B5 - Cryptography

B-SEC-500

Crypto-Security

Cryptography, Cryptanalysis & Cryptology

1.04

{EPITECH.}

Crypto-Security

binary name: challengeXX
repository name: SEC_crypto_$ACADEMICYEAR
repository rights: ramassage-tek
language: C, C++, Python
compilation: via Makefile, including re, clean and fclean rules

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

This project is designed to teach you the basics of real life cryptography implementations and attacks. It is greatly inspired from cryptopals (or Matasano) challenges, huge shout-out for them. There is no cryptography nor advanced math background needed to complete the challenges.

It turns out that many modern crypto attacks don't involve much hard math.

The project is composed of 14 exercises that get gradually harder.
You can choose among C, C++ and Python as a language to complete the exercises.
The binary names should respect the following naming pattern `challenge{id:02d}` where `id` is the challenge number e.g. `challenge07` for challenge 7.

Be sure you to finish every exercise neatly before moving on to the next, you will reuse them all later.

ALWAYS be careful of the user input. Most of today's vulnerabilities occur when a developer did not sanitize properly user-controlled data.

ALWAYS operate on raw bytes rather than on encoded strings. Use hex and base64 only for pretty printing.

ALWAYS print your results followed by a newline.

For ALL challenges that requires you to print hexadecimal, use capital letters only, i.e. 'COFFEE' instead of 'c0ffee'.

For ALL challenges, an empty input, whether its from a file or a server, should raise an error.

For C/C++, the only allowed library is openssl (libssl-dev).
For Python, you can only use the Crypto module (pycrypto).
For all languages, the only crypto primitive you are allowed to use is the AES encryption/decryption in ECB mode (CBC == cheating).

If you choose to use Python (recommended), the only version authorized is python3.6. The only external python module you can use are pycrypto v2.6.1 and requests v2.19.1. For those who previously pushed binaries, please delete them and keep your source files only. The presence of a Makefile is still mandatory (copy your `sourceXX.py` to `challengeXX`)

If you choose to use C/C++ (h4rdc0r3), please push the openssl library and recompile it with your project. The testing environment WILL NOT have the libssl-dev library installed.

Challenge 1 - Convert hex to base64

Make a program that converts the content of the file given as first argument from hexadecimal to base64 and prints it on the standard output.

Terminal

```
~/B-SEC-500> ls
Makefile  hex_to_base64.py  input01.txt
~/B-SEC-500> make
~/B-SEC-500> ls
Makefile  challenge01  input01.txt  hex_to_base64.py
~/B-SEC-500> cat input01.txt
4B
~/B-SEC-500> ./challenge01 input01.txt
Sw==
```

`123` is not a valid hexadecimal notation, neither is `A`.

Challenge 2 - Fixed XOR

Write a program that takes a file containing two hexadecimal encoded buffers of equal (and only equal) length separated by a newline and produces their XOR combination.

Terminal

```
~/B-SEC-500> cat input02.txt
5374616C6C6D616E
426C61636B486174
~/B-SEC-500> ./challenge02 input02.txt
1118000F0725001A
```

Just in case you wonder, XOR stands for *eXclusive OR*.

CHALLENGE 3 - SINGLE-BYTE XOR CIPHER

Code a program that finds the single byte key that has been XORed against a given string.
The input is a hexadecimal encoded file.
The byte used for the XOR operation should be printed on the standard output in capital hexadecimal.

```
Terminal
~/B-SEC-500> cat input03.txt
0430272762112D2436352330277862043027272 62D2F62232C2662012D2D32273023362B2D2C
~/B-SEC-500> ./challenge03 input03.txt
42
```

How can a program tell if a piece of text is in English? Think Etaoin Shrdlu.

CHALLENGE 4 - DETECT SINGLE-BYTE XOR

Given a file containing one hex-encoded string per line, among which one has been encrypted by a single-byte XOR, write a program that detects this string, decrypts it (in the 3rd challenge fashion) and prints the key used to encrypt it in hexadecimal, prefixed with the line number.

```
Terminal
~/B-SEC-500> head -n 5 input04.txt
BD969CE5A2B882F59391A59EF9FBEE94E7BA8C92E587B3F6E389E08B89B6B694BC809CFDE5F2
10110D70125E48035E1C0355734C6B7C75096646475E1754085C4E156B7540446F706D78531A
DFCEC0CF8096C282D5F6F1CDDB9B89FDC1FFD2C5EA8988D48FD2E09FF9CA96F891C9D696D9D9
6D76662F487334375F5B402F584A3F6C217D535A27214941256A3257736E36463A6E5163723E
9D86EADDC8E3EED9ED88C5CCFADC9E92C4C0C381DDF785C6FCC2E1F88290D28FE3D8ECC7DC93
~/B-SEC-500> ./challenge04 input04.txt
440 33
```

Challenge 5 - Implement repeating-key XOR

In repeating-key XOR, a key is XORed sequentially (byte by byte) to the plaintext, repeating from the first index when the key is exhausted.

