Escher Documentation

Release 0.4.0

Emarsys

Sep 27, 2017

Contents

Escher is a **stateless HTTP request signing spec** to allow secure authorization and request validation. It adds an additional security layer and an authentication layer over HTTPS. The algorithm is based on Amazon's AWS4 authentication. The goal was implementing a protocol where request's integrity is validated. It is impossible to be modified without knowing the secret.

Escher is great for creating **secure REST API servers**, both the client side **request signing** process, and the server side **validation** and **authentication** processes are implemented. The protocol also provides a solution for **presigning URLs** with expiration time.

The status is **work in progress**. We're working on the documentation and finishing the first implementations.

Your help will be *much welcomed*, we are especially interested in **code reviews and audits**, **new library implementations in different languages**, **writing the documentation** and **promotion**.

Feel free to join and discuss at Escher's general mailing list, and follow our Twitter user @escherauth.

CHAPTER 1

Announcement

You can check out our Escher announcement on NordicAPIs:

CHAPTER 2

Contents

Specification

This document defines the Escher HTTP request signing framework.

Abstract

The Escher HTTP request signing framework enables a third-party client to securely access an HTTP service. It defines the creation of an authorization header, and includes message integrity checking, and can prevent replaying messages. It is designed to be used with the HTTPS protocol.

The framework is based on Amazon's AWS4 authentication, and is compatible with Amazon services using their AWS4 protocol.

Status of This Memo

This is a work in progress specification.

Copyright Notice

1. Introduction

The Escher HTTP request signing framework is intended to provide a secure way for clients to sign HTTP requests, and servers to check the integrity of these messages. The goal of the protocol is to introduce an authentication solution for REST API services, that is more secure than the currently available protocols.

RFC 2617 (HTTP Authentication) defines Basic and Digest Access Authentication. They're widely used, but Basic Access Authentication doesn't encrypt the secret and doesn't add integrity checks to the requests. Digest Access

Authentication sends the secret encrypted, but the algorithm with creating a checksum with a nonce and using md5 should not be considered highly secure these days, and as with Basic Access Authentication, there's no way to check the integrity of the message.

RFC 6749 (OAuth 2.0 Authorization) enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. This is not helpful for a machine-to-machine communication situation, like a REST API authentication, because typically there's no third-party user involved. Additionally, after a token is obtained from the authorization endpoint, it is used with no encryption, and doesn't provide integration checking, or prevent repeating messages. OAuth 2.0 is a stateful protocol which needs a database to store the tokens for client sessions.

Amazon and other service providers created protocols addressing these issues, however there is no public standard with open source implementations available from them. As Escher is based on a publicly documented, widely, in-the-wild used protocol, the specification does not include novelty techniques.

2. Signing an HTTP Request

Escher defines a stateless signature generation mechanism. The signature is calculated from the key parts of the HTTP request, a service identifier string called **credential scope**, and a **client key** and **client secret**.

The signature generation steps are: canonicalizing the request, creating a string to calculate the signature, and adding the signature to the original request.

Escher supports two hash algorithms: SHA256 and SHA512 designed by the NSA (U.S. National Security Agency).

2.1. Canonicalizing the Request

In order to calculate a checksum from the key HTTP request parts, the HTTP request method, the request URI, the query parts, the headers, and the request body have to be canonicalized. The output of the canonicalization step will be a string including the request parts separated by `LF` (line feed, "n") characters. The string will be used to calculate a checksum for the request.

2.1.1. The HTTP method

The HTTP method defined by RFC2616 (Hypertext Transfer Protocol) is case sensitive, and must be available in upper case, no transformation has to be applied:

```
POST
```

2.1.2. The Path

The path is the absolute path of the URL. Starts with a slash (`/`) character, and does not include the query part (and the question mark).

Escher follows the rules defined by RFC3986 (Uniform Resource Identifier) to normalize the path. Basically it means:

- Convert relative paths to absolute, remove redundant path components.
- URI-encode each path components:
  - the "reserved characters" defined by RFC3986 (Uniform Resource Identifier) have to be kept as they are (no encoding applied)
  - all other characters have to be percent encoded, including `SPACE` (to `%20`, instead of `+`)

  - non-ASCII, UTF-8 characters should be percent encoded to 2 or more pieces (`á` to `%C3%A1`)
  - percent encoded hexadecimal numbers have to be upper cased (eg: `a%c2%b1b` to `a%C2%B1b`)
- Normalize empty paths to `/`.

