Using the Mega API, with Python examples!

Introduction

The new Mega has the great advantage of being built as a service that can be queried by any client through its API. That means that the community can build shiny new stunning software on top of Mega's API and take advantage of its huge capabilites.

The Mega's API is documented here, but since the project is still very young, some information might be missing if you want to develop your own client from scratch. Never mind, Mega had the great idea to open the source code of its website, so we have all that we need to start coding!

Let's talk a little bit about the API itself first. It is based on a simple HTTP/JSON request-response scheme, which makes it really easy to use. Requests are made by POSTing the JSON payload to this URL:

> *https://g.api.mega.co.nz/cs?id=sequence_number[&sid=session_id]*

Where *sequence_number* is a session-unique number incremented with each request, and *session_id* is a token identifying the user session.

The JSON payload is an array of commands:

[{'a': 'command1', 'param1': 'value1', 'param2': 'value2'}, {'a': 'command2', 'param1': 'value1', 'param2': 'value2'}]

We will only send one command per request, but we still need to put it in an array. The response is either a numeric error code or an array of per-command return objects (JSON-encoded). Since we only send one command, we will get back an array containing only one return object. Thus, we can write our first two functions.

We will use Python in all the following examples, because it's a very nice language that allows to experiment things quickly (and because I wanted to learn Python. These are my first steps, so you may see some ugly and un-pythonic things… please share all your suggestions for improvements in the comments! The good news is that if you're new to Python, you will likely understand all the code in this article without any problem 🙂). We will use PyCrypto for all the crypto-related parts.

```
seqno = random.randint(0, 0xFFFFFFFF)

def api_req(req):
  global seqno
  url = 'https://g.api.mega.co.nz/cs?id=%d%s' % (seqno, '&amp;sid=%s' % sid if sid else '')
  seqno += 1
  return json.loads(post(url, json.dumps([req])))[0]

def post(url, data):
  return urllib.urlopen(url, data).read()
```

You will notice that I'm not doing any kind of error checking ~~because I'm lazy~~ to keep the examples as simple as possible. The imports are not included, but you will find them in the complete listing at the end of this article. In the following, we will often need to base64 encode/decode data, and to convert byte strings to arrays of 32 bit integers and vice versa (for encryption and hash calculation). The utility functions that deal with this work are also given in the complete listing.

Now, we are ready to start!

Logging in

First, we need to log in. This will give us a session token to include in all subsequent requests, and the master key used to encrypt all node-specific keys. According to the Mega's developer guide:

Each user account uses a symmetric master key to ECB-encrypt all keys of the nodes it keeps in its own trees. This master key is stored on MEGA's servers, encrypted with a hash derived from the user's login password.

Each login starts a new session. For complete accounts, this involves the server generating a random session token and encrypting it to the user's private key. The user password is considered verified if it successfully decrypts the private key, which then successfully decrypts the session token.

To log in, we need to provide the server our email and a hash derived from our email and password. The hash is computed as follows (see stringhash() and prepare_key() in Mega's crypto.js, and postlogin() in Mega's login.js):

```
password_aes = prepare_key(str_to_a32(password))
uh = stringhash(email.lower(), password_aes)

def stringhash(s, aeskey):
  s32 = str_to_a32(s)
  h32 = [0, 0, 0, 0]
  for i in xrange(len(s32)):
    h32[i % 4] ^= s32[i]
  for _ in xrange(0x4000):
    h32 = aes_cbc_encrypt_a32(h32, aeskey)
  return a32_to_base64((h32[0], h32[2]))

def prepare_key(a):
  pkey = [0x93C467E3, 0x7DB0C7A4, 0xD1BE3F81, 0x0152CB56]
  for _ in xrange(0x10000):
    for j in xrange(0, len(a), 4):
      key = [0, 0, 0, 0]
      for i in xrange(4):
        if i + j &lt; len(a):
          key[i] = a[i + j]
      pkey = aes_cbc_encrypt_a32(pkey, key)
  return pkey
```

The *aes_cbc_encrypt_a32* function is given in the complete listing at the end of this article, as well as the ones dealing with base64 encoding and conversion between strings and integer arrays. Now that we have computed the hash, we can call the *us* method of the API:

```
res = api_req({'a': 'us', 'user': email, 'uh': uh})
```

The response contains 3 entries:

- csid: the session ID, encrypted with our RSA private key ;
- privk: our RSA private key, encrypted with our master key ;
- k: our master key, encrypted with the hash previoulsy computed.

All of them are base64-encoded. First, let's decrypt the master key:

```
enc_master_key = base64_to_a32(res['k'])
master_key = decrypt_key(enc_master_key, password_aes)
```

Then, we can decrypt our RSA private key:

```
enc_rsa_priv_key = base64_to_a32(res['privk'])
rsa_priv_key = decrypt_key(enc_rsa_priv_key, master_key)
```

The decryption is done by simply concatening all the decrypted AES blocks (see decrypt_key() in Mega's crypto.js). We are calling *aes_cbc_decrypt_a32()* but CBC doesn't matter here, since we are encrypting only one block (4 * 32 = 128 bits) each time.

```
def decrypt_key(a, key):
  return sum((aes_cbc_decrypt_a32(a[i:i+4], key) for i in xrange(0, len(a), 4)), ())
```

We now have to decompose it into its 4 components:

- p: The first factor of n, the RSA modulus ;
- q: The second factor of n ;
- d: The private exponent ;
- u: The CRT coefficient, equals to (1/p) mod q.

We will only need p, q and d. For more information about RSA, feel free to read this article on Wikipedia.