Exemple with 'DIFFIE-HELLMAN' to be encrypted with 'RSA' as key.

key:	R	S	A	R	S	A	R	S	A	R	S	A	R	S
plaintext:	D	I	F	F	I	E	-	H	E	L	L	M	A	N
ciphertext:	16	1A	07	14	1A	04	7F	1B	04	1E	1F	0C	13	1D

Develop a program that :

1. takes a filename as sole argument,
2. reads its first line, this will be the hex-encoded key,
3. encrypts the rest of the hex-encoded file with repeating key XOR using that key,
4. prints the result in hex format followed by a newline.

Terminal

```
~/B-SEC-500> cat -e input05.txt
474E55$
54686520726F6C65206F66206120756E697665727369747920697320612070C61636520746F206
66F737465722064656261746520616E6420746F206861766520696E7465726573746964696E67206469
7363757373696F6E732E20416E642074686520726F6C65206F662061206D616A6F7220756E69766
5727369747920697320746F206861766520706172746963756C61726C7920696E74657265737469
6E672064697363757373696F6E732E$
~/B-SEC-500> ./challenge05 input05.txt | cat -e
132630673C3A2B2B75282875266E20292723223C262E3A2C672726672F75372234242B753321752
12126332B27672A30252F21226E34292A753321752F2F23226E3C293A30352B2633273B206E312E
3D36323D262E213B346075062031673A3D226E27282230672133672F752A2F3F283C7532203C312
B273427213E6E3C346E21286E3D263830673E34353A3C243B39263C393E6E3C293A30352B263327
3B206E312E3D36323D262E213B3460$
```

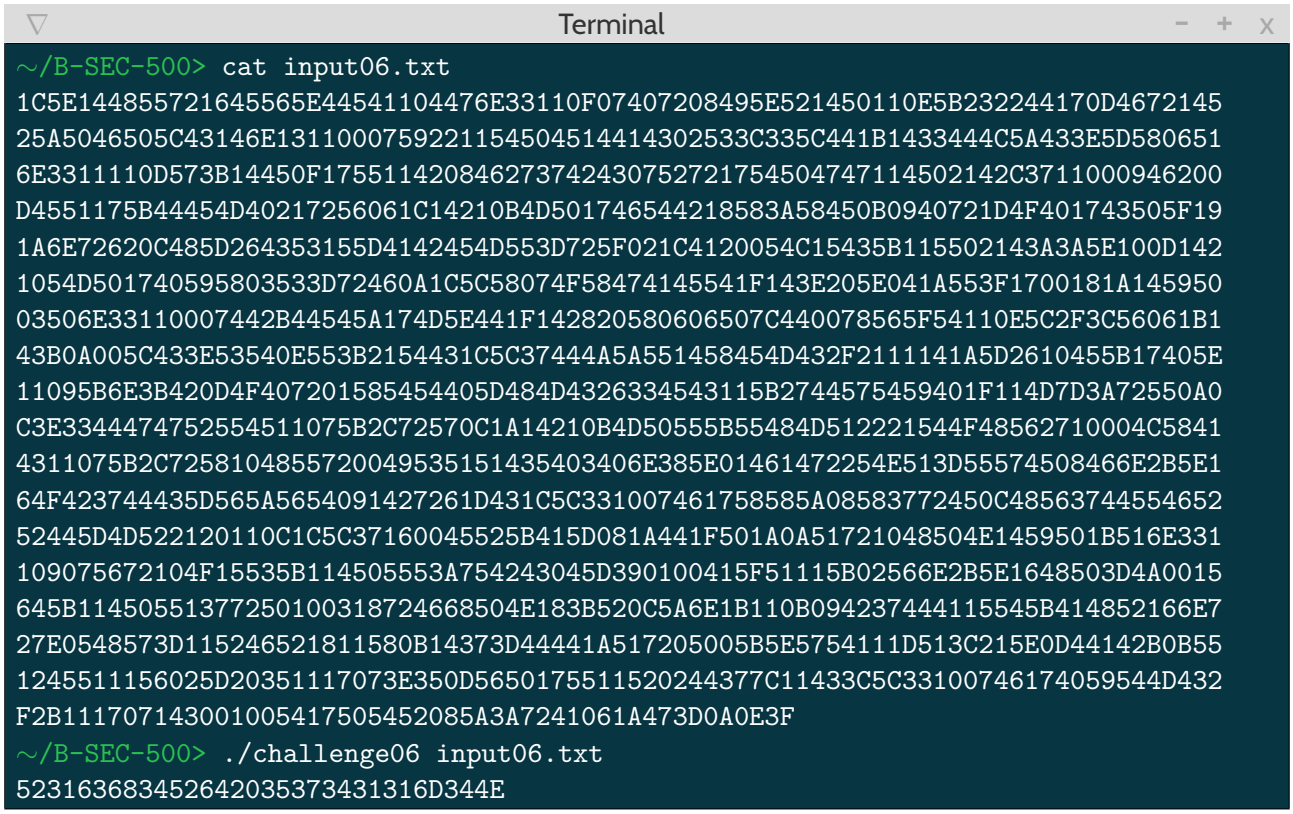
While on its own, XOR encryption is really weak, the operation is very common as a component in more complex ciphers.
When the key is the same length as the plaintext, you obtain a one-time pad (OTP), which cannot be cracked as long as the key is truly random and used only once.

