Homework #2

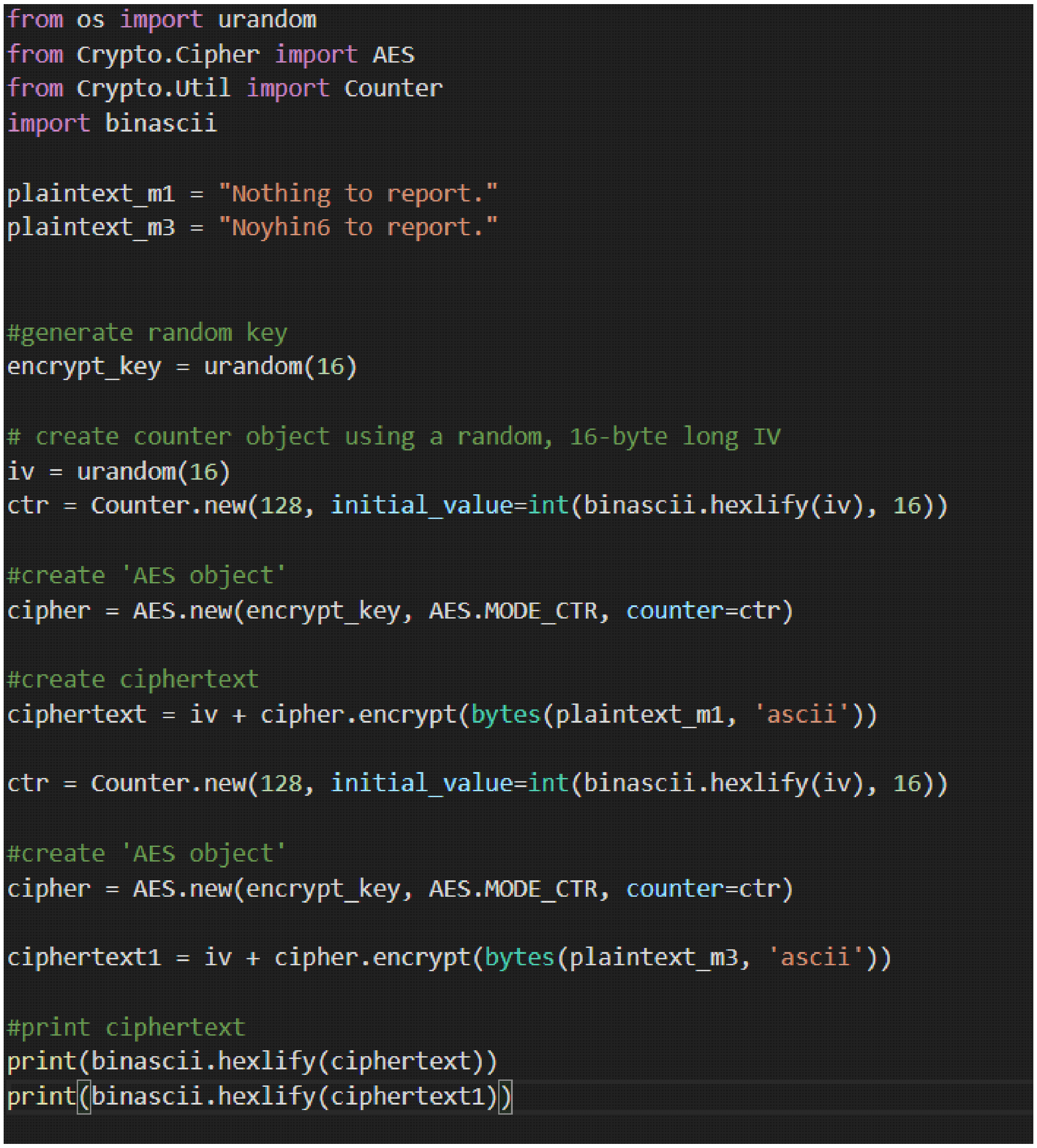
Tuesday, October 16, 2018 8:11 PM

Andrew Laux
andrew_laux@ucsb.edu

Task 1 -

a)