For example:

```
/path/resource/
```

2.1.3. The Query String

RFC3986 (Uniform Resource Identifier) should provide guidance for canonicalization of the query string, but here's the complete list of the rules to be applied:

- URI-encode each query parameter names and values
  - the "reserved characters" defined by RFC3986 (Uniform Resource Identifier) have to be kept as they are (no encoding applied)
  - all other characters have to be percent encoded, including `SPACE` (to `%20`, instead of `+`)
  - non-ASCII, UTF-8 characters should be percent encoded to 2 or more pieces (`á` to `%C3%A1`)
  - percent encoded hexadecimal numbers have to be upper cased (eg: `a%c2%b1b` to `a%C2%B1b`)
- Normalize empty query strings to empty string.
- Sort query parameters by the encoded parameter names (ASCII order).
- Do not shorten parameter values if their parameter name is the same (`key=B&key=A` is a valid output), the order of parameters in a URL may be significant (this is not defined by the HTTP standard).
- Separate parameter names and values by = signs, include = for empty values, too
- Separate parameters by `&`

For example:

```
foo=bar&abc=efg
```

2.1.4. The Headers

To canonicalize the headers, the following rules have to be followed:

- Lower case the header names.
- Separate header names and values by a `:`, with no spaces.
- Sort header names to alphabetical order (ASCII).
- Group headers with the same names into a header, and separate their values by commas, without sorting.
- Trim header values, keep all the spaces between quote characters (`"`).

For example:

```
accept:*/*
user-agent:example-client
connection:close
content-type:application/x-www-form-urlencoded
content-length:21
host:example.com
```

2.1.5. Signed Headers

The list of headers to include when calculating the signature. Lower cased value of header names, separated by `;`, like this:

```
date;host
```

2.1.6. Body Checksum

A checksum for the request body, aka the payload has to be calculated. Escher supports SHA-256 and SHA-512 algorithms for checksum calculation. If the request contains no body, an empty string has to be used as the input for the hash algorithm.

The selected algorithm will be added later to the authorization header, so the server will be able to use the same algorithm for validation.

The checksum of the body has to be presented as a lower cased hexadecimal string, for example:

```
fedcba9876543210fedcba9876543210fedcba9876543210fedcba9876543210
```

2.1.7. Concatenating the Canonicalized Parts

All the steps above produce a row of data, except the headers canonicalization, as it creates one row per headers. These have to be concatenated with `LF` (line feed, "n") characters into a string. An example:

```
POST
/path/resource/
foo=bar&abc=efg
accept:*/*
user-agent:example-client
connection:close
content-type:application/x-www-form-urlencoded
content-length:21
host:example.com
date;host
fedcba9876543210fedcba9876543210fedcba9876543210fedcba9876543210
```

2.2. Creating the Signature

The next step is creating another string which will be directly used to calculate the signature.

2.2.1. Algorithm ID

The **algorithm ID** comes from the **algo_prefix** (default value is `ESR`) and the algorithm used to calculate checksums during the signing process. The string **algo_prefix**, "HMAC", and the algorithm name should be concatenated with dashes, like this:

```
ESR-HMAC-SHA256
```

2.2.2. Long Date

The long date is the request date in the ISO 8601 *basic* format, like `YYYYMMDD + T + HHMMSS + Z`. Note that the basic format uses no punctuation. Example is:

`20141022T120000Z`

This date has to be added later, too, as a date header (default header name is `X-Escher-Date`).

2.2.3. Date and Credential Scope

Next information is the **short date**, and the **credential scope** concatenated with a `/` character. The **short date** is the request date's date part, an ISO 8601 basic formatted representation, the **credential scope** is defined by the service. Example:

`20141022/eu-vienna/yourproductname/escher_request`

This will be added later, too, as part of the authorization header (default header name is `X-Escher-Auth`).

2.2.4. Checksum of the Canonicalized Request

Take the output of step *2.1.7.*, and create a checksum from the canonicalized checksum string. This checksum has to be represented as a lower cased hexadecimal string, too. Something like this will be an output:

`0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef`

2.2.5. Concatenating the Signing String

Concatenate the outputs of steps 2.2. with `LF` characters. Example output:

```
ESR-HMAC-SHA256
20141022T120000Z
20141022/eu-vienna/yourproductname/escher_request
0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
```

2.2.6. The Signing Key

The signing key is based on the **algo_prefix**, the **client secret**, the parts of the **credential scope**, and the **request date**.

Take the **algo_prefix**, concatenate the **client secret** to it. First apply the HMAC algorithm to the **request date**, then apply the actual value on each of the **credential scope** parts (splitted at `/`). The end result is a binary signing key.

Pseudo code:

```
signing_key = algo_prefix + client_secret
signing_key = HMAC.Digest(short_request_date, signing_key)
foreach credential_scope.split('/') as scope_part
  signing_key = HMAC.Digest(scope_part, signing_key)
end_foreach
return signing_key
```

2.2.7. Create the Signature

The signature is created from the output of steps *2.2.5.* (Signing String) and *2.2.6.* (Signing Key). With the selected algorithm, create a checksum. It has to be represented as a lower cased hexadecimal string. Something like this will be an output:

`abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd`

2.3. Adding the Signature to the Request

The final step of the Escher signing process is adding the Signature to the request. Escher adds a new header to the request, by default, the header name is `X-Escher-Auth`. The header value will include the **algorithm ID** (see *2.2.1.*), the **client key**, the **short date** and the **credential scope** (see *2.2.3.*), the **signed headers** string (see *2.1.5.*) and finally the **signature** (see *2.2.7.*).