Challenge 6 - Break repeating-key XOR

You are to decrypt a hexadecimal version of a text encrypted using repeating-key XOR, contained in a file given as argument.
How?

1. Have a function that counts the number of differing bits in two strings.
   This is called the Hamming distance. For example, the Hamming distance between 'Hello' and 'World' is only 14.
2. Have a function that can break single-byte XOR (you already have that, don't you?).
3. Let KEYSIZE be the probable length of the key.
   For each KEYSIZE (say 5 to 40), find the Hamming distance between the two (or four) first blocks of KEYSIZE length.
   Normalize the result by dividing by KEYSIZE.
   The one with the smallest normalized Hamming distance is probably the length of the key.
   Just in case, you can proceed with the smallest couple of KEYSIZE values.
4. Now that you have the probable KEYSIZE, break the given ciphertext into blocks of KEYSIZE length.
5. Transpose the blocks, i.e. the first block will be composed of the first byte of each block, the second block will be composed of the second byte of each block, etc.
6. Solve each block as if it was a single-byte XOR.

You now have all the steps needed to reconstruct the original key.
Write a program that takes a hexadecimal encoded file as input and prints out, in hexadecimal, the key used to encrypt it.


Terminal
~/B-SEC-500> cat input06.txt
1C5E1448557216455656E44541104476E33110F07407208495E521450110E5B232244170D4672145
25A5046505C43146E131100075922115450451441430253 3C335C441B1433444C5A433E5D580651
6E3311110D573B14450F17551142084627374243075272175450474711450214 2C3711000946200
D4551175B44454D40217256061C14210B4D501746544218583A58450B0940721D4F401743505F19
1A6E72620C485D264353155D4142454D553D725F021C4120054C15435B115502143A3A5E100D142
1054D501740595803533D72460A1C5C58074F58474145541F143E205E041A553F1700181A145950
03506E33110007442B44545A174D5E441F142820580606507C440078565F54110E5C2F3C56061B1
43B0A005C433E53540E553B2154431C5C37444A5A551458454D432F2111141A5D2610455B17405E
11095B6E3B420D4F407201585454405D484D4326334543115B2744575459401F114D7D3A72550A0
C3E33444747525545 11075B2C72570C1A14210B4D50555B55484D512221544F48562710004C5841
4311075B2C7258104855720049535151435403406E385E01461472254E513D55574508466E2B5E1
64F423744435D565A5654091427261D431C5C3310074617585 85A08583772450C48563744554652
52445D4D522120110C1C5C37160045525B415D081A441F501A0A51721048504E1459501B516E331
109075672104F15535B114505553A754243045D390100415F51115B02566E2B5E1648503D4A0015
645B1145055137725010031872466850 4E183B520C5A6E1B110B094237444115545B414852166E7
27E0548573D115246521811580B14373D44441A517205005B5E5754111D513C215E0D44142B0B55
1245511156025D20351117073E350D5650175511520244377C11433C5C3310074617405 9544D432
F2B11170714300100541750545 2085A3A7241061A473D0A0E3F
~/B-SEC-500> ./challenge06 input06.txt
52316368345264203537343 1316D344E

Challenge 7 - AES in ECB mode

The Advanced Encryption Standard, or AES, is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.
It uses the Rijndael symmetric key encryption algorithm with a fixed block size of 128 bits and a key size of 128, 192, or 256 bits. In symmetric key encryption, a unique key is used for both encryption and decryption of messages.