All the components are multiple precision integers (MPI), encoded as a string where the first two bytes are the length of the number in bits, and the following bytes are the number itself, in big endian order (see mpi2b() and b2mpi() in Mega's rsa.js).

It’s then easy to convert a MPI to a Python long integer:

```python
def mpi2int(s):
  return int(binascii.hexlify(s[2:]), 16)
```

We can now go back to our RSA private key decomposition:

```python
privk = a32_to_str(rsa_priv_key)
rsa_priv_key = [0, 0, 0, 0]

for i in xrange(4):
  l = ((ord(privk[0]) * 256 + ord(privk[1]) + 7) / 8) + 2;
  rsa_priv_key[i] = mpi2int(privk[:l])
  privk = privk[l:]
```

Finally, we can decrypt the session id:

```python
enc_sid = mpi2int(base64urldecode(res['csid']))
decrypter = RSA.construct((rsa_priv_key[0] * rsa_priv_key[1], 0L, rsa_priv_key[2], rsa_priv_key[0], rsa_priv_key[1]))
sid = '%x' % decrypter.key._decrypt(enc_sid)
sid = binascii.unhexlify('0' + sid if len(sid) % 2 else sid)
sid = base64urlencode(sid[:43])
```

PyCrypto uses a blinding step that involves *e*, the public exponent of the RSA key, during the decryption. Since we don’t know *e*, we simply bypass this step by calling *key._decrypt()* from PyCrypto’s private API. The final sid is the base64 encoding of the first 43 characters of the decrypted csid (see api_getsid2() in Mega’s crypto.js).

We now have all that we need to query the API… so let’s get the list of our files!

Listing the files

First, let’s quote the Mega’s developer reference about their storage model:

> *MEGA’s filesystem uses the standard hierarchical file/folder paradigm. Each file and folder node points to a parent folder node, with the exception of three parent-less root folder nodes per user account – one for his personal files, one inbox for secure unauthenticated file delivery, and one rubbish bin.*
>
> *Each general filesystem node (files/folders) has an encrypted attributes object attached to it, which typically contains just the filename, but will soon be used to transport user-to-user messages to augment MEGA’s secure online collaboration capabilities.*

We can retrieve the list of all our nodes by calling the API *f* method:

```
files = api_req({'a': 'f', 'c': 1})
```

The result contains, for each node, the the following informations:

- h: The ID of the node ;
- p: The ID of the parent node (directory) ;
- u: The owner of the node ;
- t: The type of the node:
    - 0: File
    - 1: Directory
    - 2: Special node: Root (“Cloud Drive”)
    - 3: Special node: Inbox
    - 4: Special node: Trash Bin
- a: The attributes of the node. Currently only contains its name.
- k: The key of the node (used to encrypt its content and its attributes) ;
- s: The size of the node ;
- ts: The time of the last modification of the node.

Let’s talk a little more about the key. As explained by the Mega developer’s guide:

> *All symmetric cryptographic operations are based on AES-128. It operates in cipher block chaining mode for the file and folder attribute blocks and in counter mode for the actual file data. Each file and each folder node uses its own randomly generated 128 bit key. File nodes use the same key for the attribute block and the file data, plus a 64 bit random counter start value and a 64 bit meta MAC to verify the file's integrity.*

So, for directory nodes, the key *key* is just a 128 bit AES key used to encrypt the attributes of the directory (for now, just its name). But for file nodes, *key* is 256 bits long and actually contains 3 components. If we see *key* as a list of 8 32 bit integers, then:

- *(key[0] XOR key[4], key[1] XOR key[5], key[2] XOR key[6], key[3] XOR key[7])* is the 128 bit AES key *k* used to encrypt the file contents and its attributes ;
- *(key[4], key[5])* is the initialization vector for AES-CTR, that is, the upper 64 bit *n* of the counter start value used to encrypt the file contents. The lower 64 bit are starting at 0 and incrementing by 1 for each AES block of 16 bytes.
- *(key[6], key[7])* is a 64 bit meta-MAC *m* for file integrity.

Now, we have all the keys to list the names of our files! First, let's write a function to decrypt file attributes. They are JSON-encoded (e.g. *{'n': 'filename.ext'}*), prefixed with the string "MEGA" (*MEGA{'n': 'filename.ext'}*):

```
def dec_attr(attr, key):
  attr = aes_cbc_decrypt(attr, a32_to_str(key)).rstrip('\0')
  return json.loads(attr[4:]) if attr[:6] == 'MEGA{"' else False
```

Then, our main loop:

```
for file in files['f']:
  if file['t'] == 0 or file['t'] == 1:
    key = file['k'][file['k'].index(':') + 1:]
    key = decrypt_key(base64_to_a32(key), master_key)
    if file['t'] == 0: # File
      k = (key[0] ^ key[4], key[1] ^ key[5], key[2] ^ key[6], key[3] ^ key[7])
      iv = key[4:6] + (0, 0)
      meta_mac = key[6:8]
    else: # Directory
      k = key
    attributes = base64urldecode(file['a'])
    attributes = dec_attr(attributes, k)
    print attributes['n']
  elif file['t'] == 2:
    root_id = file['h'] # Root ("Cloud Drive")
  elif file['t'] == 3:
    inbox_id = file['h'] # Inbox
  elif file['t'] == 4:
    trashbin_id = file['h'] # Trash Bin
```

Ta-dah! We are now able to list all our files, and decrypt their names.

