MEETS SaaS API Authentication Manual

Table of Contents

Overview

This API for CirQlive MEETS SaaS allows for the creation of applications that gather information to generate various kinds of reports as well as automate various needed activities. Users with the appropriate permissions on a MEETS SaaS platform can generate access credentials to use the APIs within custom applications, whether those custom applications are developed in-house or provided by third parties. This document will cover authentication aspects for using these APIs. Please refer to the MEETS SaaS Usage Manual for information on which functions exists and how to utilize them.

Assistance

If you need API assistance and guidance please contact CirQlive API assistance. Please note that assistance is only supplied to engineers of record from organizations that have a MEETS SaaS contract which includes an API Assistance package. If your organization wishes to purchase an API Assistance package, please contact your MEETS SaaS account manager or CirQlive Sales. If you are an individual user or third party developer looking to develop your own MEETS-based application or services, please contact CirQlive Sales for your own Developer package.

Technology

The API is based upon usage via standard HTTP/1.1 and makes use of the following technology:

- HTTPS via TLS v1.2 for encryption and integrity protocol.
- JSON as response format.
- Software developed by the OpenSSL Project for use in the OpenSSL Toolkit for some of the encryption and security primitives.

TLS Hardening

MEETS SaaS supports strong levels of encryption, and if your client-side supports it, enforce high levels of security.

- Force SSL / TLS protocol used to *TLS 1.2*.
- Force allowed cipher suites to the following list:

- *TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384*
- *TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256*
- *TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384*
- *TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256*

Check with your implementation to determine how it names these cipher suites and how to enforce use of them.

In the popular OpenSSL library and its derivatives, these cipher suites are named:

- *ECDHE-ECDSA-AES256-GCM-SHA384*
- *ECDHE-ECDSA-AES128-GCM-SHA256*
- *ECDHE-RSA-AES256-GCM-SHA384*
- *ECDHE-RSA-AES128-GCM-SHA256*

If you're able to pass an *OpenSSL* cipher suite string within your application, use: *-ALL:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256*.

For additional guidance, consider reading Bulletproof SSL and TLS.

Credentials

Credentials for API usage can be generated from appropriate sections within the MEETS SaaS platform. After selecting the authentication method and properties you will be using, enter the supplied credentials and *API Access Root URL* into your application.

Credentials typically consist of a *username* and *password* or *key* and *secret* (although the concept is the same). You shouldn't share either of these, although revealing your *password* or *secret* to others is catastrophic. MEETS SaaS allows you to regenerate a *password* or *secret* in case you accidentally revealed information you shouldn't have.

Access Endpoint

All API calls are to a URL path underneath an *API Access Root URL*. The first part of an *API Access Root URL* will differ from MEETS SaaS instance to instance. The second part of an *API Access Root URL* will differ depending on the authentication method in use. Each API call will be to an *API Access Root URL* with the relative path to a specific API function appended to it. For information regarding functions, please see the MEETS SaaS Usage Manual.

It is recommended to create a function within your custom application which will generate URLs as required. This way you can ensure your application can work will multiple MEETS SaaS instances in case your organization purchases another, or if you are aiming to create a generic application or service which utilizes MEETS SaaS. Something similar to the following is suggested if you only plan to use a single authentication method:

```
FUNCTION URL_GENERATE(url_root, url_relative_api_function)
  RETURN CONCATENATE(url_root, url_relative_api_function)
END
```

If you plan on working with multiple authentication methods and will be working with the URL components individually where the authentication component of a URL is not included within your *url_root* parameter, and these path parameters do not end with a slash (*/*), then something similar to the following is suggested:

```
//The following function exists to map supported authentication methods to relative url components
//FUNCTION URL_RELATIVE_AUTHENTICATION_METHOD

FUNCTION URL_GENERATE(url_root, authentication_method, url_relative_api_function)
  RETURN CONCATENATE(url_root, '/', URL_RELATIVE_AUTHENTICATION_METHOD(authentication_method), '/', url_relative_api_function)
END
```

Errors

If some important piece of authentication information is missing, you will receive an HTTP 400 status code, along with a description of expectations. If all required information is present and correctly formed but simply incorrect, you will receive an HTTP 401 status code along with a generic authentication error.

Other kinds of errors will return the appropriate 4xx and 5xx status codes. Please see the MEETS SaaS API Usage Manual for more information.

Authentication Methods

The following various authentication methods are supported:

- Basic Authentication for ubiquitous but weakly secured access.
- Digest Authentication for commonly available reasonably strong access.
- RFC 5849 Signature commonly known as OAuth 1.0 (single-legged), and the same protocol underpinning LTI for popular reasonably strong access.
- Amazon Web Services Signature Version 4 (AWS Signature v4) for poplar strong access.
- CirQlive CryptoAuth for very strong access.

Please note that the strength levels are in relation to each other. All of these authentication methods are used over HTTPS, which makes any of them secure in general when used with a proper HTTPS implementation which verifies the security properties.

Comparison

To understand these technologies and pick the technology right for you, consider the following:

	Basic	Digest	RFC 5949 Signature	AWS Signature v4	CryptoAuth
Credentials Passing	Verbatim	MAC	MAC	MAC	MAC
Authentication Style	Direct	Challenge-Response	Direct	Direct	Direct
Capture-Replay Prevention	None	Very Strong	Strong	Strong	Very Strong
Message Timestamp	None	None	Second Resolution	Second Resolution	Nanosecond Resolution
Body Integrity	None	Optional	Limited	Yes	TODO
Response Verification	None	Yes	None	TODO	TODO
Hash Algorithm	None	Usually Very Weak	Usually Weak	Strong	Very Strong
Availability	Ubiquitous	Nearly Ubiquitous	Popular	Popular	CirQlive-only
Implementation Complexity	Beginner	Intermediate	Advanced	Intermediate	Very Advanced

- *Credentials Passing* — defines whether credentials are passed directly (verbatim) and can be stolen if the communication channel is insecure, or whether a message authentication code (MAC) is used which does not directly pass credentials over the network.
    - It is strongly recommended to not use *Basic Authentication* as it relies solely on the security of HTTPS to protect its credentials. If possible, force your application to only perform authentication via a stronger method to prevent man-in-the-middle attacks. If you are unable to do so, you must ensure that the certificate chain for the site is valid. In any case, if you can force a strong method such as *Digest Authentication*, or fully verify the site's certificate, you should be accessing the APIs in a mostly safe manner. If you are unable to verify the site's certificate in your application, nor can you force authentication to be something strong, then terminate immediately.
    - OAuth 2.0 and most popular implementations for 1.0 use tokens for accessing APIs. Using these tokens passes credentials verbatim over the network and is no stronger than *Basic Authentication* over HTTPS. The details in the above table for RFC 5949 Signature apply to single-legged OAuth 1.0 alone, whereas other OAuth-based technologies are weaker in terms of underlying communication security. Therefore, only the signature aspect of RFC 5949 is used by MEETS SaaS, providing a more secure method of access for these APIs which do not rely on a single point of failure as other popular OAuth-based technologies do. OAuth 2.0 using typical designs, which do not encapsulate some more complicated authentication information, has the following properties:

	OAuth 2.0
Credentials Passing	Verbatim
Authentication Style	Direct (once it is authorized)
Capture-Replay Prevention	None
Message Timestamp	None
Body Integrity	None
Response Verification	None

Hash Algorithm	None
Availability	Popular
Implementation Complexity	Advanced

- *Authentication Style* — defines whether the authentication information is directly computed and passed along, or whether it is based on a challenge containing data which must be cryptographically used as part of a response. The latter is stronger, aids *Capture-Replay Prevention*, prevents various kinds of spoofing, and offers other security benefits.
    - *Basic Authentication* and some others authentication methods are sometimes mistakenly thought of as *Challenge-Response* because those protocols challenge entry upon lack of authentication details. However these challenges provide nothing which must specifically be responded to, and therefore are not true *Challenge-Response* systems.