For your own information, have a look at the differences between symmetric and asymmetric encryption, with a particular attention for when, and why both are used.

A block cipher, such as Rijndael, transforms a fixed-sized block (usually 8 or 16 bytes) of plaintext into ciphertext. But we almost never want to transform a single block; we encrypt irregularly-sized messages.

One way we account for irregularly-sized messages is by padding, creating a plaintext that is an even multiple of the block size. The most popular padding scheme is called PKCS#7.

Write two functions:

- one that pads any block to a specific block length, by appending the number of bytes of padding to the end of the block,
- one that unpads any padded block.

For example, 'Rijndael' padded to 10 gives 'Rijndael\x02\x02'.

Then code a program that decrypts a file using AES in ECB (*Electronic Codebook*) mode using a provided key, printing the unpadded result in base64. In ECB, blocks are encrypted one by one using the same key e.g. Two same blocks of plaintext will always produced the same ciphertext blocks.
To achieve that, use the ECB primitive from the library allowed for your language.

The base64 encoded content of `input07.txt` has been encrypted with AES-128 in ECB mode under the key `EPITECHFORTHEWIN` (16 bytes long, one of the AES key lengths, which is pretty convenient).
Decrypt it.

The first line of the file given as argument is the key, hex-encoded, the rest of the file is the content to be decrypted, base64-encoded.

Terminal

```
~/B-SEC-500> cat -e input07.txt
45504954454348464F5254484557494E$
H6SOU80O6n/H6iisc5wwqYb7Cx/lIze0vE2u8iQfZC/Z+6NS/1rhFK7IOFQr+btj17ua0qcANrJpfBEL
Isz1E8dmYZScFNW9RcbIyixSXtSYIxT3bZ+uU34N3MKfkTKml0ovMbx6P3/Q4UhaH+dRWqsANQdh4OxI
PPqbd+vW+fnUaLUZVJaO3Z5K48KBn+bEZmHydZEYTnDGyXl7T4l95OfA0cX3u6IiyW1z54H4xte5EufR
shuMO3uD/EZiNW0jIZ8gPoIP6I2uHcT3Ed2q87RrjHA3sYehUCfIRQy8SxruTUFhSwmnKjbmO5EMonqR
d1HC8NLz5Eq8TQtPz19yMs1FTf/G4jdvzsOFbNw7lHneRwA1MS3I+on6mIVozAG2fizLkIetjhr3ZasG
gyYOstocALYHhp62XnVUpQeQ2f5mnUdZ2uYYcSS6cQ0zF1hU1nQWPHzEHMMdWmL1WXIRKkjoOGXJRpKO
18iDLboKE2m+tPw+BIQSCs5ia1Pmw1iLDtFtah5SEOAESIO4MDVaAMF2GXemnauNX5KrjWByQ87KcoM1
N1A6MvaglvbAied05sIJV0RjkETk8X8ZiaSZ6e0bekBWqxxBn8Gh0tMdVHkRdPDEKQjoSnhtmmstmwYu
eYtWPICAWuHUZ9my5mZHNWJKRaYK6G0dgC789c1YmvU=$
~/B-SEC-500> ./challenge07 input07.txt | cat -e
U28gaW1hZ2luZSB3aGF0IGl0IHdvdWxkIGJlIGxpa2UgaWYgcmVjaXBlcyB3ZXJlIHBhY2thZ2VkIGlu
c2lkZSBibGFjawpib3hlcy4gWW91IGNvdWxkbid0IHN1ZSB3aGF0IGluZ3JlZGllbnRzIHRoZXkncmUg
dXNpbmcsIGxldCBhbG9uZSBjaGFuZ2UKdGhlbSwgYW5kIGltYWdpbmUgaWYgeW91IG1hZGUgYSBjb3B5
IGZvciBhIGZyaWVuZCwgdGhleSB3b3VsZCBjYWxsIHlvdQphIHBpcmF0ZSBhbmQgdHJ5IHRvIHB1dCB5
b3UgaW4gcHJpc29uIGZvciB5ZWFycy4gVGhhdCB3b3JsZCB3b3VsZCBjcmVhdGUKdHJlbWVuZG91cyBv
dXRyYWdlIGZyb20gYWxsIHRoZSBwZW9wbGUgd2hvIGFyZSB1c2VkIHRvIHNoYXJpbmcgcmVjaXBlcy4K
QnV0IHRoYXQgaXMgZXhhY3RseSB3aGF0IHRoZSB3b3JsZCBvZiBwcm9wcmlldGFyeSBzb2Z0d2FyZSBp
cyBsaWtlLiBBCndvcmxkIGluIHdoaWNoIGNvbW1vbiBkZWNlbmN5IHRvd2FyZHMgb3RoZXIgcGVvcGxl
IGlzIHByb2hpYml0ZWQgb3IKcHJldmVudGVkLgo=$
```