The values of this inputs have to be concatenated like this:

```
ESR-HMAC-SHA256 Credential=CLIENT_KEY/20141022/eu-vienna/yourproductname/escher_
↪request,
SignedHeaders=date;host,␣
↪Signature=abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
```

3. Presigning a URL

The URL presigning process is very similar to the request signing procedure. But for a URL, there are no headers, no request body, so the calculation of the Signature is different. Also, the Signature cannot be added to the headers, but is included as query parameters.

A significant difference is that the presigning allows defining an expiration time. By default, it is 86400 secs, 24 hours. The current time and the expiration time will be included in the URL, and the server has to check if the URL is expired.

3.1. Canonicalizing the URL to Presign

The canonicalization for URL presigning is the same process as for HTTP requests, in this section we will cover the differences only.

3.1.1. The HTTP method

The HTTP method for presigned URLs is fixed to:

```
GET
```

3.1.2. The Path

The path is coming from the URL, and the same canonicalization process has to be applied to them as for HTTP requests.

For example:

```
/path/resource/
```

3.1.3. The Query String

The query is coming from the URL, but the algorithm, credentials, date, expiration time, and signed headers have to be added to the query parts.

```
foo=bar&abc=efg
```

3.1.4. The Headers

A URL has no headers, Escher creates the Host header based on the URL's domain information, and adds it to the canonicalized request.

For example:

```
host:example.com
```

3.1.5. Signed Headers

It will be *host*, as that's the only header included. Example:

```
host
```

4. Validating Requests

TBD

Configuring Escher

You have to have some service specific constants when you would like to use Escher. If you are implementing a client, it will be provided by the service owner. If you are a service owner, you have to define them for your users.

Note: In practice, you should only define the credential scope, and keep the defaults of the other options.

Credential Scope

The most important service constant is the *credential scope*, this is the only required parameter when you are using the library. It is a slash separated, hierarchical ID, containing the service's scope. Amazon uses it to identify the data center, the service and the protocol. An Amazon example is `us-east-1/iam/aws4_request`. You should define a scope like `eu-vienna/yourproductname/escher_request`.

Clock Skew

This option defines the maximum allowed window of the difference between the clock of the client and the server, as it's quite typical that computer times are not synced well. Without this setting, even small differences could cause the invalidity of the signature.

The default is $\pm$900 secs (15 mins), it generally works well. You might want to allow smaller difference only.

Algo Prefix, Vendor Key, Authentication Header Name and Date Header Name

These options are implemented to keep the compability with the Amazon AWS4 protocol, it is not recommended to change them.

Hash Algo

Defines the hash algorithm used to calculate the signature. There are two supported values: *SHA256* and *SHA512*.

Current Time

This parameter is only for testing purposes, as tests need to have an injected time for repeatability reasons.

Possible error messages

Message: The date header is missing

Solution: The "dateHeaderName" configuration should be the same on server and client.

Message: The authorization header is missing

Solution: The "authHeaderName" configuration should be the same on server and client.

Message: The host header is missing

Solution: The client's Escher implementation is incomplete or the host header lost between the server and the client.

Message: Could not parse auth header

Solution: The client's Escher implementation is incomplete. Specification for authorization header.

Message: The host header is not signed

Solution: The client's Escher implementation is incomplete.

Message: The date header is not signed

Solution: The client's Escher implementation is incomplete.

Message: The credential scope is invalid

Solution: The "credentialScope" configuration should be the same on server and client.

Message: Only SHA256 and SHA512 hash algorithms are allowed

Solution: The client's Escher implementation is incomplete. Escher only supports these hash algorithms.

Message: The authorization header's shortDate does not match with the request date

Solution: The client's Escher implementation is incomplete. The authorization header's short date is not equal with the request date header.

Message: The request date is not within the accepted time range

Solution: The server's or client's time is out of sync. Use an NTP (Network Time Protocol) client or a similar solution to set the accurate time.

Message: Invalid Escher key

Solution: The client is using an Escher key missing on the server.

Message: The signatures do not match

Solution: The most likely problem is either the Escher secret is wrong or the signed and sent requests are different.

Use Cases

We have collected some use cases where we think Escher is a great fit.

REST API Authentication

There are popular solutions for API authentication, but we think Escher is the best fit, because

- It's stateless,
- the communication does not include the password,
- for a middle man, it's not possible to modify the request,
- *and* it's REST friendly by including the HTTP method in the checksum calculation.

