**Break "random access read/write" AES CTR**

Back to CTR. Encrypt the recovered plaintext from [this file](http://cryptopals.com/static/challenge-data/25.txt) (the ECB exercise) under CTR with a random key (for this exercise the key should be unknown to you, but hold on to it).

Now, write the code that allows you to "seek" into the ciphertext, decrypt, and re-encrypt with different plaintext. Expose this as a function, like, *"edit(ciphertext, key, offet, newtext)"*.

Imagine the "edit" function was exposed to attackers by means of an API call that didn't reveal the key or the original plaintext; the attacker has the ciphertext and controls the offset and "new text".

Recover the original plaintext.

**Food for thought.**

A folkloric supposed benefit of CTR mode is the ability to easily "seek forward" into the ciphertext; to access byte N of the ciphertext, all you need to be able to do is generate byte N of the keystream. Imagine if you'd relied on that advice to, say, encrypt a disk.

### CTR bitflipping

There are people in the world that believe that CTR resists bit flipping attacks of the kind to which CBC mode is susceptible.

Re-implement [the CBC bitflipping exercise from earlier](http://cryptopals.com/sets/2/challenges/16) to use CTR mode instead of CBC mode. Inject an "admin=true" token.

**Recover the key from CBC with IV=Key**

Take your code from [the CBC exercise](http://cryptopals.com/sets/2/challenges/16) and modify it so that it repurposes the key for CBC encryption as the IV.

Applications sometimes use the key as an IV on the auspices that both the sender and the receiver have to know the key already, and can save some space by using it as both a key and an IV.

Using the key as an IV is insecure; an attacker that can modify ciphertext in flight can get the receiver to decrypt a value that will reveal the key.

The CBC code from exercise 16 encrypts a URL string. Verify each byte of the plaintext for ASCII compliance (ie, look for high-ASCII values). Noncompliant messages should raise an exception or return an error that includes the decrypted plaintext (this happens all the time in real systems, for what it's worth).

Use your code to encrypt a message that is at least 3 blocks long:

AES-CBC(P\_1, P\_2, P\_3) -> C\_1, C\_2, C\_3

Modify the message (you are now the attacker):

C\_1, C\_2, C\_3 -> C\_1, 0, C\_1

Decrypt the message (you are now the receiver) and raise the appropriate error if high-ASCII is found.

As the attacker, recovering the plaintext from the error, extract the key:

P'\_1 XOR P'\_3

**Implement a SHA-1 keyed MAC**

Find a SHA-1 implementation in the language you code in.

**Don't cheat. It won't work.**

Do not use the SHA-1 implementation your language already provides (for instance, don't use the "Digest" library in Ruby, or call OpenSSL; in Ruby, you'd want a pure-Ruby SHA-1).

Write a function to authenticate a message under a secret key by using a secret-prefix MAC, which is simply:

SHA1(key || message)

Verify that you cannot tamper with the message without breaking the MAC you've produced, and that you can't produce a new MAC without knowing the secret key.

### Break a SHA-1 keyed MAC using length extension

Secret-prefix SHA-1 MACs are trivially breakable.

The attack on secret-prefix SHA1 relies on the fact that you can take the ouput of SHA-1 and use it as a new starting point for SHA-1, thus taking an arbitrary SHA-1 hash and "feeding it more data".

Since the key precedes the data in secret-prefix, any additional data you feed the SHA-1 hash in this fashion will appear to have been hashed with the secret key.

To carry out the attack, you'll need to account for the fact that SHA-1 is "padded" with the bit-length of the message; your forged message will need to include that padding. We call this "glue padding". The final message you actually forge will be:

SHA1(key || original-message || glue-padding || new-message)

(where the final padding on the whole constructed message is implied)

Note that to generate the glue padding, you'll need to know the original bit length of the message; the message itself is known to the attacker, but the secret key isn't, so you'll need to guess at it.

This sounds more complicated than it is in practice.

To implement the attack, first write the function that computes the MD padding of an arbitrary message and verify that you're generating the same padding that your SHA-1 implementation is using. This should take you 5-10 minutes.

Now, take the SHA-1 secret-prefix MAC of the message you want to forge --- this is just a SHA-1 hash --- and break it into 32 bit SHA-1 registers (SHA-1 calls them "a", "b", "c", &c).

Modify your SHA-1 implementation so that callers can pass in new values for "a", "b", "c" &c (they normally start at magic numbers). With the registers "fixated", hash the additional data you want to forge.

Using this attack, generate a secret-prefix MAC under a secret key (choose a random word from /usr/share/dict/words or something) of the string:

"comment1=cooking%20MCs;userdata=foo;comment2=%20like%20a%20pound%20of%20bacon"

Forge a variant of this message that ends with ";admin=true".

### This is a very useful attack.

For instance: Thai Duong and Juliano Rizzo, who got to this attack before we did, used it to break the Flickr API.

### Break an MD4 keyed MAC using length extension

Second verse, same as the first, but use MD4 instead of SHA-1. Having done this attack once against SHA-1, the MD4 variant should take much less time; mostly just the time you'll spend Googling for an implementation of MD4.

### You're thinking, why did we bother with this?

Blame Stripe. In their second CTF game, the second-to-last challenge involved breaking an H(k, m) MAC with SHA1. Which meant that SHA1 code was floating all over the Internet. MD4 code, not so much.

**Implement and break HMAC-SHA1 with an artificial timing leak**

The psuedocode on Wikipedia should be enough. HMAC is very easy.

Using the web framework of your choosing (Sinatra, web.py, whatever), write a tiny application that has a URL that takes a "file" argument and a "signature" argument, like so:

http://localhost:9000/test?file=foo&signature=46b4ec586117154dacd49d664e5d63fdc88efb51

Have the server generate an HMAC key, and then verify that the "signature" on incoming requests is valid for "file", using the "==" operator to compare the valid MAC for a file with the "signature" parameter (in other words, verify the HMAC the way any normal programmer would verify it).

Write a function, call it "insecure\_compare", that implements the == operation by doing byte-at-a-time comparisons with early exit (ie, return false at the first non-matching byte).

In the loop for "insecure\_compare", add a 50ms sleep (sleep 50ms after each byte).

Use your "insecure\_compare" function to verify the HMACs on incoming requests, and test that the whole contraption works. Return a 500 if the MAC is invalid, and a 200 if it's OK.

Using the timing leak in this application, write a program that discovers the valid MAC for any file.

**Why artificial delays?**

Early-exit string compares are probably the most common source of cryptographic timing leaks, but they aren't especially easy to exploit. In fact, many timing leaks (for instance, any in C, C++, Ruby, or Python) probably aren't exploitable over a wide-area network at all. To play with attacking real-world timing leaks, you have to start writing low-level timing code. We're keeping things cryptographic in these challenges.

### Break HMAC-SHA1 with a slightly less artificial timing leak

Reduce the sleep in your "insecure\_compare" until your previous solution breaks. (Try 5ms to start.)

Now break it again.