Challenge 8 - Detect AES in ECB mode

The file given as parameter contains a bunch of base64-encoded strings.
One of them is an ECB-encrypted ciphertext.
Write a program able to detect it and that outputs the corresponding line number.

Terminal

```
~/B-SEC-500> cat input08.txt
hbF6M/IBmZXZ11H5QHHNaIOQU+swH/GfXZcZuPWuxIG7wXhCl3mraeckbFb/cL2+
NXQMSbuXH11s301Nj/4jWZqW9lgeGNnUVyVig2TFTblZYNlCOElHnue+xJmzYVvT
SaWpFf9qA9MC7JzSTS6a8VJiPebnoRhrexMBErFWjMkgO5IIYjznPSbzsU9M9hLN
HY2UWi8+lCIzDEDo4BKIRubNqJR/Q7rxmAxo5ltE+pfW86clf+ZdswUyzDWU3Jho
WAkKa2HAKOtO5rDdisKdO+m8h/VbSqDRl6PrB/Jb5HkJ/GihPTjhuhBnnUgl9iro
i2gIW3AKJqDLhIedWXOsuhNSIEC5cAOa08lOb+OxCuCiJgQcf1xub0lX3d+XClj4
6JbhJTtlKMA3Qge6hFB/25vgh+lbEn+vCHCJWGq81cu4RjZbtArMO0fkoM4+nkq3
cC6otLEabpae7nAlsZlv2azADIbXyxDnIWTQG6TgotNIyYtgOPGo9/53/tNPVqMY
Y/nFx1HNuGOAWZgnRqe8UvVozBMQ6jbeYuMIGbH1XMFVLXxkSsVT9aoRCpV+EwIP
B/XgY+esHNt+VfkI1XUMBVIQUv+JDKW4dNB61zm9IYIGmTZtM5BBHQB3dz8EZSkr
j9xp7vDx4wZ7kROmsJuvmlQaRkTeTmItqMVHVm/gddRb6dhz774r4ijJm2mWghDX
JiM350SPy91MYWQvHSPALQ1eBzbmE5D5o0mS60tEKeVfpoDBuDrilBV8Ruo/RJI8
IceyNglirWM+FmYV6Hv6V5Ed1saqOc14k+5esjCy8cxGqFBlZEqBLfyP8lb7nm4y
kH2G2PJq9Ux6a2KnS5zHRE0s+49dLqd4EibCr8aEGdYlXVxIdoIxEga52tD1pt9O
h1VLmXXAbZ43CGOBqn+kGodVS5l1wG2eNwhtAap/pBrE1kBC4kmaxN5VGr1d21wS
Ld4cwV+udcEgdMS+5ezIOvUgSrWlspb8R+U9bp9Rbftz3cmNbHgIu0h76nrQsmLk
t//1p9OCr+NxFCy/CxC4VGhkiOeov8JI1vVPrLuu5lwOvlu7TVT8QrzjJowYs3yW
xsNVATv/PU3axnGIBPqzMAkWl1XOaVVjfuyN+IxVrUoeRLNDytYiV6XX3yZ/g7O2
PNJPtniImCT1cmChNlx70OBol7srtzAqnAw/AqDTtiDhYE4ouriQ5kNLZ8jrRj+4
yXO/+Zc+ImjRGO5dnn2PflkTBaMijEKhTYL2zWsQ3Rn2GfHQZzbK9vZx4YecDEjn
~/B-SEC-500> ./challenge08 input08.txt
15
```

Remember that the problem with ECB is that it is stateless and deterministic i.e. the same 16 byte plaintext block will always produce the same 16 byte ciphertext.