It's Stateless

Being stateless means that the server does the authentication in a way that it doesn't have to maintain a dynamic database with API sessions.

No Password or Token Included

Using other authentication methods like HTTP Basic Authentication or OAuth, access to the communication will reveal the password or a token valid for a longer time. The Escher protocol does not include any secret, just a checksum based on your secret. This gives you a higher level of security.

No Modifications

As Escher includes all the important parts of the request when it calculates the checksum, modifying the request will invalidate the signature. It means that a middle man might be able to read the communication, but cannot modify it.

REST Friendly

Sending requests with the proper HTTP method and URIs is integral part of REST API calls. Because Escher checksum calculation includes the HTTP method and the other typical REST HTTP parameters, it won't be possible to send a REST API request differently.

Protected Resources

If you have to give temporary access within a timeframe for a resource to your users, Escher can help you with presigned URLs.

Single-sign-on, Integrations

While it's not Escher's goal to provide a single-sign-on solution, with presigned URLs you can create links between services that allow the target service to login the user properly. It works well when you open a new window for the other service, and also when you would like to display/include the service in an iframe inside your service.

Implementations

We are working hard on implementations. Please inform us if you would like to help us, and don't forget to check out our *Creating a new implementation* documentation for some instructions. All implementation should use the language's package manager (if there's something like that) to install the library, and support at least one common HTTP request representation for signing and authentication.

Official implementations

EscherRuby

- source code: EscherRuby
- gem: escher

Installation

We are supporting RubyGems as a package manager. Ruby 1.9 and newer versions are shipped with RubyGems built-in, so it should be available on your system by default. You might need an update, please check the instructions on the RubyGem's Downloads page for more information.

If RubyGems is installed, you can install the *escher* gem by:

```
gem install escher
```

Usage

From Ruby 1.9, your installed gems are autoloaded automatically, you can start using Escher after installation.

The library has 3 interfaces you can call. You can sign an HTTP request, you can presign a URL and you can validate a signed HTTP request or presigned URL (with the same method).

Signing a Request

Escher works by calculating a cryptographic signature of your request, and adding it (and other authentication information) to the request. Usually you will want to add the authentication information by appending extra headers to it. Let's say you want to send a signed POST request to http://example.com/ using the GuzzleHttp library:

```
require 'escher'
require 'net/http'

escher = Escher::Auth.new('example/credential/scope', {})

request_data = {
   method: 'GET',
   uri: '/api/examples',
   headers: [['Content-Type', 'application/json'], ['host', 'example.com']],
}

escher.sign!(request_data, { api_key_id: 'YOUR ACCESS KEY ID', api_secret: 'YOUR␣
↪SECRET' })

request = Net::HTTP::Get.new('/api/examples')
request_data[:headers].each do |header|
   request[header.first] = header.last
end

response = Net::HTTP.new('example.com').request(request)
```

Presigning a URL

In some cases you may want to send authenticated requests from a context where you cannot modify the request headers, e.g. when embedding an API generated iframe. You can however generate a presigned URL, where the authentication information is added to the query string.

```
require 'escher'
require 'net/http'

escher = Escher::Auth.new('example/credential/scope', {})
client = {api_key_id: 'YOUR ACCESS KEY ID', api_secret: 'YOUR SECRET'}

presigned_url = escher.generate_signed_url("http://example.com", client)
```

Validating a request

You can validate a request signed by the methods described above. For that you will need a database of the access keys and secrets of your clients. Escher accepts any kind of object as a key database that implements the ArrayAccess interface. (It also accepts plain arrays, however it is not recommended to use a php array for a database of API secrets - it's just there to ease testing)

```
TBD
```

EscherJS

- Source code: EscherJS

- npm package: escher-auth

Installation

We are supporting NPM as a package manager. If you have NodeJS installed NPM will be very likely installed as well. Please check out the instructions about NPM installation, if not.

If NPM is installed, you can install the *escher-auth* npm package by:

```
npm install escher-auth
```

Usage

To load the library, add the Composer autoloader to your code:

```
var Escher = require('escher-auth');
```

The library has 3 interfaces you can call. You can sign an HTTP request, you can presign a URL and you can validate a signed HTTP request or presigned URL (with the same method).

