Julien Marchand • Dev blog

Yet another dev blog

Using the Mega API, with PHP examples!

Introduction

Update: Fixed the *str_to_a32* function to handle strings with a length not multiple of 4. Thanks to Pablo M. for reporting the bug, both my email address and password were actually a multiple of 4 so I didn't notice

Here we are! I finally found the time to clean up my PHP experiments and write a PHP version of my first article explaining how to use the Mega API. I actually wrote the PHP version before the Python version, because I was learning Python and wanted to play around with the API with a language I know well before switching to Python. I will follow exactly the same structure as the Python version; just replacing the Python examples by PHP code.

For those who didn't read my first article, let's start with some reminders about the Mega API. 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.

```php
$sid = '';
$seqno = rand(0, 0xFFFFFFFF);

function api_req($req) {
  global $seqno, $sid;
  $resp = post('https://g.api.mega.co.nz/cs?id=' . ($seqno++) . ($sid ? '&sid=' . $sid : ''), json_encode(array($req)));
  $resp = json_decode($resp);
  return $resp[0];
}

function post($url, $data) {
  $ch = curl_init($url);
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
  curl_setopt($ch, CURLOPT_POST, true);
  curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
  $resp = curl_exec($ch);
  curl_close($ch);
  return $resp;
}
```

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. 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));
$uh = stringhash(strtolower($email), $password_aes);

function stringhash($s, $aeskey) {
  $s32 = str_to_a32($s);
  $h32 = array(0, 0, 0, 0);

  for ($i = 0; $i < count($s32); $i++) {
    $h32[$i % 4] ^= $s32[$i];
  }

  for ($i = 0; $i < 0x4000; $i++) {
    $h32 = aes_cbc_encrypt_a32($h32, $aeskey);
  }

  return a32_to_base64(array($h32[0], $h32[2]));
}

function prepare_key($a) {
  $pkey = array(0x93C467E3, 0x7DB0C7A4, 0xD1BE3F81, 0x0152CB56);

  for ($r = 0; $r < 0x10000; $r++) {
    for ($j = 0; $j < count($a); $j += 4) {
      $key = array(0, 0, 0, 0);

      for ($i = 0; $i < 4; $i++) {
        if ($i + $j < count($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(array('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.

```
function decrypt_key($a, $key) {
  $x = array();

  for ($i = 0; $i < count($a); $i += 4) {
    $x = array_merge($x, aes_cbc_decrypt_a32(array_slice($a, $i, 4), $key));
  }

  return $x;
}
```

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 BCMath arbitrary precision number:

```
function mpi2bc($s) {
  $s = bin2hex(substr($s, 2));
  $len = strlen($s);
  $n = 0;
  for ($i = 0; $i < $len; $i++) {
    $n = bcadd($n, bcmul(hexdec($s[$i]), bcpow(16, $len - $i - 1)));
  }
  return $n;
}
```

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

```
$privk = a32_to_str($rsa_priv_key);
$rsa_priv_key = array(0, 0, 0, 0);

for ($i = 0; $i < 4; $i++) {
  $l = ((ord($privk[0]) * 256 + ord($privk[1]) + 7) / 8) + 2;
  $rsa_priv_key[$i] = mpi2bc(substr($privk, 0, $l));
  $privk = substr($privk, $l);
}
```

Finally, we can decrypt the session id:

```
$enc_sid = mpi2bc(base64urldecode($res->csid));
$sid = rsa_decrypt($enc_sid, $rsa_priv_key[0], $rsa_priv_key[1], $rsa_priv_key[2]);
$sid = base64urlencode(substr(strrev($sid), 0, 43));
```

There is unfortunately no PHP library that allows to easily decrypt RSA from a private exponent and modulus. So I extracted some code from the now deprecated Crypt_RSA PEAR library to decrypt our session id. We'll see in another article how we can simplify all this code, but I was very late posting this article so I just left it as is:

```
function bin2int($str) {
  $result = 0;
  $n = strlen($str);
  do {
    $result = bcadd(bcmul($result, 256), ord($str[--$n]));
  } while ($n > 0);
  return $result;
}

function int2bin($num) {
  $result = '';
  do {
    $result .= chr(bcmod($num, 256));
    $num = bcdiv($num, 256);
  } while (bccomp($num, 0));
  return $result;
}

function bitOr($num1, $num2, $start_pos) {
  $start_byte = intval($start_pos / 8);
  $start_bit = $start_pos % 8;
  $tmp1 = int2bin($num1);
```

```php
  $num2 = bcmul($num2, 1 << $start_bit);
  $tmp2 = int2bin($num2);
  if ($start_byte < strlen($tmp1)) {
    $tmp2 |= substr($tmp1, $start_byte);
    $tmp1 = substr($tmp1, 0, $start_byte) . $tmp2;
  } else {
    $tmp1 = str_pad($tmp1, $start_byte, '\0') . $tmp2;
  }
  return bin2int($tmp1);
}

function bitLen($num) {
  $tmp = int2bin($num);
  $bit_len = strlen($tmp) * 8;
  $tmp = ord($tmp[strlen($tmp) - 1]);
  if (!$tmp) {
    $bit_len -= 8;
  } else {
    while (!($tmp & 0x80)) {
      $bit_len--;
      $tmp <<= 1;
    }
  }
  return $bit_len;
}

function rsa_decrypt($enc_data, $p, $q, $d) {
  $enc_data = int2bin($enc_data);
  $exp = $d;
  $modulus = bcmul($p, $q);
  $data_len = strlen($enc_data);
  $chunk_len = bitLen($modulus) - 1;
  $block_len = (int) ceil($chunk_len / 8);
  $curr_pos = 0;
  $bit_pos = 0;
  $plain_data = 0;
  while ($curr_pos < $data_len) {
    $tmp = bin2int(substr($enc_data, $curr_pos, $block_len));
    $tmp = bcpowmod($tmp, $exp, $modulus);
    $plain_data = bitOr($plain_data, $tmp, $bit_pos);
    $bit_pos += $chunk_len;
    $curr_pos += $block_len;
  }
  return int2bin($plain_data);
}
```

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(array('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'}*):

