Attacking DES on a what3words address

zrlr73

1 Initial Thoughts & Information

As soon as I saw that ECB mode had been used, my first thought was of the fact that each 64-bit block is encrypted totally independently - that is, every 64 bits (or 8 characters) can be analysed completely separately. For a brute-force attack, this means that an attacker only needs to find the right 8-character plaintext for each consecutive ciphertext block and concatenate them, rather than guess the whole input in one go.

The provision of an exe file containing the encryption key did provide the opportunity to decompile that file, but I thought that would compromise the spirit of the coursework so decided to focus on other options first.

1.1 My Computer

The computer being used for this coursework was my desktop, which has an 8-core / 16-thread AMD Ryzen 7 1700 (overclocked to 3.7GHz), 16GB of RAM and plenty of free storage space. The single-core performance of the 1700 is very good, but its real strength is its multicore throughput, so I planned to leverage this.

2 Initial attempts

My first idea was to simply bruteforce every possible 64-bit string to see how long it would take: after a bit of experimentation I found that it would take time on the order of 34 years. Since this would mean submitting the coursework quite late, I decided that idea might be a bit impractical. I also briefly researched more intelligent methods like differential cryptanalysis, but found that even with chosen plaintexts there would not be any particular performance improvement when compared with the advantage I had as a result of knowing the structure of the input.

3 Dictionary attack

3.1 Initial attack

The clear next step was to try a dictionary attack, generating possible what3words addresses from a dictionary of ~370,000 English words. I initially started generating every possible what3words address before realising that this was a bit of a Bad Idea™ and would take an unreasonable quantity of time. I then had the realisation that I could limit the generated addresses to be exactly 16 characters long, since this was the length of the cipher text and no mention of padding had been made in the coursework brief. I also found anecdotally that what3words seemed to only use words comprising 4 or more characters and was therefore able to restrict the words to only those which were 4, 5 or 6 characters long. All of these adjustments did improve things, but they were still taking an unreasonable quantity of time to generate. The memory usage also was quite excessive throughout this stage, thanks to the large sets of potential addresses.

I then had the idea of generating first and second halves separately - this would remove a lot of redundancy because, for example, four different 16-character addresses could be summarised by five 8-character ones:

`drops.rings.land  drops.rinks.tall  drops.ribs.desks  drops.river.also`

`drops.ri  ngs.land  nks.tall  bs.desks  ver.also`

saving 24 characters in this very small example. This meant that the set of items to check would be significantly smaller, especially at large scale, where thousands of full addresses would match to a single half. With this efficiency improvement in place it only took a couple of minutes to generate every possible half of a 16-character what3words address.

In this way I generated 36,098,043 first halves and 38,196,414 second halves. I then wrote code to use `encrypt.exe` to try all first halves, and found that it would run in a timeframe that wasn't totally unreasonable, so used the Python `multiprocessing` module to parallelise it on all 16 threads. Following a series of runs of 50,000 first-halves I found that with this method I was able to test one half-address roughly every 0.027 seconds on average, putting me at roughly 550 hours (or around two weeks) to try every single possibility. Clearly this was not ideal, but I ran batches on the first half for a day or so, storing a list of ~1.8m checked first halves to disk, until I had a new idea:

3.2 Using a smaller dictionary

I noticed that the large dictionary contained an overwhelming number of obscure words like 'hokes' and 'reina', and did a small amount of research to see if there were any smaller word lists that contained every 'commonly-used' (ie, not obscure) word. I couldn't find any, but I did find a list of the most frequent 10,000 English words based on a dataset from Google. I used this to generate a set of more likely what3words address halves.

Running this gave me only 1,826,012 first halves and 1,695,629 second halves. Trying every first half after subtracting those that had already been tested took roughly 10 hours in total, and eventually yielded the first half!

`tile.bil`

I then adjusted the code to generate a list of second halves with the second word filtered to only include those that began with `bil`. This gave me 4225 possible second halves, which were exhausted in 90 seconds with the answer:

`ls.print`

4 Result

In summary, I found the following address:

`tile.bills.print`

which I found to point to an unassuming car park in Philadelphia, USA. I must admit, this was initially a little underwhelming because I'd been expecting it to point to the top-secret location of a French nuclear bunker or a landmark or something. However, searching for the landscaping business at that location led me to stories I had seen before about Donald Trump's ill-fated mid-election press conference where Four Seasons Total Landscaping had been booked instead of Four Seasons Hotel. Touché!

5 Reflections

Alternative options that I had considered include using the what3words AutoSuggest API. I also noticed from profiling that encrypt.exe wasn't making great use of my CPU, and seemed to include a lot of overheads with each run, stopping it from performing the actual DES computations very efficiently. Since completing, I realised that I could have 'batched' several concatenated plaintexts into this file at once, rather than testing one at a time, to reduce the impacts of these overheads. This strategy would probably have allowed me to complete each step several times more quickly, so if attempting again I would apply it.