Attacking weak symmetric ciphers

Tufa Alexandru Daniel 1628927

27 October 2017

1 Introduction

The **Data Encryption Standard** or simply **DES** is a symmetric-key algorithm used for the encryption of electronic data. Developed in the early 1970s by IBM it was highly influential in the advancement of modern cryptography although now is considered insecure mainly because the 56-bit key size is too small. A public break was done in January 1999 by distributed.net [1] and the Electronic Frontier Foundation in less than 24 hours,while this algorithm was considered breakable by governments already in the 80s.

2 Description

The goal of this paper is to understand how to encrypt and decrypt using a library and how to break a weak symmetric cipher, in our example DES, by using a brute force method. We'll be using C as our programming language and *rpc/des_crypt* [2] as our DES library that will allow us to encrypt and decrypt using Cipher Block Chaining mode of operation. Initially we will test the *cbc_crypt* function on messages with length ranging from 1 to 99 to better understand how it works and afterwards we'll set up our brute force algorithm. We'll be using 2 main files: *crack_des_single.c* for testing encryption/decryption and doing our brute force algorithm on a single thread and *crack_des.c* for testing our brute force algorithm using 2 threads.

3 Implementation

3.1 Encryption/Decryption

Given that the offered function by the library needs some preparation before using it we'll create our own encryption and decryption functions that will simplify our code.For simplicity we'll see only the encryption function given that the decryption changes only for one constant.
We will be passed the key, the input and the length. The key will be passed as an array of 8 bytes where we'll set the least significant bit of every byte as the odd parity bit using *des_setparity*. It's important to note that even if the key is 64 bits only 56 will be really used by DES. This is a crucial fact that we'll be using in our brute force algorithm. Without losing of generality we initialize the initialization vector to "87654321" and we allocate a new portion of memory that will be our result. We need the length to be passed because the input given to us is a padded one for that is the request of the *cbc_crypt* function, a length multiple of 8. To decrypt we pass the *DES_DECRYPT* constant instead of the *DES_ENCRYPT* one with the same initialization vector.
As you can see in the *test_encryption_decryption-hw1-1628927.txt* we pass all tests for messages long from 1 to 99 bytes showing the random plaintext and key used and the ciphertext. Plaintext and ciphertext are represented using hexadecimal notation while the key is shown using a binary one where X stands for the parity bit which is not used as part of the key so we don't care about it.

3.2 Brute force attack

The algorithm behind the brute force attack mainly uses 8 cycles where in every cycle we assign every byte of the key to a new value. Given that we don't care about the least significat bit we increment by 2 every time so at the end we'll be doing 2^{54} iterations. We'll be measuring our tests gradually using the same message *AttackAtDawn* and blocking every time fewer bytes of the key. Two keys will be used for testing: a random generated one and the key with all 1s that in our case will take the longest to figure out. We will show results for monothread and 2-thread runs and even if in the beginning we will have promising results we must not fall in the trap that using a large amount of threads will solve our problem easier. It all depends on the number of independent cores so that we can divide our space of keys evenly. The number of independent cores in commercial use in 2017 ranges from 1 to 16 (AMD Threadripper) and even with a machine that uses 100

cores our key space would only decrease to 2^{50} that is still unfeasible for a commercial machine. It's not even advised to create too many threads that are going to interleave because the overhead and time spent in kernel mode for interleaving would worsen our analysis.

3.2.1 One free byte

As a measure of time we'll be using our function *get_cpu_time* that uses the *clock* primitive of Linux to obtain the number of clocks and divide by the constant *CLOCKS_PER_SEC*.

Single thread,random key: 0.000026s
Single thread,1s key: 0.000034s
2-threads,random key: 0.000030s
2-threads,1s key: 0.000040ss

3.2.2 Two free bytes

Single thread,random key: 0.002848s
Single thread,1s key: 0.004080s
2-threads,random key: 0.003240s
2-threads,1s key: 0.002664s

3.2.3 Three free bytes

Single thread,random key: 0.301701s
Single thread,1s key: 0.402128s
2-threads,random key: 0.136853s
2-threads,1s key: 0.190089s

Here is where things start becoming more interesting. As we can see from our results initially using threads was not a good idea because the overhead of creating them made our results worse but now we can see what we were expecting, that the time needed to reach the end of the key space is half.

3.2.4 Four free bytes

Single thread,random key: 40.248386s
Single thread,1s key: 49.459882s
2-threads,random key: 4.188787s
2-threads,1s key:13.048528s

As you can see if the attacker knows half of the password finding a match is truly easy. From now onwards things will become much harder for an attacker that uses a commercial machine but we'll see that in some cases he can still do a successful attack.

3.2.5 Other considerations

If we fix key[2] to 0x02 we see that the time taken in both single thread and 2-thread doubles roughly so we can estimate that the time needed to finish the computation would be 106 minutes for the single thread case and 60 for 2-thread one to find the key after blocking 3 bytes.
What if a certain website still uses DES and requests an 8 byte password that will be used as a key? In that case the attacker can be almost certain that the key space will be a lot smaller and that each byte could be between 0x20 and 0x80 that are the values in the ASCII table that are used as characters. In this case for a simple password like *01234567*, the time needed to find the five free bytes would be only 18.565726 seconds and that is a very important result because it reduces the key space drastically! Given our previous analysis for finding the key by blocking 2 bytes the algorithm would need 9 days, by blocking 1 byte it would need 40 months and to find the full key it would need 432 years.

References

[1] The internet's first general-purpose distributed computing project. http://www.distributed.net/.

[2] des_crypt linux man page. https://linux.die.net/man/3/des_crypt.