Zlot.py

Write-up of a Hack.lu 2012 CTF challenge

```
nc devel.yobi.be 2053
```

This challenge has two stages.

1) Medium: Investigate the contents of a **saved game**.

2) Hard: Get 8 (**EIGHT**) bonus points.

Good luck!

Hints:

The probability to win a bet is very, very, very low.

You might think that the strategy for solving this challenge is placing a risky bet and restoring a saved game when the bet is lost. The chance to win like this is ZERO (0, Nada, Non, Niente, Nix, Nullo, None).

This is a crypto challenge.

Check the leaked part of the source code below

$ nc devel.yobi.be 2053

Welcome to the Internet ZlotMachine. **Enter 'T' for the Tutorial.**

Your current balance is 5 credits and 1 bonus

T

The Internet ZlotMachine works like a traditional slot machine.

To get you hooked, we will give you 5 (FIVE) credits every time you visit us.

Once you have connected, there are several options:

[…]

6) **Save game (Command: S)**

   This allows you to save the current game. You will get a string back

   that you are supposed to write down somewhere. Using this string later

   will allow you to resume your game, when you come back.

   We use our SAFEJSON *hint hint*

7) **Load game (Command: L<SAVESTRING>)**

   Allows you to reload a previously saved game.

Loading backup

```
def loadState(self, statestr):
    try:
   dec = decrypt(statestr.decode("base64"))
   if '\x00' in dec:
       self.sendLine('Error loading game: invalid characters')
       return
   try:
       state = json.loads(dec)
   except Exception, e:
       self.sendLine('Error loading game: ' + str(e))
       return
   self.credits = state['credits']
   self.bonus = state['bonus']
   if self.bonus > 8:
       #XXX A few lines got lost in here during our recovery
   self.sendLine('Restored state.')
   self.sendLine(self.msgs['BALANCE'] % (self.credits, self.bonus))
    except Exception, e:
   self.sendLine('Error loading game: ' + str(e))
```

Crypto

```
def encrypt(data, key=SECRET_KEY:
    iv = urandom(16)
    cipher_obj = AES.new(key, AES.MODE_CBC, iv)
    data = addPadding(data, 16)
    return iv+cipher_obj.encrypt(data)

def decrypt(data, key=SECRET_KEY):
    iv, data = data[:16], data[16:] # d'uh
    cipher_obj = AES.new(key, AES.MODE_CBC, iv)
    padded = cipher_obj.decrypt(data)
    return delPadding(padded, 16)
```

Padding

```
def addPadding(data, block_size):
    data_len = len(data)+1 # required for the last byte
    pad_len = (block_size - data_len ) %  block_size
    pad_string = '\xff' * (pad_len) # arbitrary bytes to fill up block, -1 for
last byte. generated below
    last_byte = struct.pack('<B', pad_len+1) # little-endian unsigned char,
tells how many arbitrary bytes we have to remove
    return ''.join([data, pad_string, last_byte])

def delPadding(data, block_size):
    pad_len = struct.unpack('<B', data[-1])[0]
    if pad_len > block_size or pad_len < 1:
        raise ValueError("Encryption Error, Invalid Padding :/")
    else:
        return data[:(len(data)-pad_len)]
```