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

Then, our main loop:

```
foreach ($files->f as $file) {
  if ($file->t == 0 || $file->t == 1) {
    $key = substr($file->k, strpos($file->k, ':') + 1);
    $key = decrypt_key(base64_to_a32($key), $master_key);
    if ($file->t == 0) {
      $k = array($key[0] ^ $key[4], $key[1] ^ $key[5], $key[2] ^ $key[6], $key[3] ^ $key[7]);
      $iv = array_merge(array_slice($key, 4, 2), array(0, 0));
      $meta_mac = array_slice($key, 6, 2);
    } else {
      $k = $key;
    }
    $attributes = base64urldecode($file->a);
    $attributes = dec_attr($attributes, $k);
  } else if ($file->t == 2) {
    $root_id = $file->k;
  } else if ($file->t == 3) {
    $inbox_id = $file->k;
  } else if ($file->t == 4) {
    $trashbin_id = $file->k;
  }
}
```

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(array('a' => 'g', 'g' => 1, 'n' => $file->h));
$dl_url = $dl_url->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. We have done the latter in the Python version, so let’s try the former in PHP (we will see how to download and decrypt it on the fly in a next article, but for now it’s just simpler to download it first and then decrypt it, because we can do that in one line with mcrypt).

```
$data_enc = file_get_contents($dl_url);
$data = aes_ctr_decrypt($data_enc, a32_to_str($k), a32_to_str($iv));
file_put_contents($attributes->n, $data);
```

And as promised, *aes_ctr_decrypt* is implemented in one line with mcrypt (this is because we are decrypting the whole file at once):

```
function aes_ctr_decrypt($data, $key, $iv) {
  return mcrypt_decrypt(MCRYPT_RIJNDAEL_128, $key, $data, 'ctr', $iv);
}
```

We can now check the file integrity. Mega is using CBC-MAC for that purpose:

> *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)

Let's write the code that implements that!

```
$file_mac = cbc_mac($data, $k, $iv);
if (array($file_mac[0] ^ $file_mac[1], $file_mac[2] ^ $file_mac[3]) != $meta_mac) {
  echo "MAC mismatch";
}

function cbc_mac($data, $k, $n) {
  $padding_size = (strlen($data) % 16) == 0 ? 0 : 16 - strlen($data) % 16;
  $data .= str_repeat("\0", $padding_size);

  $chunks = get_chunks(strlen($data));
  $file_mac = array(0, 0, 0, 0);

  foreach ($chunks as $pos => $size) {
    $chunk_mac = array($n[0], $n[1], $n[0], $n[1]);
    for ($i = $pos; $i < $pos + $size; $i += 16) {
      $block = str_to_a32(substr($data, $i, 16));
      $chunk_mac = array($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);
    }
    $file_mac = array($file_mac[0] ^ $chunk_mac[0], $file_mac[1] ^ $chunk_mac[1], $file_mac[2] ^ $chunk_mac[2], $file_mac[3] ^ $chunk_ma
    $file_mac = aes_cbc_encrypt_a32($file_mac, $k);
  }

  return $file_mac;
}
```

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.

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:

```
$data = file_get_contents($filename);
$size = strlen($data);
$ul_url = api_req(array('a' => 'u', 's' => $size));
$ul_url = $ul_url->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!

```
$ul_key = array(0, 0, 0, 0, 0, 0);
for ($i = 0; $i < 6; $i++) {
  $ul_key[$i] = rand(0, 0xFFFFFFFF);
}
$data_crypted = aes_ctr_encrypt($data, a32_to_str(array_slice($ul_key, 0, 4)), a32_to_str(array($ul_key[4], $ul_key[5], 0, 0)));
$completion_handle = post($ul_url, $data_crypted);
```

As for *aes_ctr_decrypt*, *aes_ctr_encrypt* can be implemented in only one line with mcrypt, since we are encrypting the whole file at once:

```
function aes_ctr_encrypt($data, $key, $iv) {
  return mcrypt_encrypt(MCRYPT_RIJNDAEL_128, $key, $data, 'ctr', $iv);
}
```

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()*):

```
function enc_attr($attr, $key) {
  $attr = 'MEGA' . json_encode($attr);
  return aes_cbc_encrypt($attr, a32_to_str($key));
}

function encrypt_key($a, $key) {
  $x = array();

  for ($i = 0; $i < count($a); $i += 4) {
    $x = array_merge($x, aes_cbc_encrypt_a32(array_slice($a, $i, 4), $key));
  }

  return $x;
}
```

We can now create the new node:

```
$data_mac = cbc_mac($data, array_slice($ul_key, 0, 4), array_slice($ul_key, 4, 2));
$meta_mac = array($data_mac[0] ^ $data_mac[1], $data_mac[2] ^ $data_mac[3]);
$attributes = array('n' => basename($filename));
$enc_attributes = enc_attr($attributes, array_slice($ul_key, 0, 4));
$key = array($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
api_req(array('a' => 'p', 't' => $root_id, 'n' => array(array('h' => $completion_handle, 't' => 0, 'a' => base64urlencode($enc_attribute
```

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!

```
<?php
$sid = '';
$seqno = rand(0, 0xFFFFFFFF);
```

```
$master_key = '';
$rsa_priv_key = '';

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

function base64urlencode($data) {
  return str_replace(array('+', '/', '='), array('-', '_', ''), base64_encode($data));
}

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

function a32_to_base64($a) {
  return base64urlencode(a32_to_str($a));
}

function str_to_a32($b) {
  // Add padding, we need a string with a length multiple of 4
  $b = str_pad($b, 4 * ceil(strlen($b) / 4), "\0");
  return array_values(unpack('N*', $b));
}

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

function aes_cbc_encrypt($data, $key) {
  return mcrypt_encrypt(MCRYPT_RIJNDAEL_128, $key, $data, MCRYPT_MODE_CBC, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0");
}

function aes_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");
}

function aes_cbc_encrypt_a32($data, $key) {
  return str_to_a32(aes_cbc_encrypt(a32_to_str($data), a32_to_str($key)));
}
```

```php
function aes_cbc_decrypt_a32($data, $key) {
  return str_to_a32(aes_cbc_decrypt(a32_to_str($data), a32_to_str($key)));
}

function aes_ctr_encrypt($data, $key, $iv) {
  return mcrypt_encrypt(MCRYPT_RIJNDAEL_128, $key, $data, 'ctr', $iv);
}

function aes_ctr_decrypt($data, $key, $iv) {
  return mcrypt_decrypt(MCRYPT_RIJNDAEL_128, $key, $data, 'ctr', $iv);
}