With identical IV's and encryption keys the two encrypted plaintexts only differ by one bit in the in the first byte of plaintext. We also know that at least one of the messages was not empty. I verified this running a modified version of the CRT AES encryption provided in the class slides. This information violates the definition of semantic security which states that we should know nothing about the information being encrypted. However, this is only the case because identical IVs were used. If random IV's were used, as in the proper use of counter mode encryption, the chance that you could correctly attribute which of the two messages was sent with which IV is no greater than 50% which is secure.



```
from os import urandom
from Crypto.Cipher import AES
from Crypto.Util import Counter
import binascii

plaintext_m1 = "Nothing to report."
plaintext_m3 = "Noyhin6 to report."


#generate random key
encrypt_key = urandom(16)

# create counter object using a random, 16-byte long IV
iv = urandom(16)
ctr = Counter.new(128, initial_value=int(binascii.hexlify(iv), 16))

#create 'AES object'
cipher = AES.new(encrypt_key, AES.MODE_CTR, counter=ctr)

#create ciphertext
ciphertext = iv + cipher.encrypt(bytes(plaintext_m1, 'ascii'))

ctr = Counter.new(128, initial_value=int(binascii.hexlify(iv), 16))

#create 'AES object'
cipher = AES.new(encrypt_key, AES.MODE_CTR, counter=ctr)

ciphertext1 = iv + cipher.encrypt(bytes(plaintext_m3, 'ascii'))

#print ciphertext
print(binascii.hexlify(ciphertext))
print(binascii.hexlify(ciphertext1))
```

b)

I think you would have to have a situation like the one in task 5. If you have information about the plaintexts being encrypted and you are expecting a certain plaintext to be sent routinely seeing a different bit could potentially be all the information you need. This information can only gotten however because the sender is reusing keys and IV's.

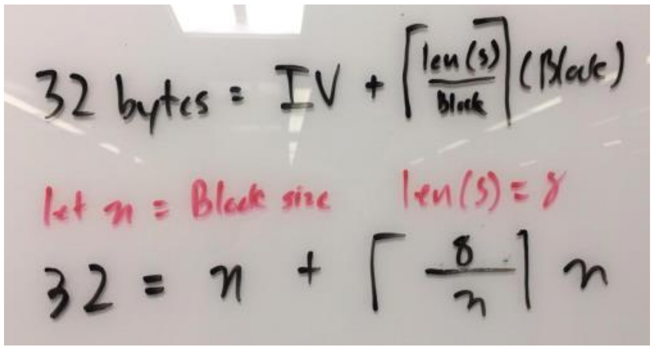
c)

With 3DES the block size is cut in half to 8 bytes or 64 bits. Such a reduction in length drastically reduces the domain of possible IV's. 2^{128} to 2^{64} is many orders of magnitude smaller. The chance that two randomly generated IV's are the same is now much greater similar to the probability of two birthdays being the same as in the birthday paradox.

Task 2 -

a)

Because we have no access to a padding oracle we cannot validate any modifications to the ciphertext. We do know for certain that the encrypted message is 8 bytes long and we know that we can modify those without interfering with the padding of the message. So safely modifying the ciphertext block is a matter of identifying which bytes correspond with the message.



$$32 \text{ bytes} = IV + \left\lceil \frac{\text{len}(s)}{\text{Block}} \right\rceil (\text{Block})$$

let n = Block size $\quad$ len(S) = 8

$$32 = n + \left\lceil \frac{8}{n} \right\rceil n$$

given the total size of the ciphertext and the size of the known message, a block size of 16 bytes is the only way to satisfy the above equation. Then we know that the message lives in the first 8 bytes of the second 16byte block. Therefore we can safely modify the first 8 bytes of the IV without creating an invalid padding.

b)

The set of valid paddings for a 16 byte block are elements of the set {01, 02, …10} For all these valid bytes the most significant 3 bits are all set to zero. Therefore I believe that, given the one to one nature of XOR operation, a simple bit flip on the last 8 bytes of the second to last block will cause the values in the padding region to be too high guaranteeing that the cipher will not decrypt.