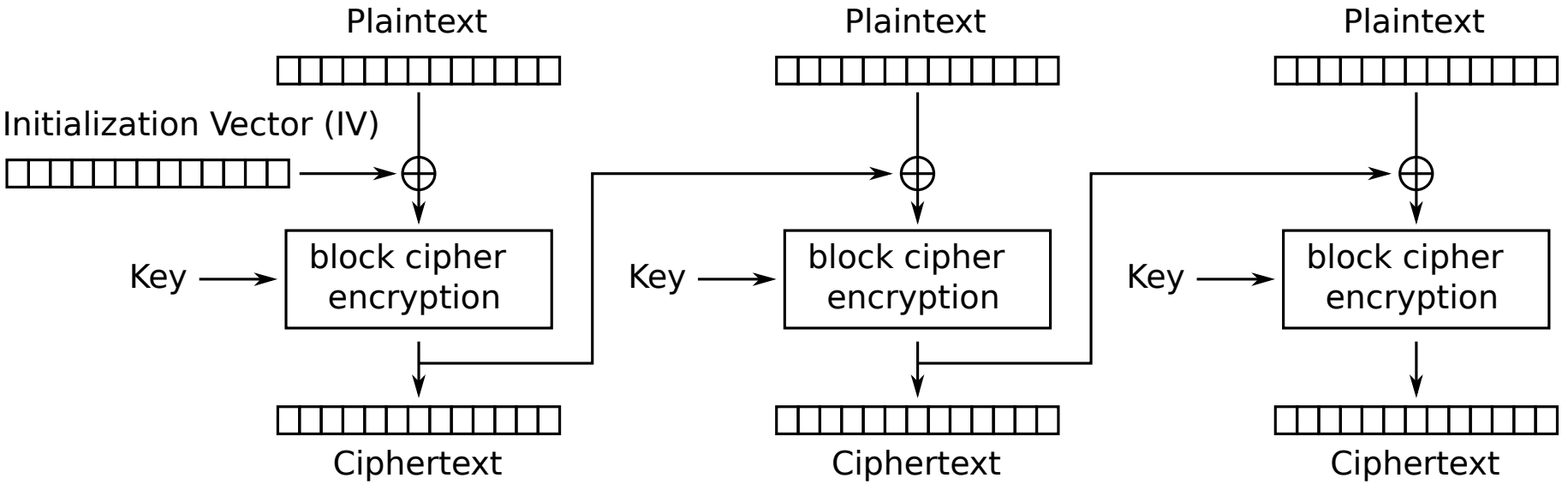
So a backup string is

a JSON string containing at least "bonus" and "credits" values, padded, encrypted with AES-CBC and encoded in base64.

Probably something like
{"credits": 5, "bonus":1, ???}

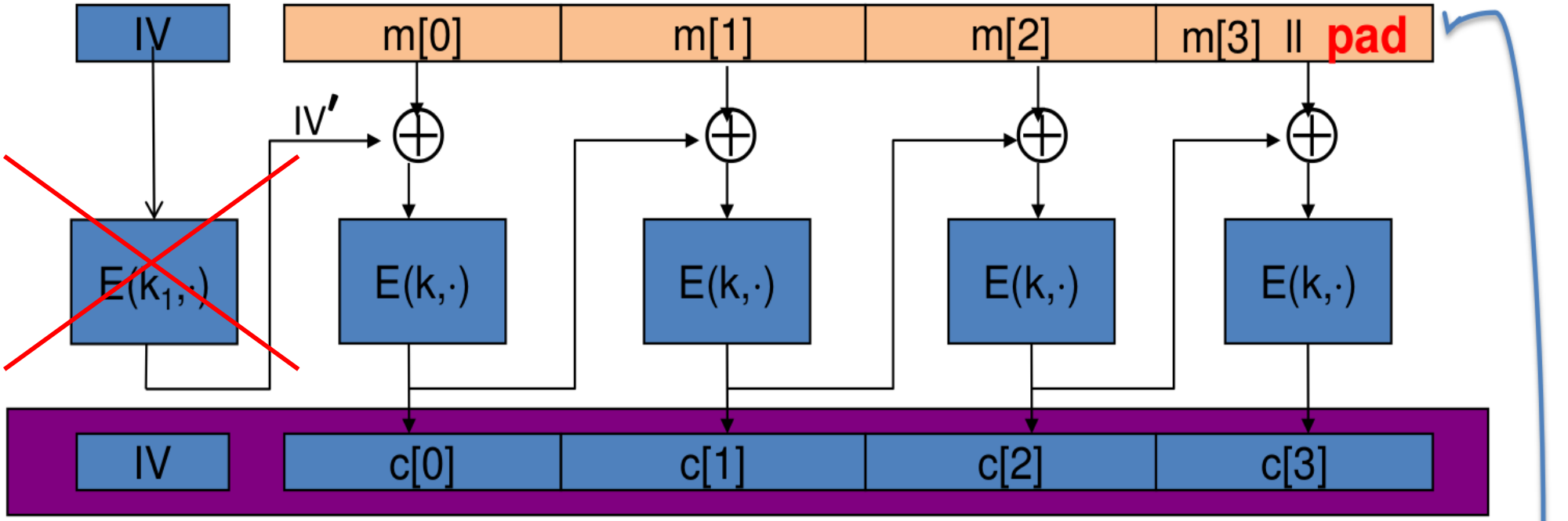
Ciphertext is 80 bytes (5 * 128 bits)