- *Capture-Replay Prevention* — defines whether the method prevents reuse of any past message. *Challenge-Response* systems with unique challenges include *Capture-Replay Prevention* as no message is accepted unless it is in response to a new unique challenge. *Capture-Replay Prevention* can also be gained solely on the client side with the use of *nonces*. A nonce is some data which guarantees unpredictability and uniqueness, preventing reuse of past messages and some other potential attacks.
    - The best way to ensure nonces are true nonces is to compose them of data guaranteed to be unique by containing an always incrementing counter or high-resolution *timestamp*, and unpredictable by including a significant amount of unpredictable random data. An unforgivable mistake, yet unfortunately common, is to compose nonces of only unique data or only random data without accounting for the other important property of a nonce.
    - *Digest Authentication* offers a server side nonce as part of its challenge, a client side nonce generated by the API consumer, and also has a *nonce counter* extending the nonce with additional unique data per communication, allowing for reuse of other parameters across multiple responses to an original challenge.
    - *RFC 5949 Signature* spreads nonces over two components called *oauth_nonce* (the random component) and *oauth_timestamp* in order to ensure neither crucial property of a nonce is overlooked. Unfortunately many implementations use predictable values for the random component, providing little extra protection. The separate *timestamp* allows messages to be dated to ensure old messages which were captured can be rejected if used at a later date.

- *Message Timestamp* — defines whether each message includes a *timestamp*, and one that is part of the message authenticity. These timestamps aid ensuring message uniqueness and can prevent old messages being delivered later by an attacker. Higher resolutions of time improve (if not guarantee) the uniqueness of a message when multiple messages may be sent close together.
    - Although *Digest Authentication* lacks message timestamps, it too can reject old messages by confirming responses are to recently issued challenges. MEETS rejects messages using older server nonces, making this option just as secure as if it had a message timestamp.

- *Body Integrity* — defines whether the body of a message is authenticated, or only the message headers. Without *Body Integrity*, an attacker can attach a different message to a valid set of authentication headers.

    - Aside from *Basic Authentication* the various methods can authenticate the full URL including the parameters within the *query string*.
        - A technique to add request *Body Integrity* to a protocol which lacks one but does authenticate its URLs is to agree on a format for adding a *message digest* (computed hash of the body) as a *query string* parameter to the request. This unfortunately does not add *Body Integrity* to responses as responses lack *URL query strings*.
    - *Digest Authentication* can only authenticate the body of a message when it is in *qop=auth-int* (authentication integrity) mode. Unfortunately this mode is poorly supported.
    - *RFC 5949 Signature* can only authenticate a request body when it is of type *application/x-www-form-urlencoded*, and cannot authenticate any kind of response bodies.

- *Response Verification* — defines whether the response sent back from the server (API results) are verifiable to be legitimate and not coming from an attacker.
    - A technique to add *Response Verification*, to any *MAC*-based protocol which lacks one is to agree on a format for adding a custom response parameter containing a MAC of all the response data.
    - *Digest Authentication* includes this feature with the response authentication component within the *Authentication-Info* response header. However many implementations fail to compute the response for verification or provide a way for getting this information.

- *Hash Algorithm* — defines whether the MAC used is based upon strong cryptographically secure algorithms.
    - MACs generally do not require extremely strong hash algorithms, as the properties of a MAC usually avoid several weaknesses a hash algorithm may have. However, it is still wise to use strong hash algorithms even as part of a MAC.
    - *Digest Authentication* only standardizes the *MD5* algorithm for use with it. This algorithm is fairly weak and should not be used beyond basic reliability needs. MEETS SaaS supports stronger hash algorithms with *Digest Authentication* as do some other implementations as well.
    - *RFC 5949 Signature* from its *MAC*-based signature methods only standardizes the *SHA-1* algorithm for use with it. This algorithm is fairly weak and should not be used beyond basic reliability needs. MEETS SaaS supports stronger hash algorithms with *RFC 5949 Signature* as do some other implementations as well.

- *Availability* — defines how common the authentication method is found within HTTP libraries or how popular the method is.
    - Although *Digest Authentication* is nearly ubiquitous, many implementations are lacking important features, or are implemented in a way that credentials will be sent to an attacker verbatim simply if the server-side asks the client to do so.

- *Implementation Complexity* — defines, in CirQlive's assessment, how complex and the skill required to implement the client-side of a particular authentication method.

Implementation

If you find yourself lacking any good existing options in your favorite programming language to provide the authentication method you selected, or are a hobbyist who enjoys implementing things, it is advisable to first familiarize yourself with the security field and learn about important practices before trying to

implement these authentication methods. This document as a whole provides important guidance throughout on implementation details along with links to additional reading. You may also consider obtaining and reading the Secure Programming Cookbook and Cryptography Engineering for good introductions to the field along with practical advice and implementation details.

Examples

There are examples below for the different authentication methods depicting how to use them in popular programming languages. These examples are provided to be educational and assist developers in getting started. Developers should review the examples and modify them to fit their needs, not necessarily to use them verbatim. The examples aim to be straight forward (wherever possible) for those knowledgeable about the given language, and will point out key ideas to be mindful of in general. It is advisable to review the linked to documentation in each example to determine how to extend them as required.

These examples all assume a basic understanding of HTTP/1.1 and its components. If your knowledge of HTTP is limited, please consult the following resources:

- Hypertext Transfer Protocol -- HTTP/1.1 (RFC 2616)
- HTTP: The Definitive Guide
- HTTP Developer's Handbook

These examples aim for minimalism in showing how to authenticate and send the basic HTTP request and get the response. They will not be overburdened with showing all kinds of more advanced techniques that can be layered on top of authentication using HTTP. You will need to consult the appropriate language and/or library's reference to learn how to perform more advanced HTTP activities.

We realize that different developers will have very different kinds of needs, and be using vastly different programming styles to accomplish their goals. Therefore, the overall structure is also minimalistic, allowing each developer to extend these examples with additional HTTP functionality as required, utilizing the techniques and methods most suited to the software being developed.

Basic Authentication

Basic Authentication is officially defined in RFC 2617, and is supported by most HTTP libraries. Due to the weak properties of this method, it is highly advisable not to use it. It is provided solely for those who do not have access to anything stronger nor can implement anything stronger themselves in a timely fashion, but still require API access and find the risks acceptable. You must ensure that the certificate chain for the site is valid when communicating to the API endpoint in order to ensure even a modicum of security.

Note: Due to *Basic Authentication* sending credentials verbatim, **you must ensure your HTTP connection only sends requests to MEETS SaaS**. Requests sent elsewhere will be giving away your access credentials. Therefore, **be sure to turn off automatic redirection in your HTTP library**, and also ensure you have your URLs going to a MEETS SaaS domain (generally **.meets.cirqlive.com*) before sending your credentials via *Basic Authentication*. **Be extremely careful using the same HTTP handle in your library for requests to multiple domains or services**.

The last path component of the *API Access Root URL* for this method is *http_basic*.

Implementation

Implementing this method requires only that you can send an HTTP header and that you can perform base64 encoding.

```
FUNCTION BASIC_AUTHENTICATION(username, password)
  HTTP_REQUEST_HEADER_APPEND('Authorization', CONCATENATE('Basic ', BASE64_ENCODE(CONCATENATE(username, ':', password))))
END
```

For the username *user*, and the password *pass*, the following request header would be sent over HTTP:

```
Authorization: Basic dXNlcjpwYXNz
```

C

Your best option in C is probably to use the libcurl library which has *Basic Authentication* support built-in. However, libcurl also contains built-in support for *Digest Authentication*, so you might as well use that method if you are able to use libcurl.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <curl/curl.h>
#include <curl/easy.h>

typedef struct
{
  uint8_t *buffer;
  size_t length;
} output_allocator;

//This function is used by curl to handle writing HTTP response bodies
//Same semantics as fwrite(), except the last parameter isn't necessarily a FILE *
static size_t output_callback(const void *ptr, size_t size, size_t nmemb, output_allocator *oa)
{
  size_t amount = size * nmemb;
  size_t amount_written = 0;
  //Increase the buffer size to the new size, returns new buffer pointer on success (freeing the old if need be), 0 otherwise
  void *p = realloc(oa->buffer, oa->length + amount);
  if (p) //Successfully enlarged
  {
    //Assign new address for buffer
    oa->buffer = p;
```

```
    //Append new data to buffer
    memcpy(oa->buffer + oa->length, ptr, amount);
    //Update new buffer size
    oa->length += amount;
    amount_written = amount;
  }
  return amount_written;
}