Challenge 9 - Implement CBC mode

In CBC (*Cypher Block Chaining*) mode, each ciphertext block is added (XORed) to the next plaintext block before being encrypted with the cipher function. The first plaintext block, which has no previous ciphertext block, is added to an initialization vector (*IV*) that plays this role.

Implement CBC mode using the ECB function you wrote earlier, using the XOR function you made to combine plaintext and ciphertext blocks.

Make a program that takes a single file as input. Its first line contains the key you will use to decrypt the ciphertext, in hexadecimal, the second line contains the IV you will use to decrypt the ciphertext, also in hexadecimal, and the third line is the ciphertext to decrypt, in base64.

Make this program decrypt the ciphertext using the provided key and IV, and print on the standard output the obtained plaintext in base64, followed by a newline.

Terminal

```
~/B-SEC-500> fold -w 80 input09.txt | head -n 15
455049544543484 64F5254484557494E
00000000000000000000000000000000
KGhWjZtE3OzS7sPuHmr622tVBVvPiPR74BrccguSs1DFlgeKii2mY82yg/FgJ6510S/P2F/6axZWHbU6
j21jxkJTcjVWO/pRVv+fctoXnSJ6lWu57B2mJSPJQBKsIXiKvTW5gZ22Uq0+oDA5K0YIMaIuWXXHCCG0
G0M6pRjqe66TJLdV3fxVlJ1kl6Um3X8p3ylIEAzwddbPmD7DZHMQFhhtVTNCA4QKtW/CF42V8vQqo6AQ
jiyMAwINxHLWDZk92L7O6tMo+3pF/ymySsCk7XEq7uYmrrAbE7FSUg+VNUTuTdyZZn4BC6q9ByqfdThu
oh6SbTTydwwaITXxWppdS8/dp41pFoOipdPRlK3ydqnERYhtrAq1KxFAfexINHG439RD6DZ4FSbHS68t
157UTNEf2rZ4SPkJEvs8qXyEWW93KYZFPBAUm0bMRCwiHLrBpbDzjOoL3MWJomTbORhHPdK9KIvhB8Nq
clkFeZt16ddPANCY/ktaWFrxUYDUu50e//pAeiG6qULeB+ow7B52V9hoSIwNNUbkKqrGCvTdI0MyEbKU
Cf+nlSWVEwQtjIl6HRoPh1/nd8gXl+o3vWsZFwuK0eaxRqGB6XgWVbXl/TOlRyoanguItNGavXDZvcJX
wWFYhumBzkMUW6aFt8enZfizW5obA6YkfUetMYl+zVxTE9nCbWWqfkPNEnp0KPRYYKhjOWKDCpgGYRXQ
ODTf5rgDBNyMA0dW6Wz8TayTNdGrcEBLQB1G7n25IcrQeZKu+DgqGNwDLPXPTbJ1lnxa9Z/rd59sGxSE
uczV0YoWNZY5K7CRfIlec7p1wJSemYeknULyZRXJb8dNsPBP0JYlr4Fr8+D9zlAwURPh5dnbaFRRL0Xt
99XwowH0vGiWe27Dgo4j1PEWH7glhpS46gzrsSQ12BzwvqmexRxd2yOH38e44P41hpj0mASYpVStP6k8
8x/eCpmFfEKpuE0WlDTP401PBLfufFg3kZo7QY4saon5qVeZzXycfFqkg61rYqZMF3an9VGoPAUnm1A5
~/B-SEC-500> ./challenge09 input09.txt | fold -w 80 | head -n 13
SW4gU2VwdGVtYmVyIDE5ODQsIEkgc3RhcnRlZCB3cml0aW5nIEdOVSBFbWFjcywgd2hpY2ggd2FzIG15
IHNlY29uZAppbXBsZW1lbnRhdGlvbiBvZiBFbWFjcywgYW5kIGJ5IGVhcmx5IDE5ODUsIGl0IHdhcyB3
b3JraW5nLiBJIGNvdWxkCnVzZSBpdCBmb3IgYWxsIG15IGVkaXRpbmcsIHdoaWNoIHdhcyBhIGJpZyBy
ZWxpZWYsIGJlY2F1c2UgSSBoYWQgbm8KaW50ZW50aW9uIG9mIGxlYXJuaW5nIHRvIHVzZSBWSSwgdGhl
IFVOSVggZWRpdG9yLiBbTGF1Z2h0ZXJdIFNvLCB1bnRpbAp0aGF0IHRpbWUsIEkgZGlkIG15IGVkaXRp
bmcgb24gc29tZSBvdGhlciBtYWNoaW5lLCBhbmQgc2F2ZWQgdGhlCmZpbGVzIHRocm91Z2ggdGhlIG5l
dHdvcmssIHNvIHRoYXQgSSBjb3VsZCB0aGUgc2FtZSB0dCB3aGVuIEdOVQpFbWFjcyB3YXMgcnVu
bmluZyB3ZWxsIGVub3VnaCBmb3IgbWUgdG8gdXNlIG10LCBpdCB3YXMgYWxzbyAtLSBvdGhlcgpwZW9w
bGUgd2FudGVkIHRvIHVzZSBpdCB0b28uCgpTbyBJIGhhZCB0byB3b3JrIG91dCB0aGUgZGV0YWlscyBv
ZiBkaXN0cmlidXRpb24uIE9mIGNvdXJzZSwgSSBwdXQgYSBjb3B5CmluIHRoZSBhbm9ueW1vdXMgRlRQ
IGRpcmVjdG9yeSwgYW5kIHRoYXQgd2FzIGZpbmUgZm9yIHBlb3BsZSB3aG8gd2VyZSBvbgp0aGUgbmV0
LiBUaGV5IGNvdWxkIHRoZW4ganVzdCBwdWxsIG92ZXIgYSB0YXIgZmlsZSwgYnV0IGEgbG90IG9mCnBy
b2dyYW1tZXJzIHRoZW4gZXZlbiB3ZXJlIG5vdCBvbiB0aGUgbmV0IGluIDE5ODUu
```