CBC @ encryption



Cipher Block Chaining (CBC) mode encryption

A CBC technicality: padding



TLS: for n>0, n byte pad is

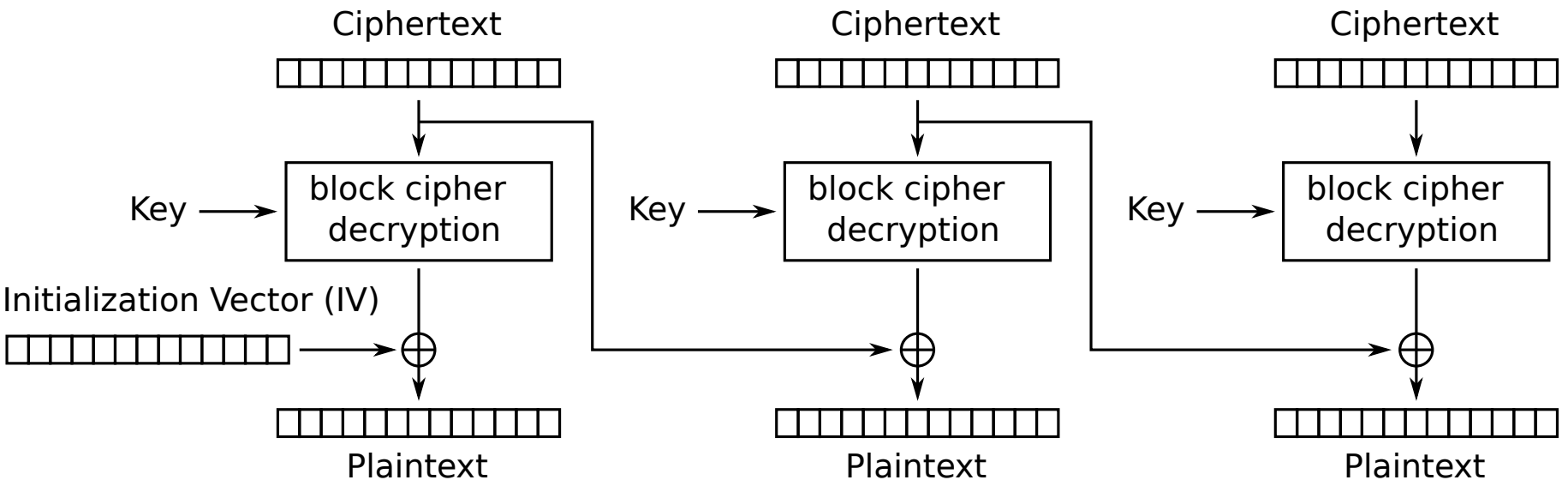
n	n	n	•••	n

if no pad needed, add a dummy block

removed during decryption

Dan Boneh

CBC @ decryption



Cipher Block Chaining (CBC) mode decryption

- Blah XOR IV = plaintext!!
  - But only for first block
- Flipping IV bits will directly flip bits in first block plaintext
  - Without affecting next blocks

Let's hope the value of "bonus" is indeed in that first block

Bonus in the saved game is 1 and we need 8.

'1' → '8'
== 0x31 → 0x38
== 0b00110001 → 0b00111000

We need to XOR that bonus byte with 0b00001001 == **0x9**

But we don't know where is that byte.

The earliest position it could get is with a JSON string starting with `{"bonus":1` so the 10th byte.

