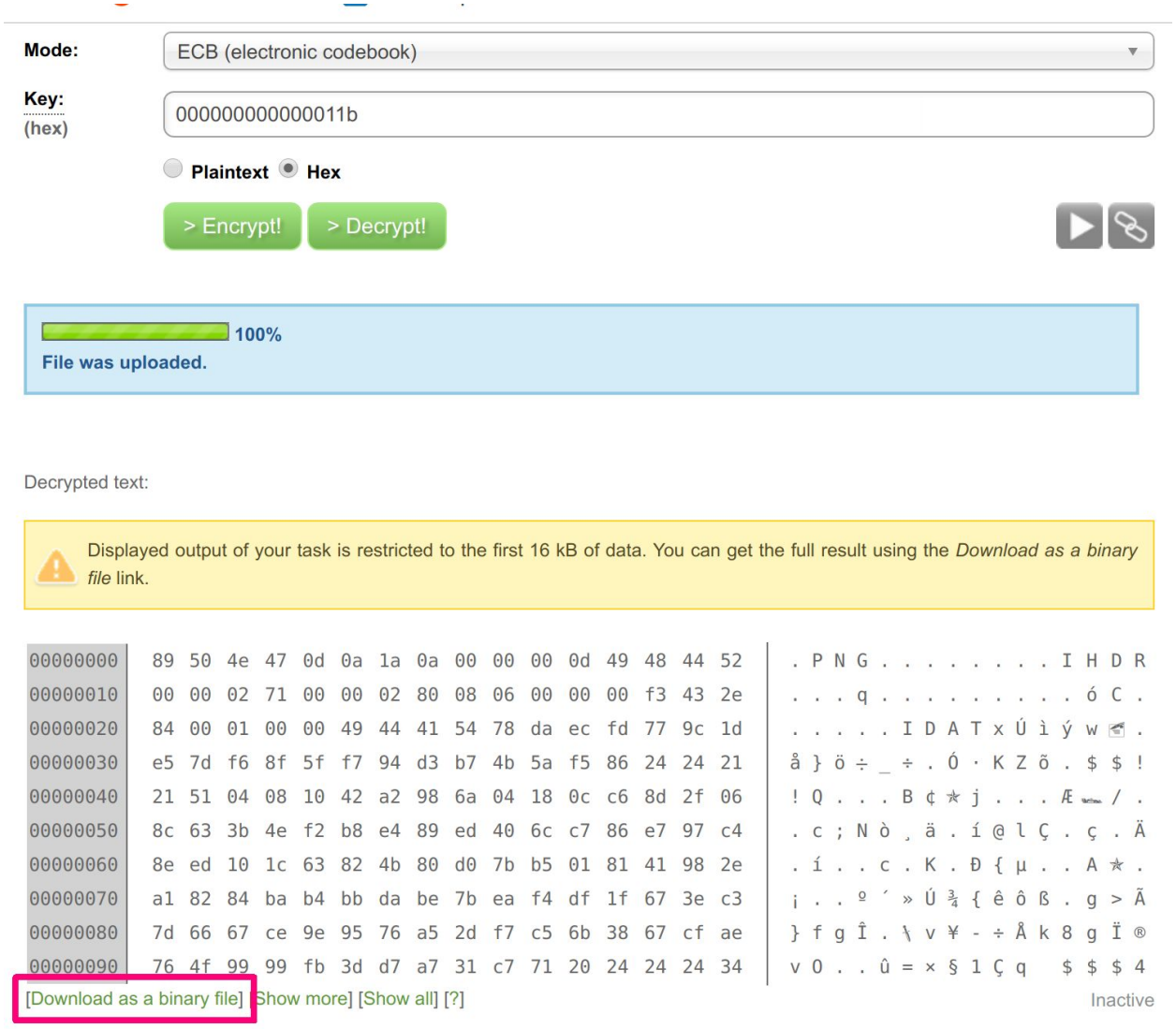

Figure 2

Repeat the decrypting process again, this time using the file you just downloaded and k2. The file downloaded after this round of decryption should be fully decrypted. Change the file

extension to png and open it up. What animal is it? Is it the best animal? (the answer is yes, of course).

Conclusion

And thus, we are done. We have shown in fairly short order that simply encrypting something a second time does not add sufficient protections. Additionally, because we abstracted the actual method of encryption and decryption as well as key generation away from the mitm function, our code could easily be converted into an attack on a different type of symmetric encryption, yay for abstractions!

To see a fully assembled version of the code discussed here, clone or download the github repo: https://github.com/SirAeroWN/two_des_meet_in_the_middle.

References

https://en.wikipedia.org/wiki/Data_Encryption_Standard
https://crack.sh/
https://en.wikipedia.org/wiki/Meet-in-the-middle_attack
http://des.online-domain-tools.com/