//Perform a request, optionally including a body, returning the HTTP response code directly and response body via the output
pointers
//A return of 0 indicates that the request could not be completed for some reason
//If returned value is not 0, caller must be sure to free the output buffer
long request(const char *username, const char *password, const char *url, const uint8_t *input, size_t input_length, uint8_t
**output, size_t *output_length)
{
  //Set initial response code to 0, meaning nothing even remotely succeeded
  long response = 0;
  CURL *curl = curl_easy_init(); //Initialize curl
  if (curl)
  {
    CURLcode request_status;
    output_allocator oa = { 0, 0 }; //Initialize output allocator

    //Use above defined function for handling output
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, output_callback);
    //Use our output allocator as the parameter to pass to output_callback()
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, &oa);

    //Turn on debugging (if you need it), comment out if you don't require it
    curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);

    curl_easy_setopt(curl, CURLOPT_HTTP_VERSION , CURL_HTTP_VERSION_1_1);

    //Turn on required SSL/TLS verification
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 1L);
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 2L); //Maximum verification
    //Force TLS 1.2 because MEETS SaaS supports it
    curl_easy_setopt(curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2);
    //Force strong hash algorithms because MEETS SaaS support them (string below is for OpenSSL / LibreSSL only)
    curl_easy_setopt(curl, CURLOPT_SSL_CIPHER_LIST, "-ALL:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256");

    curl_easy_setopt(curl, CURLOPT_HTTPAUTH, CURLAUTH_BASIC);
    curl_easy_setopt(curl, CURLOPT_USERNAME, username);
    curl_easy_setopt(curl, CURLOPT_PASSWORD, password);
```

```
    //Blindly following location is extremely dangerous with Basic Authentication, ensure it's off
    curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 0L);

    curl_easy_setopt(curl, CURLOPT_URL, url);

    if (input_length)
    {
      curl_easy_setopt(curl, CURLOPT_POST, 1L);
      curl_easy_setopt(curl, CURLOPT_POSTFIELDS, input);
      curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, input_length);
    }

    request_status = curl_easy_perform(curl);
    if (request_status == CURLE_OK) //If no errors were encountered processing this request
    {
      curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response); //Get HTTP response code

      //Point supplied output pointers to allocated buffers containing read data
      *output = oa.buffer; //Caller is responsible for freeing this buffer
      *output_length = oa.length;
    }
    else
    {
      //Free buffer just in case any was allocated. Note, it is not an error to call free(0).
      free(oa.buffer);
      fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(request_status));
    }

    curl_easy_cleanup(curl); //Cleanup curl
  }
  return response;
}
```

C++

Although C++ is mostly a superset of C, C++ provides additional features such as std::string, std::unique_ptr, and std::pair which can make code easier to work with. Therefore we provide a proper C++ example as well.

This example restructures the above C example with concepts from Effective C++ Third Edition (Items 1, 13, 20, 26), Effective STL (Items 4, 13, 16), and Effective Modern C++ (Items 5, 18).

This example utilizes the libcurl library which has *Basic Authentication* support built-in. However, libcurl also contains built-in support for *Digest Authentication*, so you might as well use that method if you are able to use libcurl.

```
#include <string>
#include <memory>
#include <utility>
#include <curl/curl.h>
#include <curl/easy.h>

namespace
{
  //This function is used by curl to handle writing HTTP response bodies
  size_t output_callback(const char *ptr, size_t size, size_t nmemb, std::string *buffer)
  {
    auto amount = size * nmemb;
    buffer->append(ptr, amount);
    return amount;
  }
}

//Perform a request, optionally including a body, returning the HTTP response code and response body or error message
//A return with the first value of 0 indicates that the request could not be completed for some reason and the second may
contain an error message
//Be prepared to catch std::bad_alloc in case of allocation failure
std::pair<long, std::string> request(const char *username, const char *password, const std::string &url, const std::string
&input = std::string())
{
  //Set initial response code to 0 and empty string, meaning nothing even remotely succeeded
  std::pair<long, std::string> response(0, std::string());
  //We will use RAII to ensure automatic cleanup of curl, even if an exception is thrown
  //curl_handle will automatically be cleaned up by curl_easy_cleanup() when its destructor from std::unique_ptr is called
  std::unique_ptr<CURL, void (*)(CURL *)> curl_handle(curl_easy_init(), curl_easy_cleanup); //Initialize curl
  CURL *curl = curl_handle.get(); //Actual pointer managed by curl_handle
  if (curl)
  {
    //Use above defined function for handling output
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, ::output_callback);
    //Use our response string as the buffer to pass to output_callback()
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, &response.second);

    //Turn on debugging (if you need it), comment out if you don't require it
    curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);

    curl_easy_setopt(curl, CURLOPT_HTTP_VERSION , CURL_HTTP_VERSION_1_1);

    //Turn on required SSL/TLS verification
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 1L);
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 2L); //Maximum verification
    //Force TLS 1.2 because MEETS SaaS supports it
```

```
    curl_easy_setopt(curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2);
    //Force strong hash algorithms because MEETS SaaS support them (string below is for OpenSSL / LibreSSL only)
    curl_easy_setopt(curl, CURLOPT_SSL_CIPHER_LIST, "-ALL:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256");

    curl_easy_setopt(curl, CURLOPT_HTTPAUTH, CURLAUTH_BASIC);
    curl_easy_setopt(curl, CURLOPT_USERNAME, username);
    curl_easy_setopt(curl, CURLOPT_PASSWORD, password);

    //Blindly following location is extremely dangerous with Basic Authentication, ensure it's off
    curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 0L);

    curl_easy_setopt(curl, CURLOPT_URL, url.c_str());

    if (!input.empty())
    {
      curl_easy_setopt(curl, CURLOPT_POST, 1L);
      curl_easy_setopt(curl, CURLOPT_POSTFIELDS, input.data());
      curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, input.length());
    }

    CURLcode request_status = curl_easy_perform(curl);
    if (request_status == CURLE_OK) //If no errors were encountered processing this request
    {
      curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response.first); //Get HTTP response code
    }
    else
    {
      response.second = "curl_easy_perform() failed: ";
      response.second.append(curl_easy_strerror(request_status));
    }
  }
  return response;
}
```

PHP

PHP like C includes the curl library, and is usually the best option for working with HTTP. It has *Basic Authentication* support built-in. However, curl in PHP also contains built-in *Digest Authentication*, so you might as well use that method if you are able to use curl in PHP.

```
<?php

//Perform a request, optionally including a body, returning an object containing the HTTP response code and response body
//A return code of 0 indicates that the request could not be completed for some reason
function request($username, $password, $url, $input = null)
```

```
{
  $response = array('data' => false, 'error' => '', 'code' => 0);
  $curl = curl_init();
  if (is_resource($curl))
  {
    //Turn on debugging (if you need it), comment out if you don't require it
    curl_setopt($curl, CURLOPT_VERBOSE, true);

    curl_setopt($curl, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_1);

    curl_setopt($curl, CURLOPT_SSL_VERIFYPEER, true);
    curl_setopt($curl, CURLOPT_SSL_VERIFYHOST, 2); //Maximum verification
    //Force TLS 1.2 because MEETS SaaS supports it
    curl_setopt($curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2);
    //Force strong hash algorithms because MEETS SaaS support them (string below is for OpenSSL / LibreSSL only)
    curl_setopt($curl, CURLOPT_SSL_CIPHER_LIST, '-ALL:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256');

    curl_setopt($curl, CURLOPT_HTTPAUTH, CURLAUTH_BASIC);
    curl_setopt($curl, CURLOPT_USERPWD, $username . ':' . $password);

    //Blindly following location is extremely dangerous with Basic Authentication, ensure it's off
    curl_setopt($curl, CURLOPT_FOLLOWLOCATION, false);

    curl_setopt($curl, CURLOPT_URL, $url);

    if ($input !== null)
    {
      curl_setopt($curl, CURLOPT_POST, true);
      curl_setopt($curl, CURLOPT_POSTFIELDS, $input);
    }

    curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);

    $response['data'] = curl_exec($curl);
    if ($response['data'] !== false)
    {
      $response['code'] = curl_getinfo($curl, CURLINFO_HTTP_CODE);
    }
    else
    {
      $response['error'] = curl_error($curl);
    }
    curl_close($curl);
  }
  return (object)$response;
```

```
}
```

Java

Java includes an HTTPS class, *HttpsURLConnection* which is somewhat clunky, but has Basic Authentication support built-in. However Java also contains built-in Digest Authentication, so you might as well use that method.

Below is a simple class that you can modify further that provides a simple interface for working with HTTP headers, *GET* and *POST*, and is *TLS Hardened*.

