Project: Cryptographic Challenges*

USPN, Master I, Securité, Avril 2020

Table des matières

1 Qualifying set

1.1 Convert hex to base64

Always operate on raw bytes, never on encoded strings. Only use `hex` and `base64` for pretty-printing.

The string

```
49276d206b696c6c696e6720796f757220627261696e206c696b65206120706f69736f6e
6f7573206d757368726f6f6d
```

should produce

```
SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hyb29t
```

So go ahead and make that happen. You'll need to use this code for the rest of the exercises.

*Original challenges by T. Ptacek, S. Devlin, A. Balducci, and M. Wielgoszewski from `https://cryptopals.com/`

1.2 Fixed XOR

Write a function that takes two equal-length buffers and produces their `XOR` combination. If your function works properly, then when you feed it the string :

```
1c0111001f010100061a024b53535009181c
```

... after hex decoding, and when `XOR`'d against :

```
686974207468652062756c6c277320657965
```

... should produce :

```
746865206b696420646f6e277420706c6179
```

1.3 Single-byte XOR cipher

The hex encoded string :

```
1b37373331363f78151b7f2b783431333d78397828372d363c78373e783a393b3736
```

... has been `XOR`'d against a single character. Find the key, decrypt the message.

You can do this by hand. But don't : write code to do it for you.

How ? Devise some method for "scoring" a piece of English plaintext. Character frequency is a good metric. Evaluate each output and choose the one with the best score.

1.4 Detect single-character XOR

One of the 60-character strings in `detecting-singlechar-xor.txt` has been encrypted by single-character XOR.

Find it.

(Your code from the previous exercise should help.)

1.5 Implement repeating-key XOR

Here is the opening stanza of an important work of the English language :

```
Burning 'em, if you ain't quick and nimble
I go crazy when I hear a cymbal
```

Encrypt it, under the key `"ICE"`, using repeating-key `XOR`.

In repeating-key `XOR`, you'll sequentially apply each byte of the key ; the first byte of plaintext will be `XOR`'d against `I`, the next `C`, the next `E`, then `I` again for the 4th byte, and so on.

It should come out to :

```
0b3637272a2b2e63622c2e69692a23693a2a3c6324202d623d63343c2a26226324272765272
a282b2f20430a652e2c652a3124333a653e2b2027630c692b20283165286326302e27282f
```

Get a feel for it. I promise, we aren't wasting your time with this.

1.6 Break repeating-key XOR

The file `repeating-xor.txt` has been `base64`'d after being encrypted with repeating-key `XOR`.

Decrypt it.

Here's how :

1. Let `KEYSIZE` be the guessed length of the key ; try values from 2 to (say) 40.
2. Write a function to compute the edit distance/Hamming distance between two strings. *The Hamming distance is just the number of differing bits.* The distance between :

   ```
   this is a test
   ```

   and

   ```
   wokka wokka!!!
   ```

   is 37. Make sure your code agrees before you proceed.
3. For each `KEYSIZE`, take the first `KEYSIZE` worth of bytes, and the second `KEYSIZE` worth of bytes, and find the edit distance between them. Normalize this result by dividing by `KEYSIZE`.
4. The `KEYSIZE` with the smallest normalized edit distance is probably the key. You could proceed perhaps with the smallest 2-3 `KEYSIZE` values. Or take 4 `KEYSIZE` blocks instead of 2 and average the distances.
5. Now that you probably know the `KEYSIZE` : break the ciphertext into blocks of `KEYSIZE` length.
6. Now transpose the blocks : make a block that is the first byte of every block, and a block that is the second byte of every block, and so on.
7. Solve each block as if it was single-character `XOR`. You already have code to do this.
8. For each block, the single-byte `XOR` key that produces the best looking histogram is the repeating-key `XOR` key byte for that block. Put them together and you have the key.

This code is going to turn out to be surprisingly useful later on. Breaking repeating-key `XOR` ("Vigenere") statistically is obviously an academic exercise, a "Crypto 101" thing. But more people "know how" to break it than can actually break it, and a similar technique breaks something much more important.

1.7 AES in ECB mode

The `Base64`-encoded content in `aes-in-ecb.txt` has been encrypted via `AES-128` in `ECB` mode under the key

```
"YELLOW SUBMARINE"
```

(case-sensitive, without the quotes ; exactly 16 characters ; I like "YELLOW SUBMARINE" because it's exactly 16 bytes long, and now you do too).

Decrypt it. You know the key, after all.

Easiest way : use `OpenSSL::Cipher` and give it `AES-128-ECB` as the cipher.

Do this with code. You can obviously decrypt this using the `OpenSSL` command-line tool, but we're having you get `ECB` working in code for a reason. You'll need it a lot later on, and not just for attacking `ECB`.

1.8 Detect AES in ECB mode

In `detect-aes-ecb.txt` are a bunch of hex-encoded ciphertexts.

One of them has been encrypted with `ECB`.

Detect it.

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

2 Block crypto

This is the first of several sets on block cipher cryptography. This is bread-and-butter crypto, the kind you'll see implemented in most web software that does crypto.

This set is relatively easy. People that clear set 1 tend to clear set 2 somewhat quickly.

Three of the challenges in this set are extremely valuable in breaking real-world crypto; one allows you to decrypt messages encrypted in the default mode of `AES`, and the other two allow you to rewrite messages encrypted in the most popular modes of `AES`.

2.1 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"
```

2.2 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 `cbc-mode.txt` is intelligible (somewhat) when `CBC` decrypted against

```
"YELLOW SUBMARINE"
```

with an `IV` of all `ASCII 0 (\x00\x00\x00 etc.)`

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 ?

2.3 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 `IV`s 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.

2.4 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.

2.5 *Byte-at-a-time ECB decryption (Harder)

Take your oracle function from Section 2.4. 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 Section 2.4 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.

2.6 *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.

2.7 *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 ?

2.8 *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.