README.md

Haggis (Crypto 100)

###ENG PL

In the task we get source code using AES CBC. The code is quite short and straightforward:

```
pad = lambda m: m + bytes([16 - len(m) % 16] * (16 - len(m) % 16))
def haggis(m):
    crypt0r = AES.new(bytes(0x10), AES.MODE_CBC, bytes(0x10))
    return crypt0r.encrypt(len(m).to_bytes(0x10, 'big') + pad(m))[-0x10:]

target = os.urandom(0x10)
print(binascii.hexlify(target).decode())

msg = binascii.unhexlify(input())

if msg.startswith(b'I solemnly swear that I am up to no good.\0') \
        and haggis(msg) == target:
    print(open('flag.txt', 'r').read().strip())
```

The server gets random 16 bytes and sends them to us. Then we need to provide a message with a pre-defined prefix. This message is concatenated with the length of the message and encrypted, and the last block of the this ciphertext has to match the random 16 bytes we were given.

We know that it's AES CBC, we know the key is all `0x0` and so is the `IV`. We also know that the cipher is using PKCS7 padding scheme.

We start by filling the prefix until the end of 16 bytes AES block. It's always easier to work with full blocks:

```
msg_start = b'I solemnly swear that I am up to no good.\x00\x00\x00\x00\x00\x00\x00'
```

We will add one more block after this one. Keeping this in mind we calculate the length of the full message and construct the length block, just as the server will do:

```
len_prefix = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@'
```

This way we know exactly what the server is going to encrypt.

It's worth to understand how CBC mode works: each block before encryption is XORed with previous block ciphertext (and first block with IV). This means that if we know the ciphertext of encrypted K blocks, we can encrypt additional blocks simply by passing the last block of K as IV. We are going to leverage this here! First we calculate the first K blocks (2 to be exact, the length block and the message we got in the task):

```
encryptor = AES.new(bytes(0x10), AES.MODE_CBC, bytes(0x10))
prefix_encrypted_block = encryptor.encrypt(len_prefix + msg_start)[-16:]
```

We need only the last block because this is what is going to be XORed with our additional payload block before encryption. We should remember that there is PCKS padding here, so by adding a whole block of our choosing, the actual last ciphertext block will be encrypted padding! So we actually need to make sure this encrypted padding block matches the given target bytes. We know the padding bytes - they will all be `0x10`, but they will get xored with the ciphertext of the payload block we are preparing before encryption. Let's call ciphertext of our payload block `CT_payload`, and the plaintext of this block as `payload`.

Let's look what exactly we need to do:

We want to get: `encrypt(CT_payload xor padding) = target` therefore by applying decryption we get:

```
CT_payload xor padding = decrypt(target)
```

and since xor twice by the same value removes itself:

```
CT_payload = decrypt(target) xor padding
```

Now let's look where we get the `CT_payload` from:

```
CT_payload = encrypt(payload xor prefix_encrypted_block)
```

and by applying decryption:

```
decrypt(CT_payload) = payload xor prefix_encrypted_block
```

and thus:

```
payload = decrypt(CT_payload) xor prefix_encrypted_block
```

And if we now combine the two we get:

```
payload = decrypt(decrypt(target) xor padding) xor prefix_encrypted_block
```

And this is how we can calculate the payload we need to send.

We implement this in python:

```
def solve_for_target(target):
    # enc(ct xor padding) = target
    # ct xor padding = dec(target)
    # ct = dec(target) xor padding
    # ct = enc (pt xor enc_prefix)
    # dec(ct) = pt xor enc_prefix
    # pt = dec(ct) xor enc_prefix
    target = binascii.unhexlify(target)
    encryptor = AES.new(bytes(0x10), AES.MODE_CBC, bytes(0x10))
    data = encryptor.decrypt(target)[-16:]  # ct xor padding
    last_block = b''
    expected_ct_bytes = b''
    for i in range(len(data)):
        expected_ct = (data[i] ^ 0x10)  # ct
        expected_ct_byte = expected_ct.to_bytes(1, 'big')
        expected_ct_bytes += expected_ct_byte
    encryptor = AES.new(bytes(0x10), AES.MODE_CBC, bytes(0x10))
    result_bytes = encryptor.decrypt(expected_ct_bytes)  # dec(ct)
    for i in range(len(result_bytes)):
        pt = result_bytes[i] ^ prefix_encrypted_block[i]  # dec(ct) xor enc_prefix
        last_block += pt.to_bytes(1, 'big')
    return binascii.hexlify(msg_start + last_block)
```

And by sending this to the server we get: `hxp{PLz_us3_7h3_Ri9h7_PRiM1TiV3z}`

###PL version

W zadaniu dostajemy kod źródłowy używający AESa CBC. Kod jest dość krótki i zrozumiały:

```
pad = lambda m: m + bytes([16 - len(m) % 16] * (16 - len(m) % 16))
def haggis(m):
    crypt0r = AES.new(bytes(0x10), AES.MODE_CBC, bytes(0x10))
    return crypt0r.encrypt(len(m).to_bytes(0x10, 'big') + pad(m))[-0x10:]

target = os.urandom(0x10)
print(binascii.hexlify(target).decode())

msg = binascii.unhexlify(input())

if msg.startswith(b'I solemnly swear that I am up to no good.\0') \
        and haggis(msg) == target:
    print(open('flag.txt', 'r').read().strip())
```

Serwer losuje 16 bajtów i wysyła je do nas. Następnie musimy odesłać wiadomość z zadanym prefixem. Ta wiadomość jest sklejana z długością wiadomości i następnie szyfrowana, a ostatni block ciphertextu musi być równy wylosowanym 16 bajtom które dostaliśmy.