```
import java.io.*;
import java.util.*;
import java.net.*;
import javax.net.ssl.HttpsURLConnection;

public class HttpsBasicClient
{
  class BasicAuthenticator extends Authenticator
  {
    private String host, user, pass, lastUrl;
    public BasicAuthenticator(String host, String user, String pass)
    {
      this.host = host;
      this.user = user;
      this.pass = pass;
    }

    public PasswordAuthentication getPasswordAuthentication()
    {
      String url = this.getRequestingURL().toString();
      //Bind the authentication only to our host and only over HTTPS
      if (this.getRequestingProtocol().equals("https") && this.getRequestingHost().equals(this.host) &&
          //And only if this is not a repeat, since a repeat indicates incorrect password
          !url.equals(this.lastUrl))
      {
        this.lastUrl = url; //Save URLs for future checks to ensure repeats do not keep looping
        return new PasswordAuthentication(this.user, this.pass.toCharArray());
      }
      return null;
    }
  }

  private String username, password;
  private int lastResponseCode;
  private Map<String, String> lastResponseHeaders;
  private String lastResponse;
```

```
  private void request(String method, String urlString, String body, Map<String, String> requestHeaders) throws
MalformedURLException, IOException
  {
    this.lastResponseCode = 0;
    this.lastResponseHeaders = new HashMap<String, String>();
    this.lastResponse = new String();

    //Evil globals
    System.setProperty("https.protocols", "TLSv1.2");
    System.setProperty("https.cipherSuites",
"TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE
_RSA_WITH_AES_128_GCM_SHA256");
    System.setProperty("jdk.certpath.disabledAlgorithms", "MD2, MD5, DSA, RSA keySize < 2048");

    URL url = new URL(urlString);

    System.setProperty("http.auth.preference", "Basic");
    Authenticator.setDefault(new BasicAuthenticator(url.getHost(), this.username, this.password));

    HttpsURLConnection http = (HttpsURLConnection)url.openConnection();
    http.setRequestMethod(method);

    for (Map.Entry<String, String> entry : requestHeaders.entrySet())
    {
      http.setRequestProperty(entry.getKey(), entry.getValue());
    }

    if (!body.isEmpty())
    {
      http.setDoOutput(true);
      http.setRequestProperty("Content-Length", Integer.toString(body.length()));
      OutputStream writer = new DataOutputStream(http.getOutputStream());
      writer.write(body.getBytes("utf-8"));
    }

    http.connect();

    this.lastResponseCode = http.getResponseCode();

    for (int i = 0; ; ++i)
    {
      String key = http.getHeaderFieldKey(i);
      String value = http.getHeaderField(i);
      if (key != null)
      {
```

```
      if (value != null)
      {
        this.lastResponseHeaders.put(key.trim(), value.trim());
      }
      else
      {
        this.lastResponseHeaders.put(key.trim(), "");
      }
    }
    else if (value == null)
    {
      break;
    }
  }

  //Java thinks it's smart by splitting up responses from HTTP 4xx and 5xx
  //Java throws an exception if trying to read from the input stream on 4xx and 5xx
  InputStream is = (this.lastResponseCode >=0 && this.lastResponseCode < 400) ?
                   http.getInputStream() :
                   http.getErrorStream();
  BufferedReader br = new BufferedReader(new InputStreamReader(is));
  StringBuilder sb = new StringBuilder();
  String output;
  while ((output = br.readLine()) != null)
  {
    sb.append(output);
  }
  br.close();
  this.lastResponse = sb.toString();
}


public HttpsBasicClient(String username, String password)
{
  this.username = username;
  this.password = password;
  this.lastResponseCode = 0;
}

//Request methods
public void get(String url, Map<String, String> headers) throws MalformedURLException, IOException
{
  this.request("GET", url, "", headers);
}

public void get(String url) throws MalformedURLException, IOException
```

```
  {
    Map<String, String> headers = new HashMap<String, String>();
    this.get(url, headers);
  }

  public void post(String url, String body, Map<String, String> headers) throws MalformedURLException, IOException
  {
    this.request("POST", url, body, headers);
  }

  public void post(String url, String body) throws MalformedURLException, IOException
  {
    Map<String, String> headers = new HashMap<String, String>();
    this.post(url, body, headers);
  }

  public void post(String url, Map<String, String> parameters, Map<String, String> headers) throws MalformedURLException,
IOException
  {
    StringBuilder sb = new StringBuilder();
    for (Map.Entry<String, String> entry : parameters.entrySet())
    {
      if (sb.length() != 0)
      {
        sb.append("&");
      }
      sb.append(URLEncoder.encode(entry.getKey(), "UTF-8"));
      sb.append("=");
      sb.append(URLEncoder.encode(entry.getValue(), "UTF-8"));
    }
    headers.put("Content-Type", "application/x-www-form-urlencoded");
    this.request("POST", url, sb.toString(), headers);
  }

  public void post(String url, Map<String, String> parameters) throws MalformedURLException, IOException
  {
    Map<String, String> headers = new HashMap<String, String>();
    this.post(url, parameters, headers);
  }

  //Response methods
  public int responseCode()
  {
    return this.lastResponseCode;
  }
```

```
  public String responseBody()
  {
    return this.lastResponse;
  }

  public String responseHeader(String key)
  {
    return lastResponseHeaders.get(key);
  }

  public Map<String, String> responseHeaders()
  {
    return lastResponseHeaders;
  }
}
```

C#

C#, or .NET as a whole, offers an HTTP client, and has Basic Authentication support built-in.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.Net;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;

class HttpWithBasicAuthentication
{
  //C# requires a callback function to validate server certificates.
  //The SslPolicyErrors enumeration contains four members: None, RemoteCertificateChainErrors, RemoteCertificateNameMismatch,
and RemoteCertificateNotAvailable.
  //As long as the certificate chain has no errors, the name on the certificate matches the domain name of the URI, and there is
a certificate, this succeeds.
  static bool ValidateServerCertificate(object sender, X509Certificate certificate, X509Chain chain, SslPolicyErrors
sslPolicyErrors)
  {
    return sslPolicyErrors == SslPolicyErrors.None;
  }

  //Perform a request, optionally including a body, receiving an object containing a WebResponse object that contains the HTTP
response code and body.
  //A return code of 0 indicates that the request could not be completed for some reason.
  static bool Request(String username, String password, String url, String body, out WebResponse response)
```

```
{
  bool completed = false;
  response = null;

  if (!username.Contains(":"))
  {
    //Force TLS 1.2 because MEETS SaaS supports it.
    //.Net 3.0 and lower are not supported.
    System.Net.ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;

    HttpWebRequest webRequest = (HttpWebRequest)WebRequest.CreateHttp(url);
    CredentialCache credentials = new CredentialCache();
    //Add new credentials bound to the request url, which prevents redirects from receiving them
    credentials.Add(new Uri(url), "Basic", new NetworkCredential(username, password));
    webRequest.Credentials = credentials;
    webRequest.ServerCertificateValidationCallback = new RemoteCertificateValidationCallback(ValidateServerCertificate);

    if (!String.IsNullOrEmpty(body))
    {
      byte[] postData = Encoding.UTF8.GetBytes(body);

      webRequest.Method = "POST";
      webRequest.ContentType = "application/x-www-form-urlencoded";
      webRequest.ContentLength = postData.Length;

      Stream postStream = webRequest.GetRequestStream();
      postStream.Write(postData, 0, postData.Length);
      postStream.Close();
    }

    try
    {
      response = webRequest.GetResponse();
      completed = true;
    }
    catch (WebException ex) //WebRequest throws a WebException on HTTP 4xx and 5xx.
    {
      Console.WriteLine(String.Format("{0}: {1}", ex.Status, ex.Message));
      response = ex.Response;
      completed = true;
    }
    catch (Exception) //Catch other potential exceptions.
    {
    }
  }
  else
```

```
    {
      Console.WriteLine("A Basic Authentication username may not contain a colon.");
    }
    return completed;
  }
}
```

Digest Authentication

Digest Authentication is officially defined in RFC 2617, and is supported by most HTTP libraries. It is a good option if your library includes a good implementation for it. Note, if your library does not allow you to **force** the use of *Digest Authentication*, but its HTTP-based authentication support will allow usage with any HTTP-based authentication method as requested by the server-side then you must ensure that the certificate chain for the site is valid when communicating to the API endpoint in order to ensure even a modicum of security. Well designed HTTP libraries such as *curl* and several others allow one to force *Digest Authentication*.