Signing a request

Escher works by calculating a cryptographic signature of your request, and adding it (and other authentication information) to the request. Usually you will want to add the authentication information by appending extra headers to it. Let's say you want to send a signed POST request to http://example.com/:

```
var escher = new Escher({
  credentialScope: 'example/credential/scope',
  accessKeyId: 'EscherExample',
  apiSecret: 'TheBeginningOfABeautifulFriendship'
});

var options = {
  host: 'example.com',
  port: 80,
  method: 'GET',
  url: '/validate_request',
  headers: [
    ['X-Escher-Date', (new Date).toUTCString()]
  ]
}

options = escher.signRequest(options, '');

http.get(options, function(resp){
  resp.on('data', function(chunk){
    console.log(chunk.toString());
  });
}).on("error", function(e){
  console.log("Got error: " + e.message);
});
```

Presigning a URL

```
var escher = new Escher({
  credentialScope: 'example/credential/scope',
  accessKeyId: 'EscherExample',
  apiSecret: 'TheBeginningOfABeautifulFriendship'
});

var presignedUrl = escher.preSignUrl('http://example.com/', 86400);
```

Validating a request

You can validate a request signed by the methods described above. For that you will need a database of the access keys and secrets of your clients. Escher accepts a function as a key database, where you can pass the client key, and it returns the client secret.

```
var escher = new Escher({
  credentialScope: 'example/credential/scope'
});

var keyDB = function(clientKey) {
  return "TheBeginningOfABeautifulFriendship";
}

escher.authenticate(request, keyDB);
```

EscherPHP

- Source code: EscherPHP
- Composer package: emartech/escher

Installation

We are supporting Composer as a package manager. Please refer to Composer's documentation if you would like to install it to your project.

If Composer is installed, you can install the *emartech-escher* composer package by:

```
composer require emartech/escher
```

It will create a *composer.json* file if does not exist, and add Escher as a required library to your project.

Usage

To load the library, add the Composer autoloader to your code:

```
<?php

require 'vendor/autoload.php';
```

The library has 3 interfaces you can call. You can sign an HTTP request, you can presign a URL and you can validate a signed HTTP request or presigned URL (with the same method).

Signing a request

Escher works by calculating a cryptographic signature of your request, and adding it (and other authentication information) to the request. Usually you will want to add the authentication information by appending extra headers to it. Let's say you want to send a signed POST request to http://example.com/ using the GuzzleHttp library:

```
<?php

$method = 'POST';
$url = 'http://example.com';
$requestBody = '{"this_is": "a_request_body"}';
$yourHeaders = array('Content-Type' => 'application/json');

$headersWithAuthInfo = Escher::create('example/credential/scope')
    ->signRequest('YOUR ACCESS KEY ID', 'YOUR SECRET', $method, $url, $requestBody,
↪$yourHeaders);
$client = new GuzzleHttp\Client();

$response = $client->post($url, array(
    'body' => $requestBody,
    'headers' => $headersWithAuthInfo
));
```

Presigning a URL

In some cases you may want to send authenticated requests from a context where you cannot modify the request headers, e.g. when embedding an API generated iframe. You can however generate a presigned URL, where the authentication information is added to the query string.

```
<?php

$presignedUrl = Escher::create('example/credential/scope')
    ->presignUrl('YOUR ACCESS KEY ID', 'YOUR SECRET', 'http://example.com');
```

Validating a request

You can validate a request signed by the methods described above. For that you will need a database of the access keys and secrets of your clients. Escher accepts any kind of object as a key database that implements the ArrayAccess interface. (It also accepts plain arrays, however it is not recommended to use a php array for a database of API secrets - it's just there to ease testing)

```
<?php

try {
    $keyDB = new ArrayObject(array(
        'ACCESS KEY OF CLIENT 1'  => 'SECRET OF CLIENT 1',
        'ACCESS KEY OF CLIENT 42' => 'SECRET OF CLIENT 42',
    ));
    Escher::create('example/credential/scope')->validateRequest($keyDB);
} catch (EscherException $ex) {
```

```
    echo 'The validation failed! ' . $ex->getMessage();
}
```

EscherJava

- source code: EscherJava
- maven package: com.emarsys.escher

Installation

Simply add the *com.emarsys.escher* artifact to your *pom.xml*.

```
<!-- https://mvnrepository.com/artifact/com.emarsys/escher -->
<dependency>
  <groupId>com.emarsys</groupId>
  <artifactId>escher</artifactId>
  <version>0.3</version>
</dependency>
```

Usage

Once the artifact is downloaded it is ready to be used.

The library has 3 interfaces you can call. You can sign an HTTP request, you can presign a URL and you can validate a signed HTTP request or presigned URL (with the same method).