The furthest we can try is on byte 16, after that we're not in the IV anymore.

```
#!/usr/bin/env python
from base64 import b64decode, b64encode
m=b64decode("3SA7TH/9a6E4vgtY0MAuuMDCKd8nvKuI5/FMxHL0zsxmza8Gaudg0N
for i in range(10,17):
    print "L", b64encode(m[:i] + chr(ord(m[i])^0x9) + m[i+1:])
$ ./break-zlot.py
L 3SA7TH/9a6E4vgJY0MAuuMDCKd8nvKuI5/FMxHL0zsxmza8Gaudg0Nme9K97iRciC
L 3SA7TH/9a6E4vgtR0MAuuMDCKd8nvKuI5/FMxHL0zsxmza8Gaudg0Nme9K97iRciC
L 3SA7TH/9a6E4vgtY2cAuuMDCKd8nvKuI5/FMxHL0zsxmza8Gaudg0Nme9K97iRciC
L 3SA7TH/9a6E4vgtY0MkuuMDCKd8nvKuI5/FMxHL0zsxmza8Gaudg0Nme9K97iRciC
L 3SA7TH/9a6E4vgtY0MAnuMDCKd8nvKuI5/FMxHL0zsxmza8Gaudg0Nme9K97iRciC
L 3SA7TH/9a6E4vgtY0MAuscDCKd8nvKuI5/FMxHL0zsxmza8Gaudg0Nme9K97iRciC
L 3SA7TH/9a6E4vgtY0MAuuMnCKd8nvKuI5/FMxHL0zsxmza8Gaudg0Nme9K97iRciC
```

Let's try them

$ nc devel.yobi.be 2053

Welcome to the Internet ZlotMachine. Enter 'T' for the Tutorial.

Your current balance is 5 credits and 1 bonus

L 3SA7TH/9a6E4vgJY0MAuuMDCKd8nvKuI5/FMxHL0zsxmza8Gaudg0Nme9K97iRciC7LtFbrC5e5o8iP/1LBP5vwQcF7nl9OzrAtoWy23e/A=

Restored state.

Your current balance is 5 credits and **8 bonus**

Nice one. Here's your flag:
9eef8f17d07c4f11febcac1052469ab9

Decoding a full backup??

No way to brute-force the key

Padding oracle attacks...

First some basic tooling...

http://codepad.org/8DbVG66y

Remember Padding code

```
def addPadding(data, block_size):
    data_len = len(data)+1 # required for the last byte
    pad_len = (block_size - data_len ) %  block_size
    pad_string = '\xff' * (pad_len) # arbitrary bytes to fill up block, -1 for
last byte. generated below
    last_byte = struct.pack('<B', pad_len+1) # little-endian unsigned char,
tells how many arbitrary bytes we have to remove
    return ''.join([data, pad_string, last_byte])

def delPadding(data, block_size):
    pad_len = struct.unpack('<B', data[-1])[0]
    if pad_len > block_size or pad_len < 1:
        raise ValueError("Encryption Error, Invalid Padding :/")
    else:
        return data[:(len(data)-pad_len)]
```

- We cut the ciphertext to one single block (+IV)
- Last byte is padding length byte
- It must be between 1 and 16
- Testing all possibilities of last byte of IV
  → all possibilities of padding length byte

`padding_oracle-test1.py`

Usual padding oracle attacks

- Usual padding types:
  - ANSI X.923: `[...] 00 00 00 04`
  - PKCS7/TLS: `[...] 04 04 04 04`
  - ISO7816-4: `[...] 80 00 00 00`
  - Zlotpy: `[...] FF FF FF 04`
- Rely on integrity of the full padding bytes
  - e.g. here all padding bytes except last should be 0xFF, so only padding length=1 should work

But author doesn't care checking those bytes

```
def delPadding(data, block_size):
    pad_len = struct.unpack('<B', data[-1])[0]
    if pad_len > block_size or pad_len < 1:
        raise ValueError("Encryption Error, Invalid Padding :/")
    else:
        return data[:(len(data)-pad_len)]
```

So what?

```
if '\x00' in dec:
    self.sendLine('Error loading game: invalid characters')
```

Kind of secondary oracle

- When padding is valid (so 1>p>16) we try all possibilities for preceding block till we get an “invalid char” error.
  If so we know last two plaintext bytes have become 00 01 with our forged IV'.

- If padding length was longer, no invalid char possible as that byte would have been discarded anyway.

