**Implement PKCS#7 padding**

A block cipher 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 blocksize. The most popular padding scheme is called PKCS#7.

So: pad any block to a specific block length, by appending the number of bytes of padding to the end of the block. For instance,

"YELLOW SUBMARINE"

... padded to 20 bytes would be:

"YELLOW SUBMARINE\x04\x04\x04\x04"

### Implement CBC mode

CBC mode is a block cipher mode that allows us to encrypt irregularly-sized messages, despite the fact that a block cipher natively only transforms individual blocks.

In CBC mode, each ciphertext block is added to the next plaintext block before the next call to the cipher core.

The first plaintext block, which has no associated previous ciphertext block, is added to a "fake 0th ciphertext block" called the *initialization vector*, or IV.

Implement CBC mode by hand by taking the ECB function you wrote earlier, making it *encrypt* instead of *decrypt* (verify this by decrypting whatever you encrypt to test), and using your XOR function from the previous exercise to combine them.

[The file here](http://cryptopals.com/static/challenge-data/10.txt) is intelligible (somewhat) when CBC decrypted against "YELLOW SUBMARINE" with an IV of all ASCII 0 (\x00\x00\x00 &c)

### Don't cheat.

Do not use OpenSSL's CBC code to do CBC mode, even to verify your results. What's the point of even doing this stuff if you aren't going to learn from it?

### An ECB/CBC detection oracle

Now that you have ECB and CBC working:

Write a function to generate a random AES key; that's just 16 random bytes.

Write a function that encrypts data under an unknown key --- that is, a function that generates a random key and encrypts under it.

The function should look like:

encryption\_oracle(your-input)

=> [MEANINGLESS JIBBER JABBER]

Under the hood, have the function *append* 5-10 bytes (count chosen randomly) *before* the plaintext and 5-10 bytes *after* the plaintext.

Now, have the function choose to encrypt under ECB 1/2 the time, and under CBC the other half (just use random IVs each time for CBC). Use rand(2) to decide which to use.

Detect the block cipher mode the function is using each time. You should end up with a piece of code that, pointed at a block box that might be encrypting ECB or CBC, tells you which one is happening.

### Byte-at-a-time ECB decryption (Simple)

Copy your oracle function to a new function that encrypts buffers under ECB mode using a *consistent* but *unknown* key (for instance, assign a single random key, once, to a global variable).

Now take that same function and have it append to the plaintext, BEFORE ENCRYPTING, the following string:

Um9sbGluJyBpbiBteSA1LjAKV2l0aCBteSByYWctdG9wIGRvd24gc28gbXkg

aGFpciBjYW4gYmxvdwpUaGUgZ2lybGllcyBvbiBzdGFuZGJ5IHdhdmluZyBq

dXN0IHRvIHNheSBoaQpEaWQgeW91IHN0b3A/IE5vLCBJIGp1c3QgZHJvdmUg

YnkK

### Spoiler alert.

Do not decode this string now. Don't do it.

Base64 decode the string before appending it. *Do not base64 decode the string by hand; make your code do it*. The point is that you don't know its contents.

What you have now is a function that produces:

AES-128-ECB(your-string || unknown-string, random-key)

It turns out: you can decrypt "unknown-string" with repeated calls to the oracle function!

Here's roughly how:

1. Feed identical bytes of your-string to the function 1 at a time --- start with 1 byte ("A"), then "AA", then "AAA" and so on. Discover the block size of the cipher. You know it, but do this step anyway.
2. Detect that the function is using ECB. You already know, but do this step anyways.
3. Knowing the block size, craft an input block that is exactly 1 byte short (for instance, if the block size is 8 bytes, make "AAAAAAA"). Think about what the oracle function is going to put in that last byte position.
4. Make a dictionary of every possible last byte by feeding different strings to the oracle; for instance, "AAAAAAAA", "AAAAAAAB", "AAAAAAAC", remembering the first block of each invocation.
5. Match the output of the one-byte-short input to one of the entries in your dictionary. You've now discovered the first byte of unknown-string.
6. Repeat for the next byte.

### Congratulations.

This is the first challenge we've given you 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. If our experience is any guideline, this attack will get you code execution in security tests about once a year.

**ECB cut-and-paste**

Write a k=v parsing routine, as if for a structured cookie. The routine should take:

foo=bar&baz=qux&zap=zazzle

... and produce:

{

foo: 'bar',

baz: 'qux',

zap: 'zazzle'

}

(you know, the object; I don't care if you convert it to JSON).

Now write a function that encodes a user profile in that format, given an email address. You should have something like:

profile\_for("foo@bar.com")

... and it should produce:

{

email: 'foo@bar.com',

uid: 10,

role: 'user'

}

... encoded as:

email=foo@bar.com&uid=10&role=user

Your "profile\_for" function should *not* allow encoding metacharacters (& and =). Eat them, quote them, whatever you want to do, but don't let people set their email address to "foo@bar.com&role=admin".

Now, two more easy functions. Generate a random AES key, then:

1. Encrypt the encoded user profile under the key; "provide" that to the "attacker".
2. Decrypt the encoded user profile and parse it.

Using only the user input to profile\_for() (as an oracle to generate "valid" ciphertexts) and the ciphertexts themselves, make a role=admin profile.

### Byte-at-a-time ECB decryption (Harder)

Take your oracle function [from #12.](http://cryptopals.com/sets/2/challenges/12) Now generate a random count of random bytes and prepend this string to every plaintext. You are now doing:

AES-128-ECB(random-prefix || attacker-controlled || target-bytes, random-key)

Same goal: decrypt the target-bytes.

### Stop and think for a second.

What's harder than challenge #12 about doing this? How would you overcome that obstacle? The hint is: you're using all the tools you already have; no crazy math is required.

Think "STIMULUS" and "RESPONSE".

### PKCS#7 padding validation

Write a function that takes a plaintext, determines if it has valid PKCS#7 padding, and strips the padding off.

The string:

"ICE ICE BABY\x04\x04\x04\x04"

... has valid padding, and produces the result "ICE ICE BABY".

The string:

"ICE ICE BABY\x05\x05\x05\x05"

... does not have valid padding, nor does:

"ICE ICE BABY\x01\x02\x03\x04"

If you are writing in a language with exceptions, like Python or Ruby, make your function throw an exception on bad padding.

Crypto nerds know where we're going with this. Bear with us.

**CBC bitflipping attacks**

Generate a random AES key.

Combine your padding code and CBC code to write two functions.

The first function should take an arbitrary input string, prepend the string:

"comment1=cooking%20MCs;userdata="

.. and append the string:

";comment2=%20like%20a%20pound%20of%20bacon"

The function should quote out the ";" and "=" characters.

The function should then pad out the input to the 16-byte AES block length and encrypt it under the random AES key.

The second function should decrypt the string and look for the characters ";admin=true;" (or, equivalently, decrypt, split the string on ";", convert each resulting string into 2-tuples, and look for the "admin" tuple).

Return true or false based on whether the string exists.

If you've written the first function properly, it should *not* be possible to provide user input to it that will generate the string the second function is looking for. We'll have to break the crypto to do that.

Instead, modify the ciphertext (without knowledge of the AES key) to accomplish this.

You're 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.

**Stop and think for a second.**

Before you implement this attack, answer this question: why does CBC mode have this property?