Downloading a file

To download a file, we first need to get a temporary download URL for this file from the API. This is done with the *g* method of the API:

```
dl_url = api_req({'a': 'g', 'g': 1, 'n': file['h']})['g']
```

A simple GET request on this URL will give us the encrypted file. We can either download the whole file first, and then decrypt it, or decrypt it on the fly during the download. The latter seems to be the best solution if we want to check the file's integrity, since the MAC has to be computed chunk by chunk:

> *File integrity is verified using chunked CBC-MAC. Chunk sizes start at 128 KB and increase to 1 MB, which is a reasonable balance between space required to store the chunk MACs and the average overhead for integrity-checking partial reads.*

According to the developer's guide, chunk boundaries are located at the following positions:

> *0 / 128K / 384K / 768K / 1280K / 1920K / 2688K / 3584K / 4608K / … (every 1024 KB) / EOF*

And a chunk MAC is computed as follows:

h := (n << 64) + n // Reminder: n = 64 upper bits of the counter start value

For each AES block d: h := AES(k,h XOR d)

The whole file MAC is obtained by applying the same algorithm to the resulting block MACs, with a start value of 0. The 64 bit meta-MAC is then defined as:

((bits 0-31 XOR bits 32-63) << 64) + (bits 64-95 XOR bits 96-127)

We now have all that we need to download a file, so… let's go! The *get_chunks()* function is given in the complete listing. It simply gives the list of chunks for a given size, according to the specification discussed above. Since it actually returns a dict *{chunk_start: chunk_length}* of all the chunks, we need to iterate over it in sorted order.

```
infile = urllib.urlopen(dl_url)
outfile = open(attributes['n'], 'wb')
decryptor = AES.new(a32_to_str(k), AES.MODE_CTR, counter = Counter.new(128, initial_value = ((iv[0] &lt;&lt; 32) + iv[1]) &lt;&lt; 64))

file_mac = [0, 0, 0, 0]
for chunk_start, chunk_size in sorted(get_chunks(file['s']).items()):
  chunk = infile.read(chunk_size)
  # Decrypt and upload the chunk
  chunk = decryptor.decrypt(chunk)
  outfile.write(chunk)

  # Compute the chunk's MAC
  chunk_mac = [iv[0], iv[1], iv[0], iv[1]]
  for i in xrange(0, len(chunk), 16):
    block = chunk[i:i+16]
    if len(block) % 16:
      block += '\0' * (16 - (len(block) % 16))
    block = str_to_a32(block)
    chunk_mac = [chunk_mac[0] ^ block[0], chunk_mac[1] ^ block[1], chunk_mac[2] ^ block[2], chunk_mac[3] ^ block[3]]
    chunk_mac = aes_cbc_encrypt_a32(chunk_mac, k)
  # Update the file's MAC
  file_mac = [file_mac[0] ^ chunk_mac[0], file_mac[1] ^ chunk_mac[1], file_mac[2] ^ chunk_mac[2], file_mac[3] ^ chunk_mac[3]]
  file_mac = aes_cbc_encrypt_a32(file_mac, k)

outfile.close()
infile.close()

# Integrity check
if (file_mac[0] ^ file_mac[1], file_mac[2] ^ file_mac[3]) != meta_mac:
  print "MAC mismatch"
```

We can now list our files and download them. How about adding new files?

Uploading a file

Uploading a file requires two steps. First, we need to request a upload URL, which is done by calling the *u* method of the API and requires to specify the file size:

```
infile = open(filename, 'rb')
size = os.path.getsize(filename)
ul_url = api_req({'a': 'u', 's': size})['p']
```

We can then generate a random 128 bit AES key for the file, and the upper 64 bits of the counter start value (initialization vector). With these two values, we can encrypt the file and start the upload by simply POSTing the file contents to the upload URL!

The upload is done chunk by chunk, in order to compute on the fly the chunk MACs that we will need later to get the meta-MAC. To upload the chunk starting at offset *x*, we simply append */x* to the upload URL.

```
infile = open(filename, 'rb')
size = os.path.getsize(filename)
ul_url = api_req({'a': 'u', 's': size})['p']

ul_key = [random.randint(0, 0xFFFFFFFF) for _ in xrange(6)]
encryptor = AES.new(a32_to_str(ul_key[:4]), AES.MODE_CTR, counter = Counter.new(128, initial_value = ((ul_key[4] &lt;&lt; 32) + ul_key[5

file_mac = [0, 0, 0, 0]
for chunk_start, chunk_size in sorted(get_chunks(size).items()):
  chunk = infile.read(chunk_size)

```

```
    # Compute the chunk's MAC
    chunk_mac = [ul_key[4], ul_key[5], ul_key[4], ul_key[5]]
    for i in xrange(0, len(chunk), 16):
      block = chunk[i:i+16]
      if len(block) % 16:
        block += '\0' * (16 - len(block) % 16)
      block = str_to_a32(block)
      chunk_mac = [chunk_mac[0] ^ block[0], chunk_mac[1] ^ block[1], chunk_mac[2] ^ block[2], chunk_mac[3] ^ block[3]]
      chunk_mac = aes_cbc_encrypt_a32(chunk_mac, ul_key[:4])

    # Update the file's MAC
    file_mac = [file_mac[0] ^ chunk_mac[0], file_mac[1] ^ chunk_mac[1], file_mac[2] ^ chunk_mac[2], file_mac[3] ^ chunk_mac[3]]
    file_mac = aes_cbc_encrypt_a32(file_mac, ul_key[:4])

    # Encrypt and upload the chunk
    chunk = encryptor.encrypt(chunk)
    outfile = urllib.urlopen(ul_url + "/" + str(chunk_start), chunk)
    completion_handle = outfile.read()
    outfile.close()

  infile.close()

  # Compute the meta-MAC
  meta_mac = (file_mac[0] ^ file_mac[1], file_mac[2] ^ file_mac[3])
```

Now that the upload is done, we have to actually create the new node on our filesystem. Notice that we saved the response of the POST to the upload URL: it is a completion handle that we will give to the API to create a new node corresponding to the completed upload.

