**The CBC padding oracle**

This is the best-known attack on modern block-cipher cryptography.

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

The first function should select at random one of the following 10 strings:

MDAwMDAwTm93IHRoYXQgdGhlIHBhcnR5IGlzIGp1bXBpbmc=

MDAwMDAxV2l0aCB0aGUgYmFzcyBraWNrZWQgaW4gYW5kIHRoZSBWZWdhJ3MgYXJlIHB1bXBpbic=

MDAwMDAyUXVpY2sgdG8gdGhlIHBvaW50LCB0byB0aGUgcG9pbnQsIG5vIGZha2luZw==

MDAwMDAzQ29va2luZyBNQydzIGxpa2UgYSBwb3VuZCBvZiBiYWNvbg==

MDAwMDA0QnVybmluZyAnZW0sIGlmIHlvdSBhaW4ndCBxdWljayBhbmQgbmltYmxl

MDAwMDA1SSBnbyBjcmF6eSB3aGVuIEkgaGVhciBhIGN5bWJhbA==

MDAwMDA2QW5kIGEgaGlnaCBoYXQgd2l0aCBhIHNvdXBlZCB1cCB0ZW1wbw==

MDAwMDA3SSdtIG9uIGEgcm9sbCwgaXQncyB0aW1lIHRvIGdvIHNvbG8=

MDAwMDA4b2xsaW4nIGluIG15IGZpdmUgcG9pbnQgb2g=

MDAwMDA5aXRoIG15IHJhZy10b3AgZG93biBzbyBteSBoYWlyIGNhbiBibG93

... generate a random AES key (which it should save for all future encryptions), pad the string out to the 16-byte AES block size and CBC-encrypt it under that key, providing the caller the ciphertext and IV.

The second function should consume the ciphertext produced by the first function, decrypt it, check its padding, and return true or false depending on whether the padding is valid.

**What you're doing here.**

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 ciphertexts provided by the first function.

The decryption here depends on a side-channel leak by the decryption function. The leak is the error message that the padding is valid or not.

You can find 100 web pages on how this attack works, so I won't re-explain it. What I'll say is this:

The fundamental insight behind this attack is that the byte 01h is valid padding, and occur in 1/256 trials of "randomized" plaintexts produced by decrypting a tampered ciphertext.

02h in isolation is *not* valid padding.

02h 02h *is* valid padding, but is much less likely to occur randomly than 01h.

03h 03h 03h is even less likely.

So you can assume that if you corrupt a decryption AND it had valid padding, you know what that padding byte is.

It is easy to get tripped up on the fact that CBC plaintexts are "padded". *Padding oracles have nothing to do with the actual padding on a CBC plaintext.* It's an attack that targets a specific bit of code that handles decryption. You can mount a padding oracle on *any CBC block*, whether it's padded or not.

### Implement CTR, the stream cipher mode

The string:

L77na/nrFsKvynd6HzOoG7GHTLXsTVu9qvY/2syLXzhPweyyMTJULu/6/kXX0KSvoOLSFQ==

... decrypts to something approximating English in CTR mode, which is an AES block cipher mode that turns AES into a stream cipher, with the following parameters:

key=YELLOW SUBMARINE

nonce=0

format=64 bit unsigned little endian nonce,

64 bit little endian block count (byte count / 16)

CTR mode is very simple.

Instead of encrypting the plaintext, CTR mode encrypts a running counter, producing a 16 byte block of keystream, which is XOR'd against the plaintext.

For instance, for the first 16 bytes of a message with these parameters:

keystream = AES("YELLOW SUBMARINE",

"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00")

... for the next 16 bytes:

keystream = AES("YELLOW SUBMARINE",

"\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00")

... and then:

keystream = AES("YELLOW SUBMARINE",

"\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00")

CTR mode does not require padding; when you run out of plaintext, you just stop XOR'ing keystream and stop generating keystream.

Decryption is identical to encryption. Generate the same keystream, XOR, and recover the plaintext.

Decrypt the string at the top of this function, then use your CTR function to encrypt and decrypt other things.

### This is the only block cipher mode that matters in good code.

Most modern cryptography relies on CTR mode to adapt block ciphers into stream ciphers, because most of what we want to encrypt is better described as a stream than as a sequence of blocks. Daniel Bernstein once quipped to Phil Rogaway that good cryptosystems don't need the "decrypt" transforms. Constructions like CTR are what he was talking about.