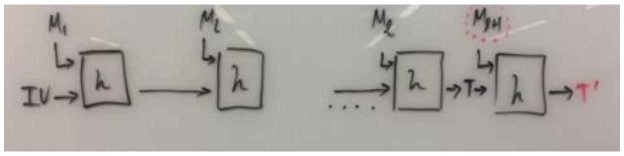
Task 3 -

a)

We can append an M_{l+1} to M such that M' = M || M_{l+1}

now we can get $T' = h(H_{l}, M')$ since h is a publicly known hash function and $h(H_{l}, M')$ doesn't require the Key.

Thus we have an M' != M and T' such that $MAC_k(M') = T'$



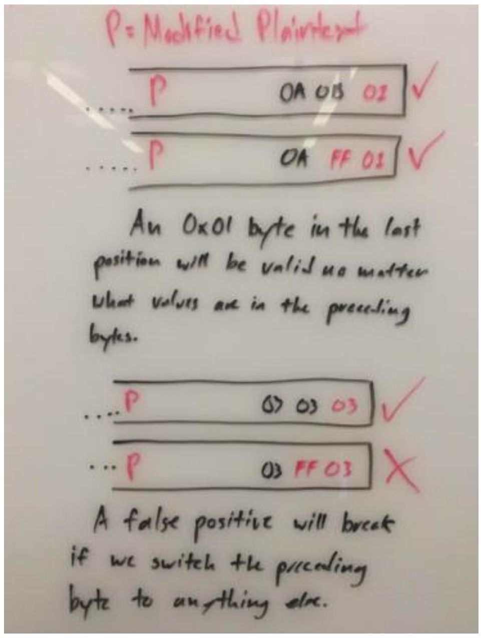
b)
HMAC solves this type of vulnerability by requiring the use of the key K at both stages of hashing. This makes it impossible to forge a tag like we did in part a.

Task 4 -

a)
When we perform X xor 0x01 we are finding some byte B such that X xor B = 0x01 because during decryption X is xor'd with B to unmask the original plaintext byte P. Both bytes B and P are unknown to us except when the oracle reports a valid decryption in which case P has been modified to 0x01 or another valid padding byte when the preceding bytes have the correct values(less likely), call these P_1 and P_2 respectively. So there are two correct values for modifying the last byte of the second to last cipher text.
X xor P1 = B1 and X xor P2 = B2
Such that one B is the correct byte and the other a false positive.
Using property of XOR a^b=c and c^b=a :
B2 xor 0x01 = X2
B1 xor 0x01 = X1

So we have two values for X, one where X xor B actually evaluates to 0x01 and one that evaluates to some other valid byte.

b)



An 0x01 byte in the last position will be valid no matter what values are in the preceding bytes.

A false positive will break if we switch the preceding byte to anything else.

Therefore to validate that we have the correct bit we alter the preceding byte in the plaintext. If it's still valid then we have the correct byte B if not then we need to continue searching possible bytes.

c)
For the second to last byte we can set the final byte of the (second to last cipher) block to 0x02 which is trivial since we have unmasked the final byte of B. Then we set the second to last byte in the block to a value X' such that X' = X xor 0x02. This way by passing the padding check with an 0x0202 as the last two bites we can unmask the second to last byte of B. We can continue to extend this using 0x030303 0x04040404 … 0x1F (x16) to uncover all 16 bytes of the cipherblock. For the preceding block we can simply truncate the deciphered block and start over again. We need at most 2^8 checks for each byte. Since we can use previously uncovered bytes to expedite each search the time in line.

Task 5 -

a)

The data must be the same from both messages. The code appended to the end of both messages is identical, given SHA-256 guarantees that no collisions can occur, the only possibility is that the same plaintexts were MAC'd.\

b)

Semantic security states that we should not know whether a message has been sent before, the example here clearly demonstrates that the same message, unencrypted, will MAC to the same value which lets us identify uniquely messages we have seen before.

c)

Yes, there is no possibility that we can guess the correct K2 for the mac function or find some M' that Mac(M) = Mac(M')
the space of SHA-256 is too large, therefore T is still unforegeable.