This is done by calling the *p* method of the API. It requires:

- The ID of the target node (the parent directory of our new node) ;
- The completion handle discussed above ;
- The type of the new node (0 for a file) ;
- The attributes of the new node (for now, just its name), encrypted with the node key ;
- The key of the node (encrypted with the master key), in the format discussed in the previous section, which means we need to XOR the key randomly generated above with the initialization vector and the meta-MAC.

So we first need two functions: one to encrypt the attributes (analogous to *dec_attr()* defined before), and the other to encrypt the key (similar to *decrypt_key()*):

```
def enc_attr(attr, key):
  attr = 'MEGA' + json.dumps(attr)
  if len(attr) % 16: # Add padding for AES encryption
    attr += '\0' * (16 - len(attr) % 16)
  return aes_cbc_encrypt(attr, a32_to_str(key))

def encrypt_key(a, key):
  return sum((aes_cbc_encrypt_a32(a[i:i+4], key) for i in xrange(0, len(a), 4)), ())
```

We can now create the new node:

```
attributes = {'n': os.path.basename(filename)}
enc_attributes = enc_attr(attributes, ul_key[:4])
key = [ul_key[0] ^ ul_key[4], ul_key[1] ^ ul_key[5], ul_key[2] ^ meta_mac[0], ul_key[3] ^ meta_mac[1], ul_key[4], ul_key[5], meta_mac[0]
api_req({'a': 'p', 't': root_id, 'n': [{'h': completion_handle, 't': 0, 'a': base64urlencode(enc_attributes), 'k': a32_to_base64(encrypt_
```

The API confirms the creation of the new node by returning all the informations given in the previous section ("Listing the files"): ID, parent ID, owner, type, attributes, key, size and last modification time (creation time in our case). The new file now appears in the list of our files. We are all done!