**Break fixed-nonce CTR mode using substitions**

Take your CTR encrypt/decrypt function and fix its nonce value to 0. Generate a random AES key.

In *successive encryptions* (*not* in one big running CTR stream), encrypt each line of the base64 decodes of the following, producing multiple independent ciphertexts:

SSBoYXZlIG1ldCB0aGVtIGF0IGNsb3NlIG9mIGRheQ==

Q29taW5nIHdpdGggdml2aWQgZmFjZXM=

RnJvbSBjb3VudGVyIG9yIGRlc2sgYW1vbmcgZ3JleQ==

RWlnaHRlZW50aC1jZW50dXJ5IGhvdXNlcy4=

SSBoYXZlIHBhc3NlZCB3aXRoIGEgbm9kIG9mIHRoZSBoZWFk

T3IgcG9saXRlIG1lYW5pbmdsZXNzIHdvcmRzLA==

T3IgaGF2ZSBsaW5nZXJlZCBhd2hpbGUgYW5kIHNhaWQ=

UG9saXRlIG1lYW5pbmdsZXNzIHdvcmRzLA==

QW5kIHRob3VnaHQgYmVmb3JlIEkgaGFkIGRvbmU=

T2YgYSBtb2NraW5nIHRhbGUgb3IgYSBnaWJl

VG8gcGxlYXNlIGEgY29tcGFuaW9u

QXJvdW5kIHRoZSBmaXJlIGF0IHRoZSBjbHViLA==

QmVpbmcgY2VydGFpbiB0aGF0IHRoZXkgYW5kIEk=

QnV0IGxpdmVkIHdoZXJlIG1vdGxleSBpcyB3b3JuOg==

QWxsIGNoYW5nZWQsIGNoYW5nZWQgdXR0ZXJseTo=

QSB0ZXJyaWJsZSBiZWF1dHkgaXMgYm9ybi4=

VGhhdCB3b21hbidzIGRheXMgd2VyZSBzcGVudA==

SW4gaWdub3JhbnQgZ29vZCB3aWxsLA==

SGVyIG5pZ2h0cyBpbiBhcmd1bWVudA==

VW50aWwgaGVyIHZvaWNlIGdyZXcgc2hyaWxsLg==

V2hhdCB2b2ljZSBtb3JlIHN3ZWV0IHRoYW4gaGVycw==

V2hlbiB5b3VuZyBhbmQgYmVhdXRpZnVsLA==

U2hlIHJvZGUgdG8gaGFycmllcnM/

VGhpcyBtYW4gaGFkIGtlcHQgYSBzY2hvb2w=

QW5kIHJvZGUgb3VyIHdpbmdlZCBob3JzZS4=

VGhpcyBvdGhlciBoaXMgaGVscGVyIGFuZCBmcmllbmQ=

V2FzIGNvbWluZyBpbnRvIGhpcyBmb3JjZTs=

SGUgbWlnaHQgaGF2ZSB3b24gZmFtZSBpbiB0aGUgZW5kLA==

U28gc2Vuc2l0aXZlIGhpcyBuYXR1cmUgc2VlbWVkLA==

U28gZGFyaW5nIGFuZCBzd2VldCBoaXMgdGhvdWdodC4=

VGhpcyBvdGhlciBtYW4gSSBoYWQgZHJlYW1lZA==

QSBkcnVua2VuLCB2YWluLWdsb3Jpb3VzIGxvdXQu

SGUgaGFkIGRvbmUgbW9zdCBiaXR0ZXIgd3Jvbmc=

VG8gc29tZSB3aG8gYXJlIG5lYXIgbXkgaGVhcnQs

WWV0IEkgbnVtYmVyIGhpbSBpbiB0aGUgc29uZzs=

SGUsIHRvbywgaGFzIHJlc2lnbmVkIGhpcyBwYXJ0

SW4gdGhlIGNhc3VhbCBjb21lZHk7

SGUsIHRvbywgaGFzIGJlZW4gY2hhbmdlZCBpbiBoaXMgdHVybiw=

VHJhbnNmb3JtZWQgdXR0ZXJseTo=

QSB0ZXJyaWJsZSBiZWF1dHkgaXMgYm9ybi4=

(This should produce 40 short CTR-encrypted ciphertexts).