Digest Authentication according to the standard uses MD5 as its hash algorithm, although some implementations support stronger methods. It also provides two flavors to use with the hash algorithm, which defines how to structure the message authentication code (MAC) used: *sess* and non-*sess*. Of the two, *sess* is the stronger. MEETS SaaS supports MD5, RIPEMD (specifically RIPEMD-160), SHA-1, the four standardized algorithms of the SHA-2 and SHA-3 (Keccak) families, Tiger, Whirlpool, and the two primary variants of BLAKE2.

Digest Authentication provides two levels of authentication, *qop=auth*, and *qop=auth-int*, where the former does not include body integrity, but the latter does. Unfortunately the latter is not well supported in libraries.

Depending on the combinations of hash algorithm, *sess*, and *qop*, the last path component of the *API Access Root URL* for this method can be one of the following:

Path component	Hash Algorithm	Default *algorithm* value	*qop* value
http_digest_md5_auth	MD5	*MD5*	*auth*
http_digest_md5_sess_auth	MD5 with *sess*	*MD5-sess*	*auth*
http_digest_md5_sess_auth-int	MD5 with *sess*	*MD5-sess*	*auth-int*
http_digest_ripemd_auth	RIPEMD / RIPEMD160 / RIPEMD-160	*RIPEMD*	*auth*
http_digest_ripemd_sess_auth	RIPEMD / RIPEMD160 / RIPEMD-160 with *sess*	*RIPEMD-sess*	*auth*
http_digest_ripemd_sess_auth-int	RIPEMD / RIPEMD160 / RIPEMD-160 with *sess*	*RIPEMD-sess*	*auth-int*
http_digest_sha1_auth	SHA / SHA1 / SHA-1	*SHA1*	*auth*
http_digest_sha1_sess_auth	SHA / SHA1 / SHA-1 with *sess*	*SHA1-sess*	*auth*
http_digest_sha1_sess_auth-int	SHA / SHA1 / SHA-1 with *sess*	*SHA1-sess*	*auth-int*

http_digest_sha224_auth	SHA224 / SHA-224	*SHA224*	*auth*
http_digest_sha224_sess_auth	SHA224 / SHA-224 with *sess*	*SHA224-sess*	*auth*
http_digest_sha224_sess_auth-int	SHA224 / SHA-224 with *sess*	*SHA224-sess*	*auth-int*
http_digest_sha256_auth	SHA256 / SHA-256	*SHA256*	*auth*
http_digest_sha256_sess_auth	SHA256 / SHA-256 with *sess*	*SHA256-sess*	*auth*
http_digest_sha256_sess_auth-int	SHA256 / SHA-256 with *sess*	*SHA256-sess*	*auth-int*
http_digest_sha384_auth	SHA384 / SHA-384	*SHA384*	*auth*
http_digest_sha384_sess_auth	SHA384 / SHA-384 with *sess*	*SHA384-sess*	*auth*
http_digest_sha384_sess_auth-int	SHA384 / SHA-384 with *sess*	*SHA384-sess*	*auth-int*
http_digest_sha512_auth	SHA512 / SHA-512	*SHA512*	*auth*
http_digest_sha512_sess_auth	SHA512 / SHA-512 with *sess*	*SHA512-sess*	*auth*
http_digest_sha512_sess_auth-int	SHA512 / SHA-512 with *sess*	*SHA512-sess*	*auth-int*
http_digest_sha3-224_auth	SHA3-224 / Keccak-224	*SHA3-224*	*auth*
http_digest_sha3-224_sess_auth	SHA3-224 / Keccak-224 with *sess*	*SHA3-224-sess*	*auth*
http_digest_sha3-224_sess_auth-int	SHA3-224 / Keccak-224 with *sess*	*SHA3-224-sess*	*auth-int*
http_digest_sha3-256_auth	SHA3-256 / Keccak-256	*SHA3-256*	*auth*
http_digest_sha3-256_sess_auth	SHA3-256 / Keccak-256 with *sess*	*SHA3-256-sess*	*auth*
http_digest_sha3-256_sess_auth-int	SHA3-256 / Keccak-256 with *sess*	*SHA3-256-sess*	*auth-int*
http_digest_sha3-384_auth	SHA3-384 / Keccak-384	*SHA3-384*	*auth*
http_digest_sha3-384_sess_auth	SHA3-384 / Keccak-384 with *sess*	*SHA3-384-sess*	*auth*
http_digest_sha3-384_sess_auth-int	SHA3-384 / Keccak-384 with *sess*	*SHA3-384-sess*	*auth-int*
http_digest_sha3-512_auth	SHA3-512 / Keccak-512	*SHA3-512*	*auth*
http_digest_sha3-512_sess_auth	SHA3-512 / Keccak-512 with *sess*	*SHA3-512-sess*	*auth*
http_digest_sha3-512_sess_auth-int	SHA3-512 / Keccak-512 with *sess*	*SHA3-512-sess*	*auth-int*
http_digest_tiger_auth	Tiger	*TIGER*	*auth*
http_digest_tiger_sess_auth	Tiger with *sess*	*TIGER-sess*	*auth*
http_digest_tiger_sess_auth-int	Tiger with *sess*	*TIGER-sess*	*auth-int*
http_digest_whirlpool_auth	Whirlpool	*WHIRLPOOL*	*auth*
http_digest_whirlpool_sess_auth	Whirlpool with *sess*	*WHIRLPOOL-sess*	*auth*
http_digest_whirlpool_sess_auth-int	Whirlpool with *sess*	*WHIRLPOOL-sess*	*auth-int*
http_digest_blake2s_auth	BLAKE2s / BLAKE2s-256	*BLAKE2S*	*auth*

http_digest_blake2s_sess_auth	BLAKE2s / BLAKE2s-256 with *sess*	*BLAKE2S-sess*	*auth*
http_digest_blake2s_sess_auth-int	BLAKE2s / BLAKE2s-256 with *sess*	*BLAKE2S-sess*	*auth-int*
http_digest_blake2b_auth	BLAKE2b / BLAKE2b-512	*BLAKE2B*	*auth*
http_digest_blake2b_sess_auth	BLAKE2b / BLAKE2b-512 with *sess*	*BLAKE2B-sess*	*auth*
http_digest_blake2b_sess_auth-int	BLAKE2b / BLAKE2b-512 with *sess*	*BLAKE2B-sess*	*auth-int*

Implementation

Implementing this method requires several steps, including the ability to send an HTTP header and receive response headers, hashing libraries, and preferably access to time and a cryptographically secure pseudo random number generator.

For the first step, you'll want to perform a simple request, probably a *HEAD* request to the desired resource. The server will respond with a *challenge*, a *WWW-Authenticate* header like the following:

```
WWW-Authenticate: Digest realm="CirQLive MEETS API", nonce="AAAAAFdrQ7sgIc8xf4uOl0j8OQUajU0OTbvUMcCd4oKKOsqAmQk3Y7P+iX0hsmR2",
algorithm="SHA512-sess", qop="auth-int", domain="/api/http_digest_sha512_sess_auth-int/"
```

The encoding of the contents of the *WWW-Authenticate* is *parameter=value* where the *value* is optionally enclosed in quotes. If a quote needs to appear within a *value* it is escaped with a backslash (\). Backslashes are also escaped with a backslash. Some servers are brittle and expect certain values to be quoted and others not, however MEETS SaaS will parse values either way. Multiple parameters are separated by commas (,). Multiple values for *qop* and *algorithm* would also be separated by a comma (example: *qop="auth,auth-int"*), although MEETS SaaS servers will never return multiple values for these parameters.