It is strictly forbidden to use a library function that will implement CBC for you.
Only ECB mode is allowed.

Challenge 10 - Byte-at-a-time ECB decryption

In this challenge you have to attack a server that

1. takes an input encoded in base64,
2. appends an unknown string to it,
3. encrypts the combination under a consistent but unknown key using ECB,
4. returns the base64 encoded result.

Here is an example with `ELLIPTIC` as input, `CURVE` as appended unknown string, and `--CRYPTOGRAPHY--` as key:

```
Terminal
~/B-SEC-500> curl 127.0.0.1:5000/challenge10 -X POST -d 'RUxMSVBUSUM='
gUxtW3vc2NJjl08d/Yq/sA==
```

You thus have a function that does `AES-128-ECB(your-string || unknown-string, random-key)`.
It turns out that you can decrypt the `unknown-string` with repeated call to that function, without having any idea of what the key is…
Here is how:

1. First discover the block size of the cipher, you already know it, but do this step anyway.
   You can easily do that by feeding payloads of incrementing length and observe when and by how much the length of the received text changes.

2. Detect that the function is using ECB.
   You also know that, but again, do this step anyway.

Before going further, make sure you understood properly the concepts involved in step 1 and 2.

3. Knowing the block size, craft an input block that is exactly 1 byte shorter than the block size (e.g. "AAAAAAA" if the block size is 8).
   Think about what the function is going to put in the last byte position. Keep that result somewhere.

4. Now, add to that crafted block a single byte, and make its value range from the minimum to the maximum value of a single byte ("AAAAAAA\x00", "AAAAAAA\x01" … "AAAAAAA\xff").
   Once the encrypted result matches with the block you kept track of ("AAAAAAAX" where X is the unknown byte), you have discovered the first byte of the unknown string.

5. Repeat the operation for the next byte, making sure the preceding byte is the one you previously discovered ("AAAAAAX\x00" the first time, then "AAAAAXY\x00" and so on).

Your program should then output the decrypted unknown string, in base64 format, followed by a newline.

Congratulations!
This is the first challenge whose solution will break real crypto.
Lots of people know that when you encrypt something in ECB mode, you can see penguins through it.
Not so many of them can decrypt the contents of those ciphertexts, and now you can.

Your program will be tested with host 127.0.0.1 and port 5000.

Don't forget to send the session cookies we give you along with each request, it contains an id that will make sure your payload is appended with the same unknown string and encrypted with the same key.

For ALL client-server challenges, messages are always base64 encoded. Also, be sure to check that the url DOES NOT end with a slash (/challenge10 != /challenge10/).

Challenge 11 - ECB cut-and-paste

You are this time provided with a server that:

1. takes an base64 encoded string as input,
2. creates a profile associated with this input,
3. encodes it in a key value fashion,
4. encrypts this encoded profile in ECB mode with an unknown key,
5. returns it in base64.