/*
 * BEGIN RSA-related stuff -- taken from PEAR Crypt_RSA package
 * http://pear.php.net/package/Crypt_RSA
 */
function bin2int($str) {
  $result = 0;
  $n = strlen($str);
  do {
    $result = bcadd(bcmul($result, 256), ord($str[--$n]));
  } while ($n > 0);
  return $result;
}

function int2bin($num) {
  $result = '';
  do {
    $result .= chr(bcmod($num, 256));
    $num = bcdiv($num, 256);
  } while (bccomp($num, 0));
  return $result;
}

function bitOr($num1, $num2, $start_pos) {
  $start_byte = intval($start_pos / 8);
  $start_bit = $start_pos % 8;
  $tmp1 = int2bin($num1);

  $num2 = bcmul($num2, 1 << $start_bit);
  $tmp2 = int2bin($num2);
  if ($start_byte < strlen($tmp1)) {
    $tmp2 |= substr($tmp1, $start_byte);
    $tmp1 = substr($tmp1, 0, $start_byte) . $tmp2;
  } else {
    $tmp1 = str_pad($tmp1, $start_byte, '\0') . $tmp2;
  }
  return bin2int($tmp1);
}

function bitLen($num) {
  $tmp = int2bin($num);
  $bit_len = strlen($tmp) * 8;
  $tmp = ord($tmp[strlen($tmp) - 1]);
  if (!$tmp) {
    $bit_len -= 8;
  } else {
    while (!($tmp & 0x80)) {
      $bit_len--;
      $tmp <<= 1;
    }
  }
  return $bit_len;
}

function rsa_decrypt($enc_data, $p, $q, $d) {
  $enc_data = int2bin($enc_data);
  $exp = $d;
  $modulus = bcmul($p, $q);
  $data_len = strlen($enc_data);
  $chunk_len = bitLen($modulus) - 1;
  $block_len = (int) ceil($chunk_len / 8);
  $curr_pos = 0;
  $bit_pos = 0;
  $plain_data = 0;
  while ($curr_pos < $data_len) {
    $tmp = bin2int(substr($enc_data, $curr_pos, $block_len));
    $tmp = bcpowmod($tmp, $exp, $modulus);
    $plain_data = bitOr($plain_data, $tmp, $bit_pos);
    $bit_pos += $chunk_len;
    $curr_pos += $block_len;
  }
  return int2bin($plain_data);
}
/*
 * END RSA-related stuff
 */
```

```php
function stringhash($s, $aeskey) {
  $s32 = str_to_a32($s);
  $h32 = array(0, 0, 0, 0);

  for ($i = 0; $i < count($s32); $i++) {
    $h32[$i % 4] ^= $s32[$i];
  }

  for ($i = 0; $i < 0x4000; $i++) {
    $h32 = aes_cbc_encrypt_a32($h32, $aeskey);
  }

  return a32_to_base64(array($h32[0], $h32[2]));
}

function prepare_key($a) {
  $pkey = array(0x93C467E3, 0x7DB0C7A4, 0xD1BE3F81, 0x0152CB56);

  for ($r = 0; $r < 0x10000; $r++) {
    for ($j = 0; $j < count($a); $j += 4) {
      $key = array(0, 0, 0, 0);

      for ($i = 0; $i < 4; $i++) {
        if ($i + $j < count($a)) {
          $key[$i] = $a[$i + $j];
        }
      }

      $pkey = aes_cbc_encrypt_a32($pkey, $key);
    }
  }

  return $pkey;
}

function encrypt_key($a, $key) {
  $x = array();

  for ($i = 0; $i < count($a); $i += 4) {
    $x = array_merge($x, aes_cbc_encrypt_a32(array_slice($a, $i, 4), $key));
  }

  return $x;
}
```

```php
function decrypt_key($a, $key) {
  $x = array();

  for ($i = 0; $i < count($a); $i += 4) {
    $x = array_merge($x, aes_cbc_decrypt_a32(array_slice($a, $i, 4), $key));
  }

  return $x;
}

function mpi2bc($s) {
  $s = bin2hex(substr($s, 2));
  $len = strlen($s);
  $n = 0;
  for ($i = 0; $i < $len; $i++) {
    $n = bcadd($n, bcmul(hexdec($s[$i]), bcpow(16, $len - $i - 1)));
  }
  return $n;
}

function api_req($req) {
  global $seqno, $sid;
  $resp = post('https://g.api.mega.co.nz/cs?id=' . ($seqno++) . ($sid ? '&sid=' . $sid : ''), json_encode(array($req)));
  $resp = json_decode($resp);
  return $resp[0];
}

function post($url, $data) {
  $ch = curl_init($url);
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
  curl_setopt($ch, CURLOPT_POST, true);
  curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
  $resp = curl_exec($ch);
  curl_close($ch);
  return $resp;
}

function login($email, $password) {
  global $sid, $master_key, $rsa_priv_key;
  $password_aes = prepare_key(str_to_a32($password));
  $uh = stringhash(strtolower($email), $password_aes);
  $res = api_req(array('a' => 'us', 'user' => $email, 'uh' => $uh));
  $enc_master_key = base64_to_a32($res->k);
  $master_key = decrypt_key($enc_master_key, $password_aes);
  if (!empty($res->csid)) {
    $enc_rsa_priv_key = base64_to_a32($res->privk);
    $rsa_priv_key = decrypt_key($enc_rsa_priv_key, $master_key);

    $privk = a32_to_str($rsa_priv_key);
    $rsa_priv_key = array(0, 0, 0, 0);

    for ($i = 0; $i < 4; $i++) {
      $l = ((ord($privk[0]) * 256 + ord($privk[1]) + 7) / 8) + 2;
      $rsa_priv_key[$i] = mpi2bc(substr($privk, 0, $l));
      $privk = substr($privk, $l);
    }

    $enc_sid = mpi2bc(base64urldecode($res->csid));
    $sid = rsa_decrypt($enc_sid, $rsa_priv_key[0], $rsa_priv_key[1], $rsa_priv_key[2]);
    $sid = base64urlencode(substr(strrev($sid), 0, 43));
  }
}

function enc_attr($attr, $key) {
  $attr = 'MEGA' . json_encode($attr);
  return aes_cbc_encrypt($attr, a32_to_str($key));
}