The *algorithm* parameter specifies the name of the hash algorithm being used, and whether it is with *sess* or not. However, there is disagreement between implementations regarding how to name some algorithms, creating incompatibilities. For example, one implementation might expect the SHA-1 algorithm with *sess* to be named *SHA1-sess*, while another expects *SHA-SESS*. You will need to refer to the documentation of the software you are using to determine how it identifies its algorithms. As explained in the table above, depending on the MEETS SaaS path component being accessed, it will use a specific algorithm, and has a default way of specifying it in *WWW-Authenticate* headers. If you are writing your own Digest Authentication implementation, you can use the default value. If you are working with an existing Digest Authentication implementation which does not match MEETS SaaS' default, and does not allow altering it, you can send an *X-Want-Digest-Algorithm-Name* header to instruct MEETS SaaS to use something else. Sending the header *X-Want-Digest-Algorithm-Name: SHA-SESS* to */api/http_digest_sha1_sess_auth-int/* will cause it to issue the *WWW-Authenticate* headers using *algorithm="SHA-SESS"* instead of *algorithm="SHA1-sess"*. The value in the *X-Want-Digest-Algorithm-Name* header is allowed to be up to 64 characters in size, consisting of alphanumeric characters, and can contain dashes (-) and underscores (_). Other values will be ignored. Regardless of the value sent in the *X-Want-Digest-Algorithm-Name* header, the actual algorithm used by MEETS SaaS for computation is taken from the path being used, and is validated against authentication settings.

There are another three key parameters in *WWW-Authenticate*, the *realm* parameter, the *nonce* parameter and *stale*. The *realm* parameter is passed back verbatim in response to the challenge. The *nonce* is required to compute valid requests. As long as requests are invalid, a *WWW-Authenticate* will be returned. If the request was technically valid but used an old *nonce*, then the *stale* parameter is present and set to *true* to indicate your credentials are valid and that you

computed your response to a challenge correctly, however you need to do so again with a newer *nonce*. The other parameters can be important but MEETS SaaS includes information you already know based on the authentication settings you selected. Some server implementations may include an *opaque* parameter in *WWW-Authenticate*, if present you must send that back verbatim as well.

In order to properly authenticate you must compute a *response* to the *challenge*, which is an *Authorization* request header like the following:

```
Authorization: Digest username="Jane Dough", realm="CirQLive MEETS API",
nonce="AAAAAFdrQ7sgIc8xf4uOl0j8OQUajU0OTbvUMcCd4oKKOsqAmQk3Y7P+iX0hsmR2",
uri="/api/http_digest_sha512_sess_auth-int/list_admins?limit=50", cnonce="YThhNTk1OGQ1YmFlOTA4NzJlNGE3NTRlNjBkNmQyYTI=",
nc=00000001, qop=auth-int, response="cac42fdcc12ab17465372b826da664e8", algorithm="SHA512-sess"
```

realm, *qop*, and *algorithm* must be set to the values you received from WWW-Authenticate. In general you should ensure *qop*, and *algorithm* are set to the values you want, ensuring the correct level of security, not to any whimsical values a server may challenge with, and in fact may be an attacker. If *qop* or *algorithm* specified multiple values, the response must be compatible with one of the values received for that parameter.

username — is the username part of your credentials.

cnonce — is a client-side nonce you will need to generate. See above in the authentication method comparison section for additional information on what nonces are and important details to be aware of.

nc — is the nonce counter you will be using, which is an 8 byte hexadecimal number you should be incrementing with each request, allowing you to reuse the same *nonce* and *cnonce* for multiple subsequent requests.

uri — is the full path part of a uri (from the first slash after the authority component of a URI, the domain and optional port number), **including the *query string* if present**.

response — is the MAC you will need to compute.

The following pseudo-code shows how to calculate *response* as well as general design structure:

```
//You can make your own nonce generator, but do something along these lines
FUNCTION NONCE_GENERATE()
  RETURN BASE64_ENCODE(CONCATENATE(HEX(TIMESTAMP()), SECURE_RANDOM_BYTES_GENERATOR(32)))
END

//HASH is a function like MD5 or SHA384

FUNCTION COMPUTE(http_method, is_sess, password, params_array, message_body)
  SET A1 = HEX(HASH(CONCATENATE(params_array[username], ':', params_array[realm], ':', password))
  IF is_sess
    SET A1 = HEX(HASH(CONCATENATE(A1, ':', params_array[nonce], ':', params_array[cnonce])))
  END
```

```
  IF params_array[qop] EQUALS 'auth-int'
    SET bodyhash = HEX(HASH(message_body))
    SET A2 = HEX(HASH(CONCATENATE(http_method, ':', params_array[uri], ':', bodyhash)))
  ELSE
    SET A2 = HEX(HASH(CONCATENATE(http_method, ':', params_array[uri])))
  END
  RETURN HEX(HASH(CONCATENATE(A1, ':', params_array[nonce], ':', params_array[nc], ':', params_array[cnonce], ':',
params_array[qop], ':', A2)))
END

FUNCTION AUTHORIZATION(cnonce, counter, params_received_array, username, password, http_method, uri_without_authority,
message_body)
  SET params_array = params_received_array
  //Remove unwanted parameters if present in the array
  REMOVE params_array[stale]
  REMOVE params_array[domain]
  //Add new parameters to the array
  SET params_array[username] = username
  SET params_array[uri] = uri_without_authority
  SET params_array[cnonce] = cnonce
  SET params_array[nc] = PAD_LEFT(HEX(counter), '0', 8)
  SET params_array[response] = COMPUTE(http_method, ENDS_WITH(params_received_array[algorithm], '-sess'), password,
params_array, message_body)

  //Build the authorization string
  SET auth = 'Digest '
  FOREACH key, value IN params_array
    IF key NOT_EQUALS 'nc' AND NOT_EQUALS 'qop'
      SET auth = CONCATENATE(auth, key, '="', ESCAPE(value), '", ')
    ELSE
      //Some servers are annoyed if nc or qop are quoted, since they never need to be quoted due to limited allowed values
      SET auth = CONCATENATE(auth, key, '=', value, ', ')
    END
  END
  RETURN CHOP_RIGHT(auth, 2) //Chop off the last two characters of comma and space
END

//For general usage, after you validated and parsed WWW-Authenticate into an array, you can do the following
SET cnonce = NONCE_GENERATE()
SET counter = 0

//For each request, determine the method (GET, POST, etc...), and the URI path without the authority, then:
SET counter = counter + 1
```

```
HTTP_REQUEST_HEADER_APPEND('Authorization', AUTHORIZATION(cnonce, counter, params_received_array, username, password,
http_method, uri_without_authority, message_body))
```

Upon successful authentication a server responds with an *Authentication-Info* response header like so.

```
Authentication-Info: cnonce="YThhNTk1OGQ1YmFlOTA4NzJlNGE3NTRlNjBkNmQyYTI=", nc="00000001", qop="auth-int",
rspauth="30328fa3c870441221b0cd636cf92a4c"
```

You should verify the parameters *cnonce*, *nc*, and *qop* all contain the values passed to the server. The *rspauth* parameter can be used to verify the response. Compute a *response* for authorization as you normally would, except use the existing counter value and the body received from the server, and leave the *http_method* blank (empty string). If it matches the value in *rspauth*, then the response is legitimate.

A server may optionally include a *nextnonce* parameter within *Authentication-Info*. If it does so, update your *nonce* to its value within your initially received *params_received_array*, and then continue as usual for requests that follow.

When updating a *nonce*, due to *nextnonce* or *stale*, it is probably a good idea to regenerate your *cnonce* and reset the nonce counter (*nc*).