Conclusion

We have seen that with a few lines of code, we can build our own Mega client pretty quickly. I'm currently working on a FUSE filesystem, to mount Mega on Linux, and will share it shortly on GitHub. But in the meantime, here is the complete listing for all the examples of this article. Hope you liked it!

```
from Crypto.Cipher import AES
from Crypto.PublicKey import RSA
from Crypto.Util import Counter

import base64
import binascii
import json
import os
import random
import struct
import sys
import urllib

sid = ''
```

```python
seqno = random.randint(0, 0xFFFFFFFF)

master_key = ''
rsa_priv_key = ''

def base64urldecode(data):
  data += '=='[(2 - len(data) * 3) % 4:]
  for search, replace in (('-', '+'), ('_', '/'), (',', '')):
    data = data.replace(search, replace)
  return base64.b64decode(data)

def base64urlencode(data):
  data = base64.b64encode(data)
  for search, replace in (('+', '-'), ('/', '_'), ('=', '')):
    data = data.replace(search, replace)
  return data

def a32_to_str(a):
  return struct.pack('&gt;%dI' % len(a), *a)

def a32_to_base64(a):
  return base64urlencode(a32_to_str(a))

def str_to_a32(b):
  if len(b) % 4: # Add padding, we need a string with a length multiple of 4
    b += '\0' * (4 - len(b) % 4)
  return struct.unpack('&gt;%dI' % (len(b) / 4), b)

def base64_to_a32(s):
  return str_to_a32(base64urldecode(s))

def aes_cbc_encrypt(data, key):
  encryptor = AES.new(key, AES.MODE_CBC, '\0' * 16)
  return encryptor.encrypt(data)

def aes_cbc_decrypt(data, key):
  decryptor = AES.new(key, AES.MODE_CBC, '\0' * 16)
  return decryptor.decrypt(data)

def aes_cbc_encrypt_a32(data, key):
  return str_to_a32(aes_cbc_encrypt(a32_to_str(data), a32_to_str(key)))

def aes_cbc_decrypt_a32(data, key):
  return str_to_a32(aes_cbc_decrypt(a32_to_str(data), a32_to_str(key)))
```

```
def stringhash(s, aeskey):
  s32 = str_to_a32(s)
  h32 = [0, 0, 0, 0]
  for i in xrange(len(s32)):
    h32[i % 4] ^= s32[i]
  for _ in xrange(0x4000):
    h32 = aes_cbc_encrypt_a32(h32, aeskey)
  return a32_to_base64((h32[0], h32[2]))

def prepare_key(a):
  pkey = [0x93C467E3, 0x7DB0C7A4, 0xD1BE3F81, 0x0152CB56]
  for _ in xrange(0x10000):
    for j in xrange(0, len(a), 4):
      key = [0, 0, 0, 0]
      for i in xrange(4):
        if i + j &lt; len(a):
          key[i] = a[i + j]
      pkey = aes_cbc_encrypt_a32(pkey, key)
  return pkey

def encrypt_key(a, key):
  return sum((aes_cbc_encrypt_a32(a[i:i+4], key) for i in xrange(0, len(a), 4)), ())

def decrypt_key(a, key):
  return sum((aes_cbc_decrypt_a32(a[i:i+4], key) for i in xrange(0, len(a), 4)), ())

def mpi2int(s):
  return int(binascii.hexlify(s[2:]), 16)

def api_req(req):
  global seqno
  url = 'https://g.api.mega.co.nz/cs?id=%d%s' % (seqno, '&amp;sid=%s' % sid if sid else '')
  seqno += 1
  return json.loads(post(url, json.dumps([req])))[0]

def post(url, data):
  return urllib.urlopen(url, data).read()

def login(email, password):
  global sid, master_key, rsa_priv_key
  password_aes = prepare_key(str_to_a32(password))
  uh = stringhash(email.lower(), password_aes)
  res = api_req({'a': 'us', 'user': email, 'uh': uh})
```

```python
  enc_master_key = base64_to_a32(res['k'])
  master_key = decrypt_key(enc_master_key, password_aes)
  if 'tsid' in res:
    tsid = base64urldecode(res['tsid'])
    if a32_to_str(encrypt_key(str_to_a32(tsid[:16]), master_key)) == tsid[-16:]:
      sid = res['tsid']
  elif 'csid' in res:
    enc_rsa_priv_key = base64_to_a32(res['privk'])
    rsa_priv_key = decrypt_key(enc_rsa_priv_key, master_key)

    privk = a32_to_str(rsa_priv_key)
    rsa_priv_key = [0, 0, 0, 0]

    for i in xrange(4):
      l = ((ord(privk[0]) * 256 + ord(privk[1]) + 7) / 8) + 2;
      rsa_priv_key[i] = mpi2int(privk[:l])
      privk = privk[l:]

    enc_sid = mpi2int(base64urldecode(res['csid']))
    decrypter = RSA.construct((rsa_priv_key[0] * rsa_priv_key[1], 0L, rsa_priv_key[2], rsa_priv_key[0], rsa_priv_key[1]))
    sid = '%x' % decrypter.key._decrypt(enc_sid)
    sid = binascii.unhexlify('0' + sid if len(sid) % 2 else sid)
    sid = base64urlencode(sid[:43])

def enc_attr(attr, key):
  attr = 'MEGA' + json.dumps(attr)
  if len(attr) % 16:
    attr += '\0' * (16 - len(attr) % 16)
  return aes_cbc_encrypt(attr, a32_to_str(key))

def dec_attr(attr, key):
  attr = aes_cbc_decrypt(attr, a32_to_str(key)).rstrip('\0')
  return json.loads(attr[4:]) if attr[:6] == 'MEGA{"' else False

def get_chunks(size):
  chunks = {}
  p = pp = 0
  i = 1

  while i &lt;= 8 and p &lt; size - i * 0x20000:
    chunks[p] = i * 0x20000;
    pp = p
    p += chunks[p]
    i += 1

  while p &lt; size:
    chunks[p] = 0x100000;
    pp = p
    p += chunks[p]

  chunks[pp] = size - pp
  if not chunks[pp]:
    del chunks[pp]

  return chunks

def uploadfile(filename):
  infile = open(filename, 'rb')
  size = os.path.getsize(filename)
  ul_url = api_req({'a': 'u', 's': size})['p']

  ul_key = [random.randint(0, 0xFFFFFFFF) for _ in xrange(6)]
  encryptor = AES.new(a32_to_str(ul_key[:4]), AES.MODE_CTR, counter = Counter.new(128, initial_value = ((ul_key[4] &lt;&lt; 32) + ul_key

  file_mac = [0, 0, 0, 0]
  for chunk_start, chunk_size in sorted(get_chunks(size).items()):
    chunk = infile.read(chunk_size)

    chunk_mac = [ul_key[4], ul_key[5], ul_key[4], ul_key[5]]
    for i in xrange(0, len(chunk), 16):
      block = chunk[i:i+16]
      if len(block) % 16:
        block += '\0' * (16 - len(block) % 16)
      block = str_to_a32(block)
      chunk_mac = [chunk_mac[0] ^ block[0], chunk_mac[1] ^ block[1], chunk_mac[2] ^ block[2], chunk_mac[3] ^ block[3]]
      chunk_mac = aes_cbc_encrypt_a32(chunk_mac, ul_key[:4])

    file_mac = [file_mac[0] ^ chunk_mac[0], file_mac[1] ^ chunk_mac[1], file_mac[2] ^ chunk_mac[2], file_mac[3] ^ chunk_mac[3]]
    file_mac = aes_cbc_encrypt_a32(file_mac, ul_key[:4])

    chunk = encryptor.encrypt(chunk)
    outfile = urllib.urlopen(ul_url + "/" + str(chunk_start), chunk)
    completion_handle = outfile.read()
    outfile.close()

  infile.close()
  meta_mac = (file_mac[0] ^ file_mac[1], file_mac[2] ^ file_mac[3])

  attributes = {'n': os.path.basename(filename)}
  enc_attributes = enc_attr(attributes, ul_key[:4])
  key = [ul_key[0] ^ ul_key[4], ul_key[1] ^ ul_key[5], ul_key[2] ^ meta_mac[0], ul_key[3] ^ meta_mac[1], ul_key[4], ul_key[5], meta_mac[
  print api_req({'a': 'p', 't': root_id, 'n': [{'h': completion_handle, 't': 0, 'a': base64urlencode(enc_attributes), 'k': a32_to_base64

def downloadfile(file, attributes, k, iv, meta_mac):
  dl_url = api_req({'a': 'g', 'g': 1, 'n': file['h']})['g']

  infile = urllib.urlopen(dl_url)
  outfile = open(attributes['n'], 'wb')
  decryptor = AES.new(a32_to_str(k), AES.MODE_CTR, counter = Counter.new(128, initial_value = ((iv[0] &lt;&lt; 32) + iv[1]) &lt;&lt; 64)

  file_mac = [0, 0, 0, 0]
  for chunk_start, chunk_size in sorted(get_chunks(file['s']).items()):
    chunk = infile.read(chunk_size)
    chunk = decryptor.decrypt(chunk)
    outfile.write(chunk)

    chunk_mac = [iv[0], iv[1], iv[0], iv[1]]
    for i in xrange(0, len(chunk), 16):
      block = chunk[i:i+16]
      if len(block) % 16:
        block += '\0' * (16 - (len(block) % 16))
      block = str_to_a32(block)
      chunk_mac = [chunk_mac[0] ^ block[0], chunk_mac[1] ^ block[1], chunk_mac[2] ^ block[2], chunk_mac[3] ^ block[3]]
      chunk_mac = aes_cbc_encrypt_a32(chunk_mac, k)

    file_mac = [file_mac[0] ^ chunk_mac[0], file_mac[1] ^ chunk_mac[1], file_mac[2] ^ chunk_mac[2], file_mac[3] ^ chunk_mac[3]]
    file_mac = aes_cbc_encrypt_a32(file_mac, k)

  outfile.close()
  infile.close()

  if (file_mac[0] ^ file_mac[1], file_mac[2] ^ file_mac[3]) != meta_mac:
    print "MAC mismatch"

def getfiles():
  global root_id, inbox_id, trashbin_id

  files = api_req({'a': 'f', 'c': 1})
  for file in files['f']:
    if file['t'] == 0 or file['t'] == 1:
      key = file['k'][file['k'].index(':') + 1:]
      key = decrypt_key(base64_to_a32(key), master_key)
      if file['t'] == 0:
        k = (key[0] ^ key[4], key[1] ^ key[5], key[2] ^ key[6], key[3] ^ key[7])
        iv = key[4:6] + (0, 0)
        meta_mac = key[6:8]
      else:
        k = key
      attributes = base64urldecode(file['a'])
      attributes = dec_attr(attributes, k)
      print attributes['n']

      if file['h'] == '0wFEFCTa':
        downloadfile(file, attributes, k, iv, meta_mac)
    elif file['t'] == 2:
      root_id = file['h']
    elif file['t'] == 3:
      inbox_id = file['h']
    elif file['t'] == 4:
      trashbin_id = file['h']
```