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

function get_chunks($size) {
  $chunks = array();
  $p = $pp = 0;

  for ($i = 1; $i <= 8 && $p < $size - $i * 0x20000; $i++) {
    $chunks[$p] = $i * 0x20000;
    $pp = $p;
    $p += $chunks[$p];
  }

  while ($p < $size) {
    $chunks[$p] = 0x100000;
    $pp = $p;
    $p += $chunks[$p];
  }

  $chunks[$pp] = ($size - $pp);
  if (!$chunks[$pp]) {
    unset($chunks[$pp]);
  }

  return $chunks;
}

function cbc_mac($data, $k, $n) {
  $padding_size = (strlen($data) % 16) == 0 ? 0 : 16 - strlen($data) % 16;
  $data .= str_repeat("\0", $padding_size);

  $chunks = get_chunks(strlen($data));
  $file_mac = array(0, 0, 0, 0);

  foreach ($chunks as $pos => $size) {
    $chunk_mac = array($n[0], $n[1], $n[0], $n[1]);
    for ($i = $pos; $i < $pos + $size; $i += 16) {
      $block = str_to_a32(substr($data, $i, 16));
      $chunk_mac = array($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);
    }
    $file_mac = array($file_mac[0] ^ $chunk_mac[0], $file_mac[1] ^ $chunk_mac[1], $file_mac[2] ^ $chunk_mac[2], $file_mac[3] ^ $chunk_ma
    $file_mac = aes_cbc_encrypt_a32($file_mac, $k);
  }

  return $file_mac;
}

function uploadfile($filename) {
  global $master_key, $root_id;

  $data = file_get_contents($filename);
  $size = strlen($data);
  $ul_url = api_req(array('a' => 'u', 's' => $size));
  $ul_url = $ul_url->p;

  $ul_key = array(0, 1, 2, 3, 4, 5);
```

```php
  for ($i = 0; $i < 6; $i++) {
    $ul_key[$i] = rand(0, 0xFFFFFFFF);
  }

  $data_crypted = aes_ctr_encrypt($data, a32_to_str(array_slice($ul_key, 0, 4)), a32_to_str(array($ul_key[4], $ul_key[5], 0, 0)));
  $completion_handle = post($ul_url, $data_crypted);

  $data_mac = cbc_mac($data, array_slice($ul_key, 0, 4), array_slice($ul_key, 4, 2));
  $meta_mac = array($data_mac[0] ^ $data_mac[1], $data_mac[2] ^ $data_mac[3]);
  $attributes = array('n' => basename($filename));
  $enc_attributes = enc_attr($attributes, array_slice($ul_key, 0, 4));
  $key = array($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_k
  return api_req(array('a' => 'p', 't' => $root_id, 'n' => array(array('h' => $completion_handle, 't' => 0, 'a' => base64urlencode($enc_
}

function downloadfile($file, $attributes, $k, $iv, $meta_mac) {
  $dl_url = api_req(array('a' => 'g', 'g' => 1, 'n' => $file->h));

  $data_enc = file_get_contents($dl_url->g);
  $data = aes_ctr_decrypt($data_enc, a32_to_str($k), a32_to_str($iv));
  file_put_contents($attributes->n, $data);

  $file_mac = cbc_mac($data, $k, $iv);
  if (array($file_mac[0] ^ $file_mac[1], $file_mac[2] ^ $file_mac[3]) != $meta_mac) {
    echo "MAC mismatch";
  }
}

function getfiles() {
  global $master_key, $root_id, $inbox_id, $trashbin_id;

  $files = api_req(array('a' => 'f', 'c' => 1));
  foreach ($files->f as $file) {
    if ($file->t == 0 || $file->t == 1) {
      $key = substr($file->k, strpos($file->k, ':') + 1);
      $key = decrypt_key(base64_to_a32($key), $master_key);
      if ($file->t == 0) {
        $k = array($key[0] ^ $key[4], $key[1] ^ $key[5], $key[2] ^ $key[6], $key[3] ^ $key[7]);
        $iv = array_merge(array_slice($key, 4, 2), array(0, 0));
        $meta_mac = array_slice($key, 6, 2);
      } else {
        $k = $key;
      }
      $attributes = base64urldecode($file->a);
      $attributes = dec_attr($attributes, $k);
      if ($file->h == 'gldU3Tab') {
        downloadfile($file, $attributes, $k, $iv, $meta_mac);
      }
    } else if ($file->t == 2) {
      $root_id = $file->k;
    } else if ($file->t == 3) {
      $inbox_id = $file->k;
    } else if ($file->t == 4) {
      $trashbin_id = $file->k;
    }
  }
}
```

Google+

Share this: Like 14 Tweet Digg reddit More

This entry was posted in Uncategorized on February 17, 2013 [/web/20140402072205/http://julien-marchand.fr/blog/using-the-mega-api-with-php-examples/] .

Using the Mega API: how to upload a file anonymously (without logging in).

I received some emails asking me how to upload a file anonymously using the API, since it is possible to upload a file from the Mega.co.nz website without logging in. I actually gave a solution in the comments of this article, but it's not very easy to find, so I'm posting it here 😃

"Anonymous" uploads use ephemeral accounts, as described in the developer's guide:

> *MEGA supports ephemeral accounts to enable pre-registration file manager operations, allowing applications to present MEGA's functionality with a lowered barrier to entry. They naturally do not have an e-mail address associated with them, the password (master key) is generated randomly, and they are subject to frequent purging. In a browser context, the credentials for ephemeral accounts live in the DOM storage.*

Thus, we actually have to log in, but with a freshly generated ephemeral account. Let's see how it works:

```
def login_anon():
  global sid, master_key
  master_key = [random.randint(0, 0xFFFFFFFF)] * 4
  password_key = [random.randint(0, 0xFFFFFFFF)] * 4
  session_self_challenge = [random.randint(0, 0xFFFFFFFF)] * 4

  user_handle = api_req({
      'a': 'up',
      'k': a32_to_base64(encrypt_key(master_key, password_key)),
      'ts': base64urlencode(a32_to_str(session_self_challenge) + a32_to_str(encrypt_key(session_self_challenge, master_key)))
  })

  print "ephemeral user handle: %s" % user_handle
  res = api_req({'a': 'us', 'user': user_handle})

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

We randomly generate a master key, a "password key" (equivalent to the hash of a regular user's password) to encrypt to master key, and a session self challenge (that will be used to check the generated password and get the session ID, since our ephemeral account does not have a RSA key pair).