C

Your best option in C is probably to use the libcurl library which has *Digest Authentication* support built-in. Unfortunately its *qop=auth-int* support does not work when a body is present. You can use *sess* though.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <curl/curl.h>
#include <curl/easy.h>

typedef struct
{
  uint8_t *buffer;
  size_t length;
} output_allocator;

//This function is used by curl to handle writing HTTP response bodies
//Same semantics as fwrite(), except the last parameter isn't necessarily a FILE *
static size_t output_callback(const void *ptr, size_t size, size_t nmemb, output_allocator *oa)
{
  size_t amount = size * nmemb;
  size_t amount_written = 0;
  //Increase the buffer size to the new size, returns new buffer pointer on success (freeing the old if need be), 0 otherwise
  void *p = realloc(oa->buffer, oa->length + amount);
```

```
  if (p) //Successfully enlarged
  {
    //Assign new address for buffer
    oa->buffer = p;
    //Append new data to buffer
    memcpy(oa->buffer + oa->length, ptr, amount);
    //Update new buffer size
    oa->length += amount;
    amount_written = amount;
  }
  return amount_written;
}

//Perform a request, optionally including a body, returning the HTTP response code directly and response body via the output
pointers
//A return of 0 indicates that the request could not be completed for some reason
//If returned value is not 0, caller must be sure to free the output buffer
long request(const char *username, const char *password, const char *url, const uint8_t *input, size_t input_length, uint8_t
**output, size_t *output_length)
{
  //Set initial response code to 0, meaning nothing even remotely succeeded
  long response = 0;
  CURL *curl = curl_easy_init(); //Initialize curl
  if (curl)
  {
    CURLcode request_status;
    output_allocator oa = { 0, 0 }; //Initialize output allocator

    //Use above defined function for handling output
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, output_callback);
    //Use our output allocator as the parameter to pass to output_callback()
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, &oa);

    //Turn on debugging (if you need it), comment out if you don't require it
    curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);

    curl_easy_setopt(curl, CURLOPT_HTTP_VERSION , CURL_HTTP_VERSION_1_1);

    //Turn on required SSL/TLS verification
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 1L);
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 2L); //Maximum verification
    //Force TLS 1.2 because MEETS SaaS supports it
    curl_easy_setopt(curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2);
    //Force strong hash algorithms because MEETS SaaS support them (string below is for OpenSSL / LibreSSL only)
    curl_easy_setopt(curl, CURLOPT_SSL_CIPHER_LIST, "-ALL:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256");
```

```
    //Unfortunately we cannot also enforce the various suboptions of Digest
    curl_easy_setopt(curl, CURLOPT_HTTPAUTH, CURLAUTH_DIGEST);
    curl_easy_setopt(curl, CURLOPT_USERNAME, username);
    curl_easy_setopt(curl, CURLOPT_PASSWORD, password);

    curl_easy_setopt(curl, CURLOPT_URL, url);

    if (input_length)
    {
      curl_easy_setopt(curl, CURLOPT_POST, 1L);
      curl_easy_setopt(curl, CURLOPT_POSTFIELDS, input);
      curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, input_length);
    }

    request_status = curl_easy_perform(curl);
    if (request_status == CURLE_OK) //If no errors were encountered processing this request
    {
      curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response); //Get HTTP response code

      //Point supplied output pointers to allocated buffers containing read data
      *output = oa.buffer; //Caller is responsible for freeing this buffer
      *output_length = oa.length;
    }
    else
    {
      //Free buffer just in case any was allocated. Note, it is not an error to call free(0).
      free(oa.buffer);
      fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(request_status));
    }

    curl_easy_cleanup(curl); //Cleanup curl
  }
  return response;
}
```

C++

Although C++ is mostly a superset of C, C++ provides additional features such as std::string, std::unique_ptr, and std::pair which can make code easier to work with. Therefore we provide a proper C++ example as well.

This example restructures the above C example with concepts from Effective C++ Third Edition (Items 1, 13, 20, 26), Effective STL (Items 4, 13, 16), and Effective Modern C++ (Items 5, 18).

This example utilizes the libcurl library which has *Digest Authentication* support built-in. Unfortunately its *qop=auth-int* support does not work when a body is present. You can use *sess* though.

```
#include <string>
#include <memory>
#include <utility>
#include <curl/curl.h>
#include <curl/easy.h>

namespace
{
  //This function is used by curl to handle writing HTTP response bodies
  size_t output_callback(const char *ptr, size_t size, size_t nmemb, std::string *buffer)
  {
    auto amount = size * nmemb;
    buffer->append(ptr, amount);
    return amount;
  }
}

//Perform a request, optionally including a body, returning the HTTP response code and response body or error message
//A return with the first value of 0 indicates that the request could not be completed for some reason and the second may
contain an error message
//Be prepared to catch std::bad_alloc in case of allocation failure
std::pair<long, std::string> request(const char *username, const char *password, const std::string &url, const std::string
&input = std::string())
{
  //Set initial response code to 0 and empty string, meaning nothing even remotely succeeded
  std::pair<long, std::string> response(0, std::string());
  //We will use RAII to ensure automatic cleanup of curl, even if an exception is thrown
  //curl_handle will automatically be cleaned up by curl_easy_cleanup() when its destructor from std::unique_ptr is called
  std::unique_ptr<CURL, void (*)(CURL *)> curl_handle(curl_easy_init(), curl_easy_cleanup); //Initialize curl
  CURL *curl = curl_handle.get(); //Actual pointer managed by curl_handle
  if (curl)
  {
    //Use above defined function for handling output
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, ::output_callback);
    //Use our response string as the buffer to pass to output_callback()
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, &response.second);

    //Turn on debugging (if you need it), comment out if you don't require it
    curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);

    curl_easy_setopt(curl, CURLOPT_HTTP_VERSION , CURL_HTTP_VERSION_1_1);
```

```
    //Turn on required SSL/TLS verification
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 1L);
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 2L); //Maximum verification
    //Force TLS 1.2 because MEETS SaaS supports it
    curl_easy_setopt(curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2);
    //Force strong hash algorithms because MEETS SaaS support them (string below is for OpenSSL / LibreSSL only)
    curl_easy_setopt(curl, CURLOPT_SSL_CIPHER_LIST, "-ALL:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256");

    curl_easy_setopt(curl, CURLOPT_HTTPAUTH, CURLAUTH_DIGEST);
    curl_easy_setopt(curl, CURLOPT_USERNAME, username);
    curl_easy_setopt(curl, CURLOPT_PASSWORD, password);

    curl_easy_setopt(curl, CURLOPT_URL, url.c_str());

    if (!input.empty())
    {
      curl_easy_setopt(curl, CURLOPT_POST, 1L);
      curl_easy_setopt(curl, CURLOPT_POSTFIELDS, input.data());
      curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, input.length());
    }

    CURLcode request_status = curl_easy_perform(curl);
    if (request_status == CURLE_OK) //If no errors were encountered processing this request
    {
      curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response.first); //Get HTTP response code
    }
    else
    {
      response.second = "curl_easy_perform() failed: ";
      response.second.append(curl_easy_strerror(request_status));
    }
  }
  return response;
}
```

PHP

PHP like C includes the curl library, and is usually the best option for working with HTTP. It has *Digest Authentication* support built-in. Unfortunately its *qop=auth-int* support does not work when a body is present. You can use *sess* though.

```
<?php

//Perform a request, optionally including a body, returning an object containing the HTTP response code and response body
//A return code of 0 indicates that the request could not be completed for some reason
```

```
function request($username, $password, $url, $input = null)
{
  $response = array('data' => false, 'error' => '', 'code' => 0);
  $curl = curl_init();
  if (is_resource($curl))
  {
    //Turn on debugging (if you need it), comment out if you don't require it
    curl_setopt($curl, CURLOPT_VERBOSE, true);

    curl_setopt($curl, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_1);

    curl_setopt($curl, CURLOPT_SSL_VERIFYPEER, true);
    curl_setopt($curl, CURLOPT_SSL_VERIFYHOST, 2); //Maximum verification
    //Force TLS 1.2 because MEETS SaaS supports it
    curl_setopt($curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2);
    //Force strong hash algorithms because MEETS SaaS support them (string below is for OpenSSL / LibreSSL only)
    curl_setopt($curl, CURLOPT_SSL_CIPHER_LIST, '-ALL:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256');

    //Unfortunately we cannot also enforce the various suboptions of Digest
    curl_setopt($curl, CURLOPT_HTTPAUTH, CURLAUTH_DIGEST);
    curl_setopt($curl, CURLOPT_USERPWD, $username . ':' . $password);

    curl_setopt($curl, CURLOPT_URL, $url);

    if ($input !== null)
    {
      curl_setopt($curl, CURLOPT_POST, true);
      curl_setopt($curl, CURLOPT_POSTFIELDS, $input);
    }

    curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);

    $response['data'] = curl_exec($curl);
    if ($response['data'] !== false)
    {
      $response['code'] = curl_getinfo($curl, CURLINFO_HTTP_CODE);
    }
    else
    {
      $response['error'] = curl_error($curl);
    }
    curl_close($curl);
  }
  return (object)$response;
}
```

Java

Java includes an HTTPS class, *HttpsURLConnection* which is somewhat clunky, but has Digest Authentication support built-in. Unfortunately it does not support *qop=auth-int*. You can use *sess* though. Unlike some other implementations, Java will perform response verification if you ask it too, like the example below does.

Below is a simple class that you can modify further that provides a simple interface for working with HTTP headers, *GET* and *POST*, and is *TLS Hardened*.