Google+

This entry was posted in Uncategorized on January 28, 2013 [/web/20140527103321/http://julien-marchand.fr/blog/using-mega-api-with-python-examples/] .

40 thoughts on “Using the Mega API, with Python examples!”

lucho
January 29, 2013 at 4:36 am

This is very interesting! Thanks for sharing 😀

AoEmaster
January 29, 2013 at 11:07 am

Hi.

Can you please help me with the mega api ?
I'm trying to just get file size and file name with a knowed url. (in PHP)

Here is my code :
http://pastebin.com/RDwG6Pf6

I don't unterstand how to decrypt the "at" attribute of the response. Is it really the filename ?

It would very nice if you could help me

Julien Marchand Post author
January 29, 2013 at 1:11 pm

I think it's the attributes of the file, so yeap, its filename. You can decrypt it with the dec_attr() function

---

Julien Marchand Post author
January 29, 2013 at 1:14 pm

In PHP:

```
function dec_attr($attr, $key) {
  $b = trim(aes128_cbc_decrypt($attr, a32_to_str($key)));
  if (substr($b, 0, 6) != 'MEGA{"') return false;
  return json_decode(substr($b, 4));
}
```

And I use mcrypt for AES encryption/decryption:

```
function aes128_cbc_decrypt($data, $key) {
  return mcrypt_decrypt(
    MCRYPT_RIJNDAEL_128,
    $key, $data,
    MCRYPT_MODE_CBC,
    "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
  );
}
```

a32_to_str() is implemented with pack():

```
function a32_to_str($hex) {
  return call_user_func_array("pack", array_merge(array("N*"), $hex));
}
```

AoEmaster

January 29, 2013 at 5:36 pm

Je vois que tu as un .fr et que tu portes un nom bien français, donc je vais parler français vu que je le suis, et j'espère qu'on se comprendra :p

Déjà, vraiment désolé mais je n'ai jamais touché à python de ma vie et j'ai du mal à comprendre ton exemple et à faire le lien avec du PHP. J'aimerais réellement réussir à decrypter l'attribut mais je n'y arrive vraiment pas. Ma clef est stocké est stocké dans une chaine (la chaine contenu dans l'url) et je ne sais pas comment adapter ça avec le code que tu m'as fourni.

Je suis pas un pro mais bon, si tu pouvais essayer de prendre le temps de faire fonctionner mon script, je t'en serais vraiment reconnaissant car je commence à déséperer. Je m'embrouille trop avec ces fonctions de cryptages, base64, aes 12 bits etc…

Merci d'avance 😀

Julien Marchand Post author
January 29, 2013 at 6:23 pm

Bonne pioche 😀

Donc, tu as d'un côté l'ID d'un fichier, et de l'autre sa clé *key* déjà déchiffrée (à l'origine, elle est chiffrée avec la master key du propriétaire du fichier, mais elle est justement distribuée déchiffrée par son propriétaire lorsqu'il partage le fichier. C'est elle que l'on voit dans les liens Mega: https://mega.co.nz/#!file_hash!file_key).

Tu peux donc reprendre ce qui est fait par la boucle principale de la fonction *getfiles()* après l'étape de déchiffrement de la clé.

Tout d'abord, il faut extraire sa composante *k*, qui a servi a chiffrer le contenu du fichier et ses attributs: *k = [key[0] ^ key[4], key[1] ^ key[5], key[2] ^ key[6], key[3] ^ key[7]]*.