Then, the *getfiles()* and *uploadfile()* functions are the same as in my first article. So, let's upload a file anonymously and get its public URL to share it on the web:

```
login_anon()
getfiles()
uploaded_file = uploadfile('/home/julienm/mega/test_file.png')
print getpublicurl(uploaded_file['f'][0])
```

We have to call the *getfiles()* function to get the ID of the root node, to which we are uploading our file. The *uploadfile()* method is simply modified to change the final "print" into a "return" and return informations about the uploaded file.

The *getpublicurl()* method gets the public handle of the file (that is not the same as the "private" handle that you see in the '*h*' attribute when listing your files, you can actually enable or disable the public handle for a file, whether you want it to be public or not), decrypts its key, and concatenates the two informations to obtain the public URL:

```
def getpublicurl(file):
  public_handle = api_req({'a': 'l', 'n': file['h']})
  key = file['k'][file['k'].index(':') + 1:]
  decrypted_key = a32_to_base64(decrypt_key(base64_to_a32(key), master_key))
  return "http://mega.co.nz/#!%s!%s" % (public_handle, decrypted_key)
```

That's it! It works 😃

```
julienm@rchand:~/mega$ python anon_upload.py
ephemeral user handle: 5hq-EIBu_yc

http://mega.co.nz/#!V51SVYzY!pMS4P8hyBqFBC3QdhOYNG4xEbJ8Kj8dYQFuxdDt6dMU

julienm@rchand:~/mega$
```

I'm going to post part 3 of the MegaFS series very soon, as well as a PHP and a Java version of all my examples. Stay tuned 😉

Google+

Share this: Like 0 Tweet Digg reddit More

This entry was posted in Uncategorized on February 6, 2013 [/web/20140402072205/http://julien-marchand.fr/blog/using-the-mega-api-how-to-upload-a-file-anonymously-without-logging-in/] .

MegaFS, a FUSE filesystem wrapper for Mega. Part 2: Reading and writing files.

In the previous article, we implemented the beginning of a filesystem where we can list our Mega files and navigate through them. It is now time to read/download them, and create/upload new ones! In order to to that, we need to implement 5 FUSE callbacks:

- *mknod(path, mode, dev)*: called to create a new, empty file ;
- *open(path, flags)*: called to open a file ;
- *read(path, size, offset, fh)*: called to read *size* bytes from a file at a given *offset ;*
- *write(path, buf, offset, fh)*: called to write the contents of *buf* to a file at a given *offset ;*
- *release(path, flags, fh)*: called to release a file (the opposite of *open*, see the FUSE FAQ to know the relation between *release* and the *close* system call).

Reading files

The *read* implementation can be tricky because of the *size* and *offset* parameters. They can take any value, but since the contents of the file contents are encrypted, we can only deal with data starting and ending on an AES block boundary (16 bytes). We could handle that by downloading the blocks containing the requested range (the Mega API supports partial downloads via the HTTP Range header and/or a suffix appended to the download URL), decrypting them and returning only the part really requested.

But for now, we will follow a simpler approach: just download the entire file and store it in a temporary file when *open* is called. Then, we can just seek/read this temporary file whenever *read* is called.

```
def open(self, path, flags):
  if path not in self.files:
    return -errno.ENOENT

  if (flags &amp; 3) == os.O_RDONLY:
    (tmp_f, tmp_path) = tempfile.mkstemp(prefix='megafs')
    os.close(tmp_f)
    if self.client.downloadfile(self.files[path], tmp_path):
      return open(tmp_path, "rb")
    else:
      return -errno.EACCESS
  else:
    return -errno.EINVAL
```

We currently only support files opened in read-only mode (flag O_RDONLY), so we start by checking that (the access mode is stored in the two least significant bits of *flags*, which explains the “*& 3*” mask). We then create the temporary file, download the entire Mega file to it (the *downloadfile* method is pretty much the same as in my first article, with an additional parameter to specify the target path), and check that the download succeeded by checking the meta-MAC (a thing we wouldn’t be able to do with partial downloads because we don’t know the MACs of each chunk). We finally return a handle to this temporary file, which will be used by *read* and *release*:

```
def read(self, path, size, offset, fh):
  fh.seek(offset)
  return fh.read(size)
```

Our approach makes the *read* implementation very simple: just seek to the desired offset and read the temporary file. We are now almost done, and only need to delete the temporary file when the file is released:

```
def release(self, path, flags, fh):
  fh.close()
  os.unlink(fh.name)
```

That’s all! We can now move to the *write* implementation.

Writing files

The writing part is even more tricky than the reading part: we have not only the encryption problem (we can only handle data aligned on AES block boundaries to be able to encrypt it, so we would need to buffer the data), but we also have to know the final size of the file before writing it! It is indeed required by the API ‘*u*‘ method, used to request an upload URL. Moreover, partial uploads must start and end on a chunk boundary (see the Mega developer’s guide, section 5 – “File encryption and integrity checking”, but this is in fact pretty much the same problem as the encryption one, with largest boundaries).

We will address these problems while keeping our code as simple as possible by using quite the same technique as earlier: we will first write all the data to a temporary file, and then upload it when the file is released. That way, we will know its final size when requesting an upload URL from the API.

Let's start with the changes to the *open* method. The only writing mode allowed is *O_WRONLY* | *O_CREAT* | *O_EXCL* (see the Mega developer's guide, section 1.3 – "Storage model"), that is, write-only with creation of the file: we can't write to an existing file, nor can we open a file for both reading and writing. But in fact, *open* is not in charge of creating the file: a call to *open(path, flags* | *O_CREAT)* will be turned into a call to *mknod* (to create the file) followed by a call to *open(path, flags)*. Thus, we have to implement *mknod*:

```
def mknod(self, path, mode, dev):
  if path in self.files:
    return -errno.EEXIST

  dirname, basename = os.path.split(path)
  self.files[dirname]['children'].append(basename)
  self.files[path] = {'t': 0, 'ts': int(time.time()), 's': 0}
```

When *mknod(path)* is called, we add a new, empty file to our filesystem at *path*, so that it can be found by the *getattr* and *open* methods (they are called just after *mknod* and need to see that the file has been created). It obviously has no Mega ID ('*h*' field), and this will enable us to distinguish these "special" files from the others. We can now make changes to *open* to handle openings in write-only mode:

```
def open(self, path, flags):
  if path not in self.files:
    return -errno.ENOENT

  if (flags &amp; 3) == os.O_RDONLY:
    (tmp_f, tmp_path) = tempfile.mkstemp(prefix='megafs')
    os.close(tmp_f)
    if 'h' not in self.files[path]:
      return open(tmp_path, "rb")
    elif self.client.downloadfile(self.files[path], tmp_path):
      return open(tmp_path, "rb")
    else:
      return -errno.EACCESS
  elif (flags &amp; 3) == os.O_WRONLY:
    if 'h' in self.files[path]:
      return -errno.EEXIST
    (tmp_f, tmp_path) = tempfile.mkstemp(prefix='megafs')
    os.close(tmp_f)
    return open(tmp_path, "wb")
  else:
    return -errno.EINVAL
```

For the *O_RDONLY* case, we just add a condition to check that the file is a “real” file (with a Mega ID, and not an empty file created by *mknod*) before downloading it.