Signing a Request

Escher works by calculating a cryptographic signature of your request, and adding it (and other authentication information) to the request. Usually you will want to add the authentication information by appending extra headers to it. Let's say you want to send a signed POST request to http://example.com/ using the apache HttpClient.

```
public class ClientTest {

  public static void main(String... args) {
          String url = "http://example.com/";

          HttpClient client = HttpClientBuilder.create().build();
          HttpRequestBase request = new HttpGet(url);
          request.addHeader("host", "example.com");
          request.addHeader("Content-Type", ContentType.APPLICATION_JSON.toString());
          request = signRequest(request);

          HttpResponse response = client.execute(request);
          System.out.println("Response Code : " + response.getStatusLine().
↪getStatusCode());
          System.out.println(fetchResponse(response));
  }


  private static HttpRequestBase signRequest(HttpRequestBase request) throws␣
↪EscherException {
```

```
        MyEscherRequest escherRequest = new MyEscherRequest(request, "");

        Escher escher = new Escher("eu/suite/ems_request")
                .setAuthHeaderName("X-Ems-Auth")
                .setDateHeaderName("X-Ems-Date")
                .setAlgoPrefix("EMS");

        escher.signRequest(escherRequest, "ACCESS_KEY_ID", "SECRET", Arrays.asList(
↪"Content-Type", "X-Ems-Date", "host"));

        return escherRequest.getHttpRequest();
    }


    private static String fetchResponse(HttpResponse response) throws IOException {
        try (BufferedReader rd = new BufferedReader(
                new InputStreamReader(response.getEntity().getContent()))) {
            return rd.lines().collect(Collectors.joining("\n"));
        }
    }
}


class MyEscherRequest implements EscherRequest {

    private HttpRequestBase httpRequest;
    private String body;


    public MyEscherRequest(HttpRequestBase httpRequest, String body) {
        this.httpRequest = httpRequest;
        this.body = body;
    }


    @Override
    public String getHttpMethod() {
        return httpRequest.getMethod();
    }


    @Override
    public URI getURI() {
        return httpRequest.getURI();
    }


    @Override
    public List<EscherRequest.Header> getRequestHeaders() {
        return Arrays.asList(httpRequest.getAllHeaders())
                .stream()
                .map(header -> new EscherRequest.Header(header.getName(), header.
↪getValue()))
                .collect(Collectors.toList());
    }


    @Override
```

```
  public void addHeader(String fieldName, String fieldValue) {
      httpRequest.addHeader(fieldName, fieldValue);
  }


  @Override
  public String getBody() {
      return body;
  }


  public HttpRequestBase getHttpRequest() {
      return httpRequest;
  }
}
```

The full client demo code is available here.

Presigning a URL

In some cases you may want to send authenticated requests from a context where you cannot modify the request headers, e.g. when embedding an API generated iframe. You can however generate a presigned URL, where the authentication information is added to the query string.

```
Escher escher = new Escher("eu/suite/ems_request")
          .setAuthHeaderName("X-Ems-Auth")
          .setDateHeaderName("X-Ems-Date")
          .setAlgoPrefix("EMS");

escher.presignUrl('http://example.com', "ACCESS_KEY_ID", "SECRET", 60);
```

The full client demo code is available here.

Validating a request

You can validate a request signed by the methods described above. For that you will need a database of the access keys and secrets of your clients. Escher accepts a *Map<String, String>* object as the key database.

```
public class ServerTest {

  public static void main(String... args) throws Exception {

    HttpServer server = HttpServer.create(new InetSocketAddress(8888), 0);
    server.createContext("/", exchange -> {
       String response;
       try {
            authenticate(exchange);

            response = "Everything is OK\n";

        } catch (EscherException e) {
            response = e.getMessage();
        }

        exchange.sendResponseHeaders(200, response.length());
```

```
        try (OutputStream os = exchange.getResponseBody()) {
            os.write(response.getBytes());
        }

      });
      server.setExecutor(null);
      server.start();
      System.out.println("started");
  }


  private static void authenticate(HttpExchange exchange) throws EscherException {
      EscherRequest request = new MyServerEscherRequest(exchange);
      Escher escher = new Escher("eu/suite/ems_request")
              .setAuthHeaderName("X-Ems-Auth")
              .setDateHeaderName("X-Ems-Date")
              .setAlgoPrefix("EMS");

      Map<String, String> keyDb = new HashMap<>();
      keyDb.put("ACCESS_KEY_ID", "SECRET");

      escher.authenticate(request, keyDb, new InetSocketAddress("localhost", 8888));
  }
}


class MyServerEscherRequest implements EscherRequest {

  private HttpExchange exchange;

  public MyServerEscherRequest(HttpExchange exchange) {
      this.exchange = exchange;
  }

  @Override
  public String getHttpMethod() {
      return exchange.getRequestMethod();
  }

  @Override
  public URI getURI() {
      try {
          return new URIBuilder(exchange.getRequestURI())
                  .setScheme("http")
                  .build();
      } catch (URISyntaxException e) {
          throw new RuntimeException(e);
      }
  }

  @Override
  public List<Header> getRequestHeaders() {
      List<Header> headers = new ArrayList<>();
      exchange.getRequestHeaders().forEach((fieldName, fieldValues) ->
              fieldValues.forEach(fieldValue ->
                      headers.add(new Header(fieldName, fieldValue))
              )
```

```
        );
        return headers;
    }

    @Override
    public void addHeader(String fieldName, String fieldValue) {
        throw new RuntimeException("Should not be called");
    }

    @Override
    public String getBody() {
        try (BufferedReader br = new BufferedReader(new InputStreamReader(exchange.
↪getRequestBody()))) {
            return br.lines().collect(Collectors.joining("\n"));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

The full server demo code is available here.

Creating a new implementation

Implementing the Escher protocol is not trivial, there are a lot of details you have to care about. We have created an overview, a guide and a checklist to help you if you would like to create an Escher implementation for a new programming language.

1. Read our *Overview* about the specification
2. Follow our *Guide* to start
3. Audit your code with the *Checklist for Auditing*
4. And *Publish Your Library* as an easy to install package

Overview

This page is just a short overview about this process, please read the *Specification* for details.

Signing a Request

Signing an HTTP request is only a few steps.

1. Canonicalizing a Request

Maybe this is the most difficult step, creating the canonicalized version of the HTTP requests, according to the specification, needs a lot of code.

Let's start with this HTTP request:

```
POST /path/resource/?foo=bar&abc=efg HTTP/1.1
Accept: */*
User-Agent: example-client
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 21
Host: example.com

message=Hello%20World
```

Escher creates a string from it that represents the request in a canonicalized form. The string is based on the different parts of the request, separated by line breaks (`\n`). Each part is canonicalized, for example, the header names are lower cased and ordered, the query parameters are ordered, and so.

Our example request's will look something like this after canonicalization:

```
POST
/path/resource/
foo=bar&abc=efg
accept:*/*
user-agent:example-client
connection:close
content-type:application/x-www-form-urlencoded
content-length:21
host:example.com
date;host
fedcba9876543210fedcba9876543210fedcba9876543210fedcba9876543210
```

2. Calculating the Signature

The next step includes creating an HMAC checksum from the canonicalized request string, and creating another new line separated string, now including the algorithm ID, the current time, the credential scope and the canonicalized request's checksum.

It will look something like this:

```
ESR-HMAC-SHA256
20141022T120000Z
20141022/eu-vienna/yourproductname/escher_request
0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
```

Also we are creating a key to calculate the signature. It is based on the API secret, the algo_prefix ("ESR" by default), the current date, and the credential scope.

Escher takes the date and the parts of credential scope, and calculates a checksum with each part, on the algo_prefix and API secret:

```
signing_key = algo_prefix + API_secret
key_parts = [current_date] + credential_scope.split('/')
key_parts.each { |data|
  signing_key = Digest::HMAC.digest(data, signing_key, algo)
}
return signing_key
```

At the end, with the string above and this signing_key, it calculates a checksum with HMAC.

3. Adding the Signature to the HTTP Headers

The final step is adding the signature to the request, as a new header. If the request has no host, or has no date header, these have to be added.

At the end of the process, the HTTP request will be like this:

```
POST /path/resource/?foo=bar&abc=efg HTTP/1.1
Accept: */*
User-Agent: example-client
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 21
Host: example.com
X-Escher-Date: 20141022T120000Z
X-Escher-Auth: ESR-HMAC-SHA256 Credential=API_KEY/20141022/eu-vienna/yourproductname/
↪escher_request,
  SignedHeaders=host;x-esr-date,
  Signature=abcdef01234567890abcdef01234567890abcdef01234567890abcdef0123456