Ensuite, on déchiffre simplement les attributs grâce à cette clé (ils ont été chiffrés en AES-128 mode CBC). Ce qui donne, pour reprendre ton code :

```
function getLinkInfos($file_hash, $file_key) {
  $sequence_number = mt_rand(1, 99999999999);
  $ch = curl_init('https://g.api.mega.co.nz/cs?id='.$sequence_number);

  $data = array(array('a' => 'g', 'p' => $file_hash));
  $data_string = json_encode($data);

  curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
  curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
  curl_setopt($ch, CURLOPT_HTTPHEADER, array('Content-Type: application/json'));
  curl_setopt($ch, CURLOPT_VERBOSE, 1);
  curl_setopt($ch, CURLOPT_POST, true);
  curl_setopt($ch, CURLOPT_POSTFIELDS, $data_string);

  $output = curl_exec($ch);
  $res = json_decode($output, true);

  if (!isset($res[0]['s']) || !isset($res[0]['at'])) {
    return false;
  } else {
    $key = base64_to_a32($file_key);
    $k = array($key[0] ^ $key[4], $key[1] ^ $key[5], $key[2] ^ $key[6], $key[3] ^ $key[7]);

    $enc_attributes = base64urldecode($res[0]['at']);
    $attributes = dec_attr($enc_attributes, $k);

    return array('size' => $res[0]['s'], 'name' => $attributes->n);
  }
}
```

La clé et les attributs du fichiers n'étant rien d'autre que des données binaires (dont la représentation ASCII ne contient pas que des caractères affichables), ils ont été encodés en base64 avant d'être inclus dans la réponse de l'API, d'où la présence des fonctions *base64_to_a32()* et *base64urldecode()*.

Et voici les fonctions utilitaires utilisées (les mêmes que celles de l'article, mais implémentées en PHP) :

```
function a32_to_str($hex) {
  return call_user_func_array("pack", array_merge(array("N*"), $hex));
}

function str_to_a32($b) {
  return array_values(unpack('N*', $b));
}

function base64urldecode($data) {
  $data .= substr('==', (2 - strlen($data) * 3) % 4);
  $data = str_replace(array('-', '_', ','), array('+', '/', ''), $data);
  return base64_decode($data);
}

function base64_to_a32($s) {
  return str_to_a32(base64urldecode($s));
}
```

```
function dec_attr($attr, $key) {
  $b = trim(aes128_cbc_decrypt($attr, a32_to_str($key)));
  if (substr($b, 0, 6) != 'MEGA{"') return false;
  return json_decode(substr($b, 4));
}

function aes128_cbc_decrypt($data, $key) {
  return mcrypt_decrypt(
    MCRYPT_RIJNDAEL_128,
    $key, $data,
    MCRYPT_MODE_CBC,
    "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
  );
}
```

On obtient bien la taille et le nom du fichier 🙂

```
array(2) {
  ["size"]=>
  int(6468441)
  ["name"]=>
  string(25) "Darktek-Krize Kardiak.mp3"
}
```

AoEmaster

January 29, 2013 at 6:55 pm

OH MON DIEU MERCI !!!!!!!!!!!

Ca fonctionne enfin !!! des jours que je me bat. Je n’aurais jamais réussi seul.

Par contre, comment bien utiliser le sequence number ?

Julien Marchand Post author
January 29, 2013 at 10:10 pm

De rien 😉 Le sequence number doit être initialisé aléatoirement (mega.co.nz l'initialise entre 0 et 0xFFFFFFFF) et incrémenté à chaque requête. Plus précisément, à chaque requête *différente* : je ne fais ici aucune reprise sur erreur, mais si tu renvoies une requête qui a échoué (code d'erreur EAGAIN de l'API), il faut garder le même sequence number.

Robin Houtevelts
March 11, 2013 at 9:10 pm

Loving it!

Thanks for not only providing the code but also a great explanation!
You sir, are AWESOME!

Baron
January 29, 2013 at 11:45 am

You need to pad out the password with null characters to make its length a multiple of 4.

password = password.ljust(int(math.ceil(len(password) / 4.0)) * 4,”)
password_aes = prepare_key(str_to_a32(password))

Baron
January 29, 2013 at 11:47 am

The formatting has stripped out some characters
password = password.ljust(int(math.ceil(len(password) / 4.0)) * 4,”#backslash zero#”)

Julien Marchand Post author
January 29, 2013 at 12:45 pm

Thanks! My password length is already a multiple of 4, so I didn’t notice that… 😉 By the way, we also need it for the email. In fact, we need it for any string we pass to str_to_a32… so I added the padding there.

Baron
January 31, 2013 at 2:26 pm

Cheers 😃

Pingback: MegaFS, a FUSE filesystem wrapper for Mega. Part 1: Listing files. | Julien Marchand • Dev blog

foobar
January 29, 2013 at 7:55 pm

Files links can be shared to non-mega users via links with an embedded encryption key. Does the API support these non-user accounts? Are these the "ephemeral accounts" specified in the API docs? If so, how would your example be modified since those users don't have user/pass to authenticate with.

Julien Marchand Post author
January 29, 2013 at 10:08 pm

You are the second one to ask this question, so I just wrote a little article to explain that: http://julien-marchand.fr/blog/using-the-mega-api-how-to-download-a-public-file-or-a-file-you-know-the-key-without-logging-in/

It's just a matter of a parameter name in the API *g* method 😉 And it doesn't involve the ephemeral accounts mentionned in the API docs, you don't have to have an account at all (and no account is created for that).