For the *O_WRONLY* case, we first check that the file is an empty file created by *mknod*: if it’s not the case, that means that we are trying to write to an existing file, and that is not allowed by the API. Then we create the temporary file we are going to write all the data into, and we return a handle to this file that will be used by *write* and *release*:

```
def write(self, path, buf, offset, fh):
  fh.seek(offset)
  fh.write(buf)
  return len(buf)
```

As for the *read* method, our approach makes the *write* implementation very simple: just seek to the desired offset and write the contents of the buffer to the temporary file.

Now, let’s add some code to the *release* method to effectively upload the file when we are done writing to it:

```
def release(self, path, flags, fh):
  fh.close()
  if fh.mode == "wb":
    dirname, basename = os.path.split(path)
    uploaded_file = self.client.uploadfile(fh.name, self.files[dirname]['h'], basename)
    if 'f' in uploaded_file:
      uploaded_file = self.client.processfile(uploaded_file['f'][0])
      self.files[path] = uploaded_file
  os.unlink(fh.name)
```

If the temporary file was opened for writing, we upload it and create the new node on Mega. The *uploadfile* method is the same as in my first article, with an additional parameter to specify the source path. It returns the information on the new node, but with its key and attributes still encrypted. Thus, we give it to a *processfile* method that is just the main loop of the *getfiles* method (from the first article) extracted to a method (you can find the complete code on GitHub):

```
def getfiles(self):
  files = self.api_req({'a': 'f', 'c': 1})
  files_dict = {}
  for file in files['f']:
    files_dict[file['h']] = self.processfile(file)
  return files_dict
```

```
def processfile(self, file):
  if file['t'] == 0 or file['t'] == 1:
    key = file['k'][file['k'].index(':') + 1:]
    key = decrypt_key(base64_to_a32(key), self.master_key)
    if file['t'] == 0:
      file['k'] = (key[0] ^ key[4], key[1] ^ key[5], key[2] ^ key[6], key[3] ^ key[7])
      file['iv'] = key[4:6] + (0, 0)
      file['meta_mac'] = key[6:8]
    else:
      file['k'] = key
    attributes = base64urldecode(file['a'])
    attributes = dec_attr(attributes, file['k'])
    file['a'] = attributes
  elif file['t'] == 2:
    self.root_id = file['h']
    file['a'] = {'n': 'Cloud Drive'}
  elif file['t'] == 3:
    self.inbox_id = file['h']
    file['a'] = {'n': 'Inbox'}
  elif file['t'] == 4:
    self.trashbin_id = file['h']
    file['a'] = {'n': 'Rubbish Bin'}
  return file
```

We are now done! Let’s test the new version of our filesystem:

```
julienm@rchand:~$ python MegaFS/megafs.py ~/megafs
Email [go.julienm@gmail.com]:
Password:
julienm@rchand:~$ cd megafs/Cloud\ Drive/
julienm@rchand:~/megafs/Cloud Drive$ cp ~/MegaFS/megafs.py .
julienm@rchand:~/megafs/Cloud Drive$ head megafs.py
from megaclient import MegaClient
import errno
import fuse
import getpass
import os
import stat
import tempfile
import time

fuse.fuse_python_api = (0, 2)
julienm@rchand:~/megafs/Cloud Drive$ tail megafs.py

if __name__ == '__main__':
  email = raw_input("Email [%s]: " % getpass.getuser())
  if not email:
    email = getpass.getuser()
  password = getpass.getpass()
  client = MegaClient(email, password)
  fs = MegaFS(client)
  fs.parse(errex=1)
  fs.main()
julienm@rchand:~/megafs/Cloud Drive$ ls -l megafs.py
-rw-rw-rw- 1 root root 3821 févr.  2 22:42 megafs.py
julienm@rchand:~/megafs/Cloud Drive$ cd
julienm@rchand:~$ sudo umount ~/megafs
julienm@rchand:~$
```

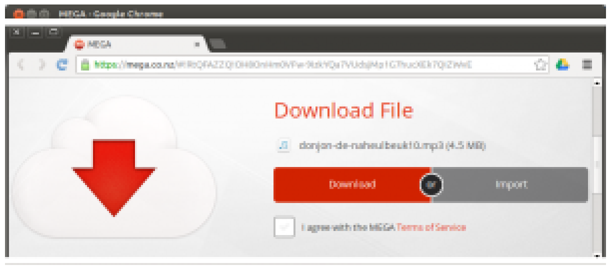
In the next article, we will see how to rename/move files, change their attributes, and delete them. Meanwhile, you can find the complete source code on the MegaFS project page on GitHub: https://github.com/CyberjujuM/MegaFS!

This entry was posted in Uncategorized on February 2, 2013 [/web/20140402072205/http://julien-marchand.fr/blog/megafs-a-fuse-filesystem-wrapper-for-mega-part-2-reading-and-writing-files/] .

Using the Mega API: how to download a public file (or a file you know the key), without logging in.

In my first article, I showed how to log into the Mega API, list all of your own files, download them, and upload new files. But I didn't talk about how to download public files (files that you know the link/the key), without logging in, just as a visitor on http://mega.co.nz can do.

Let's take this file as an example:


MEGA - Google Chrome
MEGA
Download File
donjon-de-naheulbeuk10.mp3 (4.5 MB)
Download
or
Import
I agree with the MEGA Terms of Service

If we look at the URL, we can notice two components, separated by a '*!*':

- *RtQFAZZQ*: the file ID ;
- *OH8OnHm0VFw-9IzkYQa7VUdsjMp1G7hucXEk7QIZWvE*: the file key, already decrypted (the key is stored encrypted with the owner's master key on Mega's servers, but when he decides to share the file, he shares the decrypted key so that other people can decrypt the attributes of the file and its contents).