message=Hello%20World
```

More

To get more information, you can read our detailed *Specification*, our check out one of the *Implementations*.

Guide

An average size of an Escher implementation should be no more than 500-1000 lines of code, depending on how granular are the methods you create. Every modern programming language has the supporting libraries (like HMAC calculation) you will need, your job is to go through the specification step-by-step and implement the methods.

We have tried our best to divide the tasks and create blocks you can implement within some hours. This way you will be able to estimate the time you need for a full implementation, and you also will have the status of your code.

Also, there are language independent test cases you can run against your code. We don't recommend starting with the tests. According to our experience, that's the harder way. At the end, you should add the tests, which will help a lot finalizing your implementation.

Building Blocks

This guide has to be finished, but already can give you some ideas. We are going to add more details. You can always check one of the implementations to get more ideas about how it works. At the moment, the EscherRuby implementation is in the best shape.

1. Implement a Generic EscherRequest Class

For a programming language, typically there's no general request representation. Some languages have a commonly used request object, but for other languages, every framework might have its own representation. It is also typical that the request object a client creates is different from the request object a controller on the server side receives.

It's a good practice creating an EscherRequest class you can instantiate with different types of request objects. It can provide an interface that Escher can use easily.

The Request class should be instantiated with request objects, and provide the following interface for the Escher implementation:

- get the HTTP method (a string)
- get the Headers (should be an array of key-value pairs)
- get the Host (a string)
- get the Path (a string starting with /, not including the question mark, query parameters)
- get the Query parameters (should be an array of key-value pairs)
- get the Body (a string)
- set a Header
- has a Header?
- give back the request (with possible modifications via header settings, but as the original object's class)

2. Create an Escher Class

Create an Escher class you can instantiate with the following parameters and their defaults. Only the credential scope is required, the others are optional:

- credential_scope - no default
- algo_prefix = 'ESR'
- vendor_key = 'Escher'
- hash_algo = 'SHA256'
- current_time = Time.now
- auth_header_name = 'X-Escher-Auth'
- date_header_name = 'X-Escher-Date'
- clock_skew = 900

3. Canonicalize a Request

Create a method that can take a request object, and create a canonicalized request, using the Escher instances' parameters. Don't care about the details yet, but handle the basic cases only. There will be a step later about implementing all the details of the specification. The output should be something like at *2.1.7. Concatenating the Canonicalized Parts*.

4. Create a Signing String

Build a signing string from the canonicalized request and the parameters of the Escher instance object. Example output is available at *2.2.5. Concatenating the Signing String*. It should be very straightforward to build the signing string.

5. Create a Signing Key

Implement the signing key creation algorithm described at *2.2.6. The Signing Key*. It should be a short code. Note that the output will be a binary value.

6. Create the Signature

Create a signature from the signing string and the signing key.

7. Add the Signature to the Headers

Call the EscherRequest's header settings, and set a proper Auth header, according to *2.3. Adding the Signature to the Request*.

8. Canonicalize the Path Properly

Now go back to your path canonicalization, and fine tune it according to the specification. Most of the time, ready-to-use libraries will be available for you, but sometimes they are not fully compatible with the standards or the Escher specification.

9. Canonicalize the Query Parameters Properly

And fine tune the query parameter canonicalization as well. If EscherRequest provides the query parameters as an array, then

10. Canonicalize the Headers Properly

TBD

Checklist for Auditing

Are you ready with your implementation, and would you like to double check if it works well and secure? First, you should run the test package, but we are also providing you this checklist you can go through and audit your code.

Request Signing

Request Canonicalization

It has to canonicalize the request properly.

- Is it correctly parsing *request URI* into *path* and *query*?
- Is it upper casing the HTTP method?
- Is it normalizing paths correctly?
- **Is it normalizing query string correctly?**
  - Must URL encode query keys
  - Must URL encode query values
  - Must handle empty value correctly
  - Must sort keys
- **Is it normalizing headers correctly?**
  - Must lowercase header names
  - Must trim value beginning

  - Must trim value ending
  - Must keep spaces when they are inside quotes
- **Is it creating the signed headers list correctly?**
  - Must only include actually signed headers
  - Must merge duplicate key values into a single one separated by commas
  - Must sort by header key/name
- **Is it normalizing signed headers list correctly?**
  - Must lowercase headers keys/names
  - Must sort headers
  - Must join headers with a semicolon
- **Is it hashing the payload correctly?**
  - Must be lowercase hexadecimal representation
- **Is it correctly formatting the canonicalized request?**
  - Must have uppercase HTTP verb on the 1st line
  - Must have the path in the 2nd line
  - Must have the canonicalized query string on the 3rd line
  - Must include headers
  - Must include an empty line after the header section
  - Must include the signed headers
  - Must include the payload hash
- **Is it correctly formatting the string to sign?**
  - **Must format the algorithm description correctly**
    - Must include a vendor prefix
    - Must include the constant "HMAC"
    - Must include the uppercase algorithm name
    - Must be separated by dashes ([PREFIX]-HMAC-[ALGO])
  - **Must format the date correctly**
    - Must be in ISO8601 format
    - Must be in GMT/UTC timezone
  - Must prefix credential scope with date (day) and a slash
  - **Must have the following structure**
    - Must include the algorithm description on the 1st line
    - Must include the date on the 2nd line
    - Must include the credential scope on the 3rd line
    - Must include the hash of the canonical request on the 4th line
- **Is it calculating the signature correctly?**

    - **Must calculate a signing key correctly**
        * Must hash combine the credential scope
        * Must NOT use hexdigest
    - Must calculate signature using HMAC and the defined hash algorithm
- **Is it assembling the Authentication header correctly?**
    - Must use the correct algorithm format
    - Must use the correct credential format
    - Must use the correct signed headers format
    - Must use the correct signature format
    - **Must have the following structure:**
        * Must include the algorithm description
        * Must have a space after algorithm description
        * Must state credential scope
        * Must have a comma and a space after credential scope
        * Must state signed headers
        * Must have a comma and a space after signed headers
        * Must state signature
- **Is the implementation consistent?**
    - Regarding date handling
    - Regarding algorithm
    - Regarding credential scope
    - Regarding full credential scope

Client must provide default constants and hide request canonicalization and signature.

Server must check signatures correctly.

Publish Your Library

If you have an implementation, even if it's a work in progress, please share the news with us. It would be great if you can also open source your project, and we can start using it together, as we are doing with other implementations.

If you agree to include your library on the site, we are going to check your source code, and do a basic audit and give some feedback. If everything goes fine, it will be promoted on this site.