Pingback: Using the Mega API: how to download a public file (or a file you know the key), without logging in | Julien Marchand • Dev blog

foobar
January 30, 2013 at 1:26 am

Thank you for your previous reply.

I noticed that you use urlopen().read() directly on the dl_url which downloads the entire file at once. Is there a reason you didn't download the chunks separately?

Julien Marchand Post author
February 6, 2013 at 12:33 am

Sorry for the late answer. I'm actually doing *infile = urllib.urlopen(dl_url)* and then *chunk = infile.read(chunk_size)*, so I'm reading the file chunk by chunk (I have to do that in order to compute the meta-MAC, because it's based on the chunk MACs. I could also download the whole file at once and then read it chunk by chunk to compute the meta-MAC, but it's just easier to directly download it chunk by chunk).

---

wink
February 1, 2013 at 4:07 pm

Bonjour Julien,

J'aimerai avoir, si posible une traduction des fonctions utilitaires que tu as fait en commentaire ci-dessous mais en langage Java. Je n'ai pas un niveau suffisant en php ou python pour lire/comprendre ces code.

Merci d'avance pour ton aide.

Julien Marchand Post author
February 6, 2013 at 12:34 am

Je vais essayer de trouver le temps d'en faire une version Java dans pas longtemps 🙂

---

j-muller
February 1, 2013 at 9:05 pm

Bonjour Julien,

Ton post est très instructif. J'aimerais toutefois savoir si il y a une méthode de l'API (je n'en ai pas trouvé personnellement) qui me permettrait de récupérer la clef d'un fichier donné (la clef qui est dans l'URL lorsqu'on télécharge un fichier).
Si il n'existe pas de méthode, connais-tu un moyen de la décrypter ?

Merci d'avance. 🙂

Julien Marchand Post author
February 6, 2013 at 12:29 am

Hello !

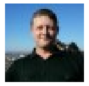
Il n'en existe pas 🙂 L'intérêt de Mega est justement que tous tes fichiers sont chiffrés, donc à moins que tu ne décides toi même de les publier et de partager leur clé, personne ne pourra le télécharger (ou plus exactement, le déchiffrer).

José Ibañez

February 2, 2013 at 10:03 pm

Great job! Thanks for share

Pingback: MegaFS, a FUSE filesystem wrapper for Mega. Part 2: Reading and writing files. | Julien Marchand • Dev blog



George
February 5, 2013 at 5:21 pm

Hi,

Has anyone manged a simple upload example using PHP rather than Python?

Julien Marchand Post author
February 6, 2013 at 12:27 am

Since I'm new to Python, I've actually written all the examples of the blog in PHP first to get more familiar with the API. I'll clean them up and post them by the end of tomorrow

---

DinoEO
February 7, 2013 at 6:39 pm

Ok cool merci pour ton travail

Chris
February 8, 2013 at 12:40 pm

Je suis très interressé aussi par ton dev PHP si tu veux bien le mettre à dispo (et du coup met un système de donation sur ton blog…)

DinoEO
February 8, 2013 at 10:32 pm

Sa serait cool si a chaque poste déjà fait

d'ajouter les source PHP

comme sa on aurais la version Python et PHP

Pingback: Using the Mega API: how to upload a file anonymously (without logging in). | Julien Marchand • Dev blog

Pierrick HALGAND
February 15, 2013 at 9:05 am

Bonjour,

Je découvre cet article avec beaucoup d'attention m'intéressant actuellement à l'API Mega.
Tout d'abord merci beaucoup pour toutes ces informations et tout ces exemples de code.

Pour ma part je développe plutôt en PHP n'ayant malheureusement que très peu utilisé Perl.
J'essaie donc actuellement de porter les exemples Perl en PHP, mais je suis actuellement confronté à un premier problème, je ne sais traduire en PHP le statement : for _ in xrange(0×10000) et en particulier à quoi correspond ce caractère "_" ?

Par avance merci pour votre réponse.

Pingback: Using the Mega API, with PHP examples! | Julien Marchand • Dev blog

uten
February 20, 2013 at 11:32 pm

Nice work!
I'm really looking forward to read your articles about fuse and mega. I might even try to find some time to learn Pyton so I can follow the code and test it properly.

To anyone trying out the code. Copying the code gave some easy to find html formating errors like <, but also a harder to find $amp;sid inside a string witch is valid code but gives an (unhandled ..;o) error from api_req(…). I got -15 <> returned rather than the expected data.

Thanks for your time and effort on this Julien.
Best regards
Uten

Nima
February 21, 2013 at 4:45 pm

hello
thanks for this article!! it helped me a lot!
I trying to write an uploader in php. my only problem is encryption.
I'm using this php class:
http://www.phpclasses.org/package/4238-PHP-Encrypt-and-decrypt-data-with-AES-in-pure-PHP.html
but it's not working!
what is cbc_mode? I know nothing about AES. can you help me plz?!!?

ibk
March 5, 2013 at 12:20 am

Hi,
I can't find crypto.js in http://g.static.mega.co.nz/crypto.js

geomorilllo
March 11, 2013 at 12:23 am

I think the new address is https://eu.static.mega.co.nz/crypto_15.js

nlaa
March 19, 2013 at 7:02 pm

I tried some fast servers to upload from to mega, it takes very long… did you know about it?

Juande
April 2, 2013 at 10:33 pm

Hi, you're a genius, you make everything look really easy, I tried to pass "crypto.js" and "rsa.js" a python and the result was not good.
I was surprised that is also relatively manejabe with PHP, thanks for posting these articles.

---

christelle ledroit
April 4, 2013 at 7:59 pm

how to deal with partial download ? i can't figure out