Wiemy że to AES CBC, wiemy że klucz to same `0x0` i tak samo `IV` to same `0x0`. Wiemy też że jest tam padding PKCS7.

Zacznijmy od dopełnienia bloku z prefixem do 16 bajtów. Zawsze wygodniej pracuje się na pełnych blokach:

```
msg_start = b'I solemnly swear that I am up to no good.\x00\x00\x00\x00\x00\x00\x00'
```

Dodamy jeszcze jeden blok za tym prefixem. Mając to na uwadze obliczamy długość pełnej wiadomości i tworzymy blok z długością tak samo jak zrobi to serwer:

```
len_prefix = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00@'
```

W ten sposób wiemy dokładnie co serwer będzie szyfrował.

Warto rozumieć jak działa tryb CBC: każdy blok przed szyfrowaniem jest XORowany z ciphertextem poprzedniego bloku (a pierwszy blok z IV). To oznacza że jeśli znamy ciphertext zakodowanych K bloków, możemy zakodować dodatkowe bloku po prostu poprzez ustawienie jako IV ostatniego bloku znanego ciphertextu. Wykorzystamy tutaj tą własność!

Najpierw obliczmy ciphtextex pierwszyh K bloków (dla ścisłości 2 bloków - bloku z długością wiadomości oraz z prefixem):

```
encryptor = AES.new(bytes(0x10), AES.MODE_CBC, bytes(0x10))
prefix_encrypted_block = encryptor.encrypt(len_prefix + msg_start)[-16:]
```

Potrzebujemy tylko ostatni blok ponieważ tylko on jest wykorzystywany w szyfrowaniu naszego przygotowywanego bloku poprzez XORowanie z nim. Musimy pamiętać że mamy tutaj padding PKCS7 więc w jeśli dodamy pełny blok to ostatni blok szyfrogramu będzie zaszyfrowanym paddingiem! Więc w rzeczywistości chcemy żeby to padding zakodował się do oczekiwanych wylosowanych 16 bajtów. Wiemy ile wynoszą bajty paddingu - wszystkie będą `0x10`, ale są xorowane z ciphertextem naszego przygotowywanego boku. Oznaczmy szyfrogram tego bloku jako `CT_payload` a jego wersje odszyfrowaną jako `payload`.

Popatrzmy co chcemy osiągnąć:

Chcemy dostać: `encrypt(CT_payload xor padding) = target` więc deszyfrując obustronnie:

`CT_payload xor padding = decrypt(target)`

a ponieważ xor dwa razy przez tą samą wartość się znosi:

`CT_payload = decrypt(target) xor padding`

Popatrzmy teraz skąd bierze się `CT_payload`:

`CT_payload = encrypt(payload xor prefix_encrypted_block)`

i deszyfrując obustronnie:

`decrypt(CT_payload) = payload xor prefix_encrypted_block`

więc:

`payload = decrypt(CT_payload) xor prefix_encrypted_block`

I jeśli teraz połączymy te dwa równania mamy:

`payload = decrypt(decrypt(target) xor padding) xor prefix_encrypted_block`

I w ten sposób uzyskaliśmy przepis na wyliczenie bajtów payloadu do wysłania. Implementujemy to w pythonie:

```
def solve_for_target(target):
    # enc(ct xor padding) = target
    # ct xor padding = dec(target)
    # ct = dec(target) xor padding
    # ct = enc (pt xor enc_prefix)
    # dec(ct) = pt xor enc_prefix
    # pt = dec(ct) xor enc_prefix
    target = binascii.unhexlify(target)
    encryptor = AES.new(bytes(0x10), AES.MODE_CBC, bytes(0x10))
    data = encryptor.decrypt(target)[-16:]  # ct xor padding
    last_block = b''
    expected_ct_bytes = b''
    for i in range(len(data)):
        expected_ct = (data[i] ^ 0x10)  # ct
        expected_ct_byte = expected_ct.to_bytes(1, 'big')
        expected_ct_bytes += expected_ct_byte
    encryptor = AES.new(bytes(0x10), AES.MODE_CBC, bytes(0x10))
    result_bytes = encryptor.decrypt(expected_ct_bytes)  # dec(ct)
    for i in range(len(result_bytes)):
        pt = result_bytes[i] ^ prefix_encrypted_block[i]  # dec(ct) xor enc_prefix
        last_block += pt.to_bytes(1, 'big')
    return binascii.hexlify(msg_start + last_block)
```

I po wysłaniu na serwer dostajemy: `hxp{PLz_us3_7h3_Ri9h7_PRiM1TiV3z}`