To download the file, we can follow almost the same steps as in the *getfiles()* and *downloadfile()* functions (see my previous article for more details):

- Decompose the key into its three components: *k*, *iv* and *meta_mac* ;
- Get informations about the file (its attributes, size and download URL): this is done with the API *g* method, that we used to get the download URL of our files in the previous article. But instead of giving the ID of the file as a *n* parameter, we will pass it as a *p* parameter.
- Download the file using the download URL, decrypt it and check its meta-MAC.

So… here we go!

```python
def getfile(file_id, file_key):
  key = base64_to_a32(file_key)
  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]

  file = api_req({'a': 'g', 'g': 1, 'p': file_id})
  dl_url = file['g']
  size = file['s']
  attributes = base64urldecode(file['at'])
  attributes = dec_attr(attributes, k)

  print "Downloading %s (size: %d), url = %s" % (attributes['n'], size, dl_url)

  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"
  else:
    print "MAC OK"
getfile('RtQFAZZQ', 'OH8OnHm0VFw-9IzkYQa7VUdsjMp1G7hucXEk7QIZWvE')
```

All the utility functions are the same as in the previous article.

We can now test our program and see the result

```
julienm@rchand:~$ python mega/megalol_propre.py
Downloading donjon-de-naheulbeuk10.mp3 (size: 4676674), url = http://gfs262n152.userstorage.mega.co.nz/dl/yKpztNG6YnZ1bQLVBMVnNxMWOljOE
MAC OK
julienm@rchand:~$ ls -l donjon-de-naheulbeuk10.mp3
-rw-rw-r-- 1 julienm julienm 4676674 janv. 29 23:04 donjon-de-naheulbeuk10.mp3
julienm@rchand:~$ file donjon-de-naheulbeuk10.mp3
donjon-de-naheulbeuk10.mp3: MPEG ADTS, layer III, v1, 128 kbps, 44.1 kHz, Stereo
julienm@rchand:~$
```

Google+

Share this: Like 5 Tweet Digg reddit More

This entry was posted in Uncategorized on January 29, 2013 [/web/20140402072205/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/] .

MegaFS, a FUSE filesystem wrapper for Mega. Part 1: Listing files.

In this series of articles, we will implement a FUSE filesystem wrapper for Mega, that will allow us to mount our Mega space on Linux. FUSE (Filesystem in userspace) is a kernel module allowing to create our own filesystems directly in userspace, without editing kernel code. Implementing a filesystem wrapper for

FUSE in Python is really simple thanks to *fuse-python* (*aptitude install python-fuse*). We only have to subclass *fuse.Fuse* to implement our filesystem operations and add a few lines of boilerplate code:

```
import fuse

fuse.fuse_python_api = (0, 2)

class DummyFS(fuse.Fuse):
  def __init__(self, *args, **kw):
   fuse.Fuse.__init__(self, *args, **kw)

  def getattr(self, path):
    return 0

  def readdir(self, path):
    return 0

  def open(self, path, flags):
    return 0

  def read(self, path, length, offset):
    return 0

  # ...
fs = DummyFS()
fs.parse(errex=1)
fs.main()
```

For more information about FUSE, feel free to read the Wikipedia article, the FUSE Python tutorial, and the FUSE simple filesystem howto.

In this first article, we will focus on showing the list of our files, and having a filesystem we can *cd* and *ls* around in. To achieve this, we'll have to implement two functions:

- *getattr()*, to get the attributes of a file (type, size, creation/access/modification time, owner, permissions…)
- *readdir()*, to list the contents of a directory.

But first, we need to think about the data structure that will hold our filesystem structure. Let's keep things simple and just use a dictionnary mapping paths to file objects. The file objects are the ones returned by the API *f* method (see my previous article for a reminder), with the attributes decrypted and the key decomposed into its three parts (*k*, *iv* and *meta_mac*). The directories will contain a additional entry *children* listing their contents.

```
{'/Cloud Drive': {
  'a': {'n': 'Cloud Drive'},
  'children': ['file1.mp3', 'file2.png', 'lol'],
  'h': 'hash1',
  'k': '',
  'p': '',
  't': 2,
  'ts': 1234567890,
  'u': 'user1'
},
'/Cloud Drive/file1.mp3': {
  'a': {'n': 'file1.mp3'},
  'h': 'hash2',
  'k': (12345, 67890, 54321, 09876),
  'iv': (12345, 67890, 0, 0),
  'meta_mac': (12345, 67890),
  'p': 'hash1',
  's': 12345,
  't': 0,
  'ts': 1234567890,
  'u': 'user1'
},
'/Cloud Drive/file2.png': {
  'a': {'n': 'file2.png'},
  'h': 'hash3',
  'k': (12345, 67890, 54321, 09876),
  'iv': (12345, 67890, 0, 0),
  'meta_mac': (12345, 67890),
  'p': 'hash1',
  's': 12345,
  't': 0,
  'ts': 1234567890,
  'u': 'user1'
},
'/Cloud Drive/lol': {
  'a': {'n': 'lol'},
  'children': ['lol1.png'],
  'h': 'hash4',
  'k': (12345, 67890, 54321, 09876),
  'p': 'hash1',
  't': 1,
  'ts': 1234567890,
  'u': 'user1'
},
'/Cloud Drive/lol/lol1.png': {
  'a': {'n': 'lol1.png'},
  'h': 'hash5',
  'iv': (12345, 67890, 0, 0),
  'k': (12345, 67890, 54321, 09876),
  'meta_mac': (12345, 67890),
  'p': 'hash4',
  's': 12345,
  't': 0,
  'ts': 1234567890,
  'u': 'user1'
}}
```

To build this dict, we can simply iterate over the list of files returned by the API. But in order to add a file to its parent's list of children, we need to create the parent entry in the dict before the child entries. We could have sorted the dict to ensure that (sort of topological sort), but instead we will simply test if the parent entry exists when adding a child, and create it if necessary.

```
for file_h, file in self.client.getfiles().items():
  path = self.getpath(files, file_h)
  dirname, basename = os.path.split(path)
  if not dirname in self.files:
    self.files[dirname] = {'children': []}
  self.files[dirname]['children'].append(basename)
  if path in self.files:
    self.files[path].update(file)
  else:
    self.files[path] = file
    if file['t'] &gt; 0:
      self.files[path]['children'] = []
```