Because the CTR nonce wasn't randomized for each encryption, each ciphertext has been encrypted against the same keystream. This is very bad.

Understanding that, like most stream ciphers (including RC4, and obviously any block cipher run in CTR mode), the actual "encryption" of a byte of data boils down to a single XOR operation, it should be plain that:

CIPHERTEXT-BYTE XOR PLAINTEXT-BYTE = KEYSTREAM-BYTE

And since the keystream is the same for every ciphertext:

CIPHERTEXT-BYTE XOR KEYSTREAM-BYTE = PLAINTEXT-BYTE (ie, "you don't

say!")

Attack this cryptosystem piecemeal: guess letters, use expected English language frequence to validate guesses, catch common English trigrams, and so on.

**Don't overthink it.**

Points for automating this, but part of the reason I'm having you do this is that I think this approach is suboptimal.

### Break fixed-nonce CTR statistically

[In this file](http://cryptopals.com/static/challenge-data/20.txt) find a similar set of Base64'd plaintext. Do with them exactly what you did with the first, but solve the problem differently.

Instead of making spot guesses at to known plaintext, treat the collection of ciphertexts the same way you would repeating-key XOR.

Obviously, CTR encryption appears different from repeated-key XOR, *but with a fixed nonce they are effectively the same thing.*

To exploit this: take your collection of ciphertexts and truncate them to a common length (the length of the smallest ciphertext will work).

Solve the resulting concatenation of ciphertexts as if for repeating- key XOR, with a key size of the length of the ciphertext you XOR'd.

### Implement the MT19937 Mersenne Twister RNG

You can get the psuedocode for this from Wikipedia.

If you're writing in Python, Ruby, or (gah) PHP, your language is probably already giving you MT19937 as "rand()"; **don't use rand()**. Write the RNG yourself.

### Crack an MT19937 seed

Make sure your MT19937 accepts an integer seed value. Test it (verify that you're getting the same sequence of outputs given a seed).

Write a routine that performs the following operation:

* Wait a random number of seconds between, I don't know, 40 and 1000.
* Seeds the RNG with the current Unix timestamp
* Waits a random number of seconds again.
* Returns the first 32 bit output of the RNG.

You get the idea. Go get coffee while it runs. Or just simulate the passage of time, although you're missing some of the fun of this exercise if you do that.

From the 32 bit RNG output, discover the seed.

### Clone an MT19937 RNG from its output

The internal state of MT19937 consists of 624 32 bit integers.

For each batch of 624 outputs, MT permutes that internal state. By permuting state regularly, MT19937 achieves a period of 2\*\*19937, which is Big.

Each time MT19937 is tapped, an element of its internal state is subjected to a tempering function that diffuses bits through the result.

The tempering function is invertible; you can write an "untemper" function that takes an MT19937 output and transforms it back into the corresponding element of the MT19937 state array.

To invert the temper transform, apply the inverse of each of the operations in the temper transform in reverse order. There are two kinds of operations in the temper transform each applied twice; one is an XOR against a right-shifted value, and the other is an XOR against a left-shifted value AND'd with a magic number. So you'll need code to invert the "right" and the "left" operation.

Once you have "untemper" working, create a new MT19937 generator, tap it for 624 outputs, untemper each of them to recreate the state of the generator, and splice that state into a new instance of the MT19937 generator.

The new "spliced" generator should predict the values of the original.

### Stop and think for a second.

How would you modify MT19937 to make this attack hard? What would happen if you subjected each tempered output to a cryptographic hash?

### Create the MT19937 stream cipher and break it

You can create a trivial stream cipher out of any PRNG; use it to generate a sequence of 8 bit outputs and call those outputs a keystream. XOR each byte of plaintext with each successive byte of keystream.

Write the function that does this for MT19937 using a 16-bit seed. Verify that you can encrypt and decrypt properly. This code should look similar to your CTR code.

Use your function to encrypt a known plaintext (say, 14 consecutive 'A' characters) prefixed by a random number of random characters.

From the ciphertext, recover the "key" (the 16 bit seed).

Use the same idea to generate a random "password reset token" using MT19937 seeded from the current time.

Write a function to check if any given password token is actually the product of an MT19937 PRNG seeded with the current time.