IV xor d(C0) = P0

IV' xor d(C0) = P'0 with last bytes 00 01

→ IV xor IV' xor d(C0) xor d(C0) = P0 xor P'0

→ IV[-2:] xor IV'[-2:] = P0[-2:] xor “0001”

→ P0[-2:] = IV[-2:] xor IV'[-2:] xor “0001”

`padding_oracle-test2.py`

Change IV' so P'0[-2:] becomes 0001 → 0002
IV'[-1] = IV'[-1] ^ 01 ^ 02

Try to trigger “invalid char” by manipulating IV'[-3]
When it happens, P'0[-3:] is now 000002

Etc

At the end we've an IV' for which P'0 = 0000000000000000000000000000000F
(one byte of data = 0x00 followed by padding)

→ P0 = IV xor IV' xor “00..0F”

→ P1 = C0 xor IV'' xor “00..0F”, etc

`padding_oracle-test3.py`

Usual padding oracle attacks

- ANSI X.923:
  - `[...]` `01`
  - `[...]` `0002`
  - `[...]` `000003`
  - `[...]` `00000004`
  - `0000000000000000000000000000000010`

Usual padding oracle attacks

- PKCS7/TLS:
  - `[...]` **`01`**
  - `[...]` **`0202`**
  - `[...]` **`030303`**
  - `[...]` **`04040404`**
  - **`10101010101010101010101010101010`**

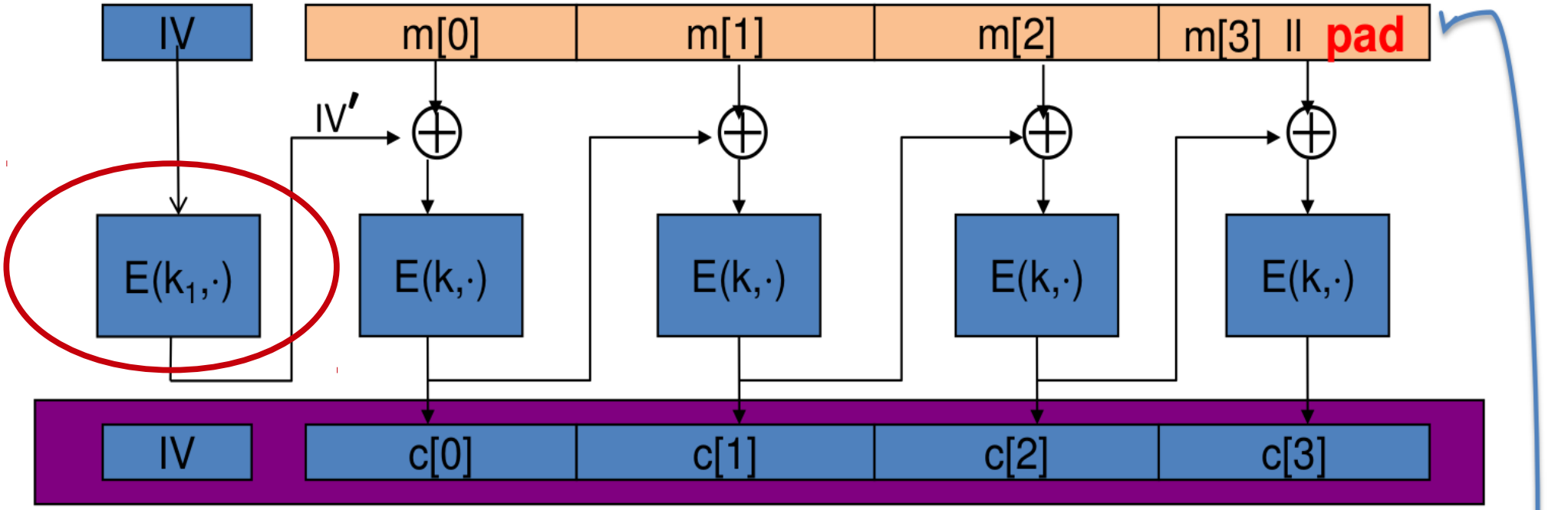
Usual padding oracle attacks

- ISO7816-4:
  - [...] 80
  - [...] 8000
  - [...] 800000
  - [...] 80000000
  - 8000000000000000000000000000000000

To recap, we've got

- Error messages allowing oracle attacks
- IV directly combined with plaintext
- AES-CBC alone provides confidentiality, not integrity!
- No MAC (Message Authentication Code)

A CBC technicality: padding



TLS: for n>0, n byte pad is

n	n	n	•••	n

if no pad needed, add a dummy block

removed during decryption

Thank you

- Thanks to Fluxfingers for their amazing CTFs
- Thanks to Frederik Braun for having created this one and having shared the source code so that you could try in live too