We now need to get the path associated with a file, that its, its name concatened with the name of all its parents (with a “/” delimiter between them). The only trick is that Mega allows files in the same directory to have the same name (and thus, the same path). But we need the paths to be unique, so we will remember all the computed paths, and add a suffix to files that collide with other files. Thus, we will end up with *file.ext*, *file (1).ext*, *file (2).ext*, etc.

```
def getpath(self, files, hash):
  if not hash:
    return ""
  elif not hash in self.hash2path:
    path = self.getpath(files, files[hash]['p']) + "/" + files[hash]['a']['n']

    i = 1
    filename, fileext = os.path.splitext(path)
    while path in self.hash2path.values():
      path = filename + ' (%d)' % i + fileext
      i += 1

    self.hash2path[hash] = path.encode()
  return self.hash2path[hash]
```

A little reminder about the client.getfiles() function, slightly modified from the previous article to add the decrypted attributes and key to the file objects, and wrap it into a class (see megaclient.py on GitHub):

```
def getfiles(self):
  files = self.api_req({'a': 'f', 'c': 1})
  files_dict = {}
  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), self.master_key)
      if file['t'] == 0:
        file['k'] = (key[0] ^ key[4], key[1] ^ key[5], key[2] ^ key[6], key[3] ^ key[7])
        file['iv'] = key[4:6] + (0, 0)
        file['meta_mac'] = key[6:8]
      else:
        file['k'] = key
      attributes = base64urldecode(file['a'])
      attributes = dec_attr(attributes, file['k'])
      file['a'] = attributes
    elif file['t'] == 2:
      self.root_id = file['h']
      file['a'] = {'n': 'Cloud Drive'}
    elif file['t'] == 3:
      self.inbox_id = file['h']
      file['a'] = {'n': 'Inbox'}
    elif file['t'] == 4:
      self.trashbin_id = file['h']
      file['a'] = {'n': 'Rubbish Bin'}
    files_dict[file['h']] = file
  return files_dict
```

Now we can implement our FUSE callbacks! Let's start with *getattr()*:

```
def getattr(self, path):
  if path not in self.files:
    return -errno.ENOENT

  file = self.files[path]
  st = fuse.Stat()
  st.st_atime = file['ts']
  st.st_mtime = st.st_atime
  st.st_ctime = st.st_atime
  if file['t'] == 0:
    st.st_mode = stat.S_IFREG | 0666
    st.st_nlink = 1
    st.st_size = file['s']
  else:
    st.st_mode = stat.S_IFDIR | 0755
    st.st_nlink = 2 + len([child for child in file['children'] if self.files[os.path.join(path, child)]['t'] &gt; 0])
    st.st_size = 4096
  return st
```

Our *files* dict makes it very concise and easy to write. We set the filesystem attributes of the files according to the informations given by the Mega API:

- We only know the last modification time of the file, so we set the creation, access and modification time to the same value.
- We set the other attributes according to the file type:

- In case of a file, we can fill in the file size, the number of hard links pointing to this file is only 1, and we set some convenient permissions (0666, rw-rw-rw-).
- In case of a directory, the size is 4096, the number of hard links pointing to it is 2 + its number of sub-directories (the directory itself, the '.' special file inside of it, and all the '..' special files inside its subdirectories). We set 0755 (rwxr-xr-x) permissions.

And now *readdir()*:

```
def readdir(self, path, offset):
  dirents = ['.', '..'] + self.files[path]['children']
  for r in dirents:
    yield fuse.Direntry(r)
```

Very concise too, it simply returns the list of the given directory's children, without forgetting the two special files '.' and '..'.

That's it! We can now test our fresh new filesystem:

```
julienm@rchand:~$ python MegaFS/megafs.py ~/megafs
julienm@rchand:~$ cd megafs
julienm@rchand:~/megafs$ ls -la
total 36
drwxr-xr-x   5 root    root     4096 janv. 29 16:44 .
drwxr-xr-x 101 julienm julienm 20480 janv. 29 16:44 ..
drwxr-xr-x   4 root    root     4096 janv. 19 18:45 Cloud Drive
drwxr-xr-x   2 root    root     4096 janv. 19 18:45 Inbox
drwxr-xr-x   2 root    root     4096 janv. 19 18:45 Rubbish Bin
julienm@rchand:~/megafs$ cd Cloud\ Drive/
julienm@rchand:~/megafs/Cloud Drive$ ls -la
total 612656
drwxr-xr-x 4 root root      4096 janv. 19 18:45 .
drwxr-xr-x 5 root root      4096 janv. 29 16:44 ..
-rw-rw-rw- 1 root root   5951970 janv. 21 12:48 Call Me Maybe vs She Wolf (YaYa Mashup).mp3
-rw-rw-rw- 1 root root     18599 janv. 29 14:09 epicwin (1).png
-rw-rw-rw- 1 root root     18599 janv. 29 14:08 epicwin (2).png
```

```
-rw-rw-rw- 1 root root     18599 janv. 28 02:49 epicwin.png
drwxr-xr-x 3 root root      4096 janv. 20 21:25 lol
drwxr-xr-x 2 root root      4096 janv. 28 15:45 lol (1)
-rw-rw-rw- 1 root root     39315 janv. 24 21:58 lulz.png
-rw-rw-rw- 1 root root    270695 janv. 25 15:27 WHOOOHOOOOOO.PNG
julienm@rchand:~$ sudo umount ~/megafs
julienm@rchand:~$
```

Seems to work 🙂 All the files belong to *root:root*, because we did not provide a *uid* and *gid* in our *getattr()* method, but we will handle that later. In the next article, we will see how to open and read files, so that we can cp them or open them directly from our Mega mountpoint! Meanwhile, you can find the complete source code of this example and follow the project on GitHub: https://github.com/CyberjujuM/MegaFS.

Google+

Share this: Like 1 Tweet Digg reddit More

This entry was posted in Uncategorized on January 29, 2013 [/web/20140402072205/http://julien-marchand.fr/blog/megafs-a-fuse-filesystem-wrapper-for-mega-part-1-listing-files/] .

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:

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

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

```
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:

```
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 = ''
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)
```

```python
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})

  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+

Share this: Like 26 Tweet Digg reddit More

This entry was posted in Uncategorized on January 28, 2013 [/web/20140402072205/http://julien-marchand.fr/blog/using-mega-api-with-python-examples/] .