Omitting the encryption part, the server does `profile_for("foo@bar.com") -> "email=foo@bar.com&uid=10&role=user"`.

For testing purposes, you can easily recode the profile_for function, and encrypt it with a random key that will remain unknown from you.

Using only the input of the function to generate valid ciphertexts, at url `/challenge11/new_profile`, and the ciphertexts themselves, craft a ciphertext that will decrypt to a profile with `role=admin`.
Once you have this crafted block, POST it back to the server, base64 encoded, on `/challenge11/validate`.
If your ciphertext correctly decrypts to a profile with the admin role, you receive back a token in base64 that you have to print on the standard output followed by a newline to complete the challenge.

No, the answer is not to give *h4ck3r@l4m3r.n00b&role=admin* as input.
Special characters '&' and '=' will be escaped.

This challenge is doable in only two calls to `new_profile`. One to craft a block whose `role=` will be aligned to the end of a block, and one that will craft a block with `admin` aligned to the beginning of a block, padded to the size of a block.

Challenge 12 - Harder Byte-at-a-time ECB decryption

Same as challenge 10, except that the function now prefixes your input with a random string of random length, which gives you: `AES-128-ECB(random-prefix || your-string || unknown-string, random-key)`.

Now, what makes it harder to find the unknown string compared to challenge 10?

You already have most of the tools needed to decrypt that unknown string.
Improvise. Adapt. Overcome.

Challenge 13 - CBC bitflipping attacks

You now have a server that

1. takes an base64 encoded string as input,
2. deletes the ";" and "=" within the string,
3. prefixes it with something like "title=Announcement;content=",
4. suffixes it with something like ";type=jibberjabber;",
5. encrypts this key-value encoded string in CBC mode with an unknown key and iv,
6. returns it in base64.

Using only the input of the function to generate valid ciphertexts, at url `/challenge13/encrypt`, and the ciphertexts themselves, craft a ciphertext that will decrypt to a message containing `;admin=true;`.
Once you have this crafted block, POST it back to the server, base64 encoded, on `/challenge13/decrypt`.
Again, if your ciphertext correctly decrypts to a message with the admin role, you receive back a token that you have to print in base64 on the standard output followed by a newline to complete the challenge.

For this exercise, you are relying on the fact that in CBC mode, a 1-bit error in a ciphertext block:
- Completely scrambles the block the error occurs in
- Produces the identical 1-bit error(/edit) in the next ciphertext block.

When calculating the index at which you will have to operate the bitflips, always compute the prefix's length before. It will not always be "title=Announcement;userdata="

Challenge 14 - The CBC padding oracle

This attack is the most popular attack on modern block-cipher cryptography.

You have a server that provides you with an IV and a ciphertext. This ciphertext is not based on your input. The IV and ciphertext will be given to you in that order on `/challenge14/encrypt`, both in base64, separated by a newline. The verb for that request is GET.

You then have a second entrypoint, `/challenge14/decrypt`, that takes a ciphertext and and its IV, decrypts it, and check if it has a valid padding. If it does, it returns an 'OK' message with status code 200. If it doesn't, it returns a 'Bad padding' with status code 500. The verb for that request is POST.

This pair of functions approximates AES-CBC encryption as its deployed serverside in web applications; the second function models the server's consumption of an encrypted session token, as if it was a cookie.

It turns out that it's possible to decrypt the ciphertext provided by the first function with repeated call to the second function.

The decryption here works thanks to a *side-channel* leak from the decryption function. The actual leak is the error message stating that the padding is valid or not.

There are thousands of webpages that describe this attack online, the entire process will thus not be re-explained here.

However, fundamentally

0x01 on its own is a valid padding byte if placed at the last index of a 16 bytes message. This will occur 1/256 times in a plaintext produced by the decryption of a tampered ciphertext.

0x02 on its own is *not* a valid padding.

0x02 0x02 is a valid padding, but is way less likely to occur randomly than 0x01.

0x03 0x03 0x03 is even less likely.

etc.

You can then assume that if you corrupt a ciphertext and the decrypted message has a valid padding, you can know what the padding byte is.

Again, the `/challenge14/encrypt` route doesn't take any parameter.
The `/challenge14/decrypt`, takes a IV, followed by a ciphertext, both in base64 and separated by a newline.

Once the ciphertext provided by the encrypt route is decrypted, print it in base64 on the standard output, followed by a newline.

Understand this: Padding oracles have nothing to do with the actual padding on a CBC plaintext. It's an attack that targets the code that handles decryption.