```
import java.io.*;
import java.util.*;
import java.net.*;
import javax.net.ssl.HttpsURLConnection;

public class HttpsDigestClient
{
  class DigestAuthenticator extends Authenticator
  {
    private String host, user, pass, lastUrl;
    public DigestAuthenticator(String host, String user, String pass)
    {
      this.host = host;
      this.user = user;
      this.pass = pass;
    }

    public PasswordAuthentication getPasswordAuthentication()
    {
      String url = this.getRequestingURL().toString();
      //Bind the authentication only to our host and only over HTTPS and specifically for Digest
      if (this.getRequestingScheme().equals("digest") &&
          this.getRequestingProtocol().equals("https") &&
          this.getRequestingHost().equals(this.host) &&
          //And only if this is not a repeat, since a repeat indicates incorrect password
          !url.equals(this.lastUrl))
      {
        this.lastUrl = url; //Save URLs for future checks to ensure repeats do not keep looping
        return new PasswordAuthentication(this.user, this.pass.toCharArray());
      }
      return null;
    }
  }

  private String username, password;
  private int lastResponseCode;
  private Map<String, String> lastResponseHeaders;
```

```
  private String lastResponse;

  private void request(String method, String urlString, String body, Map<String, String> requestHeaders) throws
MalformedURLException, IOException
  {
    this.lastResponseCode = 0;
    this.lastResponseHeaders = new HashMap<String, String>();
    this.lastResponse = new String();

    //Evil globals
    System.setProperty("https.protocols", "TLSv1.2");
    System.setProperty("https.cipherSuites",
"TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE
_RSA_WITH_AES_128_GCM_SHA256");
    System.setProperty("jdk.certpath.disabledAlgorithms", "MD2, MD5, DSA, RSA keySize < 2048");

    URL url = new URL(urlString);

    System.setProperty("http.auth.preference", "Digest");
    System.setProperty("http.auth.digest.validateServer", "true");
    Authenticator.setDefault(new DigestAuthenticator(url.getHost(), this.username, this.password));

    HttpsURLConnection http = (HttpsURLConnection)url.openConnection();
    http.setRequestMethod(method);

    for (Map.Entry<String, String> entry : requestHeaders.entrySet())
    {
      http.setRequestProperty(entry.getKey(), entry.getValue());
    }

    if (!body.isEmpty())
    {
      http.setDoOutput(true);
      http.setRequestProperty("Content-Length", Integer.toString(body.length()));
      OutputStream writer = new DataOutputStream(http.getOutputStream());
      writer.write(body.getBytes("utf-8"));
    }

    http.connect();

    this.lastResponseCode = http.getResponseCode();

    for (int i = 0; ; ++i)
    {
      String key = http.getHeaderFieldKey(i);
      String value = http.getHeaderField(i);
```

```
    if (key != null)
    {
      if (value != null)
      {
        this.lastResponseHeaders.put(key.trim(), value.trim());
      }
      else
      {
        this.lastResponseHeaders.put(key.trim(), "");
      }
    }
    else if (value == null)
    {
      break;
    }
  }

  //Java thinks it's smart by splitting up responses from HTTP 4xx and 5xx
  //Java throws an exception if trying to read from the input stream on 4xx and 5xx
  InputStream is = (this.lastResponseCode >=0 && this.lastResponseCode < 400) ?
                   http.getInputStream() :
                   http.getErrorStream();
  BufferedReader br = new BufferedReader(new InputStreamReader(is));
  StringBuilder sb = new StringBuilder();
  String output;
  while ((output = br.readLine()) != null)
  {
    sb.append(output);
  }
  br.close();
  this.lastResponse = sb.toString();
}


public HttpsDigestClient(String username, String password)
{
  this.username = username;
  this.password = password;
  this.lastResponseCode = 0;
}

//Request methods
public void get(String url, Map<String, String> headers) throws MalformedURLException, IOException
{
  this.request("GET", url, "", headers);
}
```

```
  public void get(String url) throws MalformedURLException, IOException
  {
    Map<String, String> headers = new HashMap<String, String>();
    this.get(url, headers);
  }

  public void post(String url, String body, Map<String, String> headers) throws MalformedURLException, IOException
  {
    this.request("POST", url, body, headers);
  }

  public void post(String url, String body) throws MalformedURLException, IOException
  {
    Map<String, String> headers = new HashMap<String, String>();
    this.post(url, body, headers);
  }

  public void post(String url, Map<String, String> parameters, Map<String, String> headers) throws MalformedURLException,
IOException
  {
    StringBuilder sb = new StringBuilder();
    for (Map.Entry<String, String> entry : parameters.entrySet())
    {
      if (sb.length() != 0)
      {
        sb.append("&");
      }
      sb.append(URLEncoder.encode(entry.getKey(), "UTF-8"));
      sb.append("=");
      sb.append(URLEncoder.encode(entry.getValue(), "UTF-8"));
    }
    headers.put("Content-Type", "application/x-www-form-urlencoded");
    this.request("POST", url, sb.toString(), headers);
  }

  public void post(String url, Map<String, String> parameters) throws MalformedURLException, IOException
  {
    Map<String, String> headers = new HashMap<String, String>();
    this.post(url, parameters, headers);
  }

  //Response methods
  public int responseCode()
  {
    return this.lastResponseCode;
```

```
  }

  public String responseBody()
  {
    return this.lastResponse;
  }

  public String responseHeader(String key)
  {
    return lastResponseHeaders.get(key);
  }

  public Map<String, String> responseHeaders()
  {
    return lastResponseHeaders;
  }
}
```

C#

C# has a built-in HTTP library. However it incorrectly computes *Digest Authentication* MACs if a query string is present in URLs. You may want to consider a different option with C#.

RFC 5849 Signature

RFC 5849 Signature is as its name implies is officially defined in RFC 5849, along with companion definition from the OAuth website, and has many libraries of differing quality available in different languages. It is a reasonable option if you can use a good implementation for it.

Note, some libraries use predictable nonces, only obtaining a miniscule level of security. Unlike *Digest Authentication* there are no server-side nonces to ensure a reasonable amount of random data hides the authentication secret. It falls solely on the client-side to ensure enough random data is present.

Note, most RFC 5849 Signature (OAuth 1.0) libraries include HTTP handling, but do not expose key HTTP and SSL/TLS features such as enforcing a version of TLS to use, or to limit support to the appropriate cipher suites. Some of them will not even do any server certificate validation, meaning data transfer will not be secured, and the level of security falls upon the properties of RFC 5849 Signature itself and the quality of its implementation. Carefully review your library to ensure it passes muster. If it does not, or you are not certain it does, consider using one of the other authentication options.

Note, RFC 5849 includes the body of *POST* into its MAC iff it is defined to be a *query string*, meaning the *Content-Type* is *application/x-www-form-urlencoded*. You will need to adjust your usage and modify examples appropriately depending on whether you are sending a *query string* or other kinds of data via *POST*.

RFC 5849 *Signature* from its MAC-based signature methods only standardizes the SHA-1 algorithm for use with it. Some implementations however support stronger methods. With MEETS SaaS, you can also use SHA256 and SHA512.

Depending on the hash algorithm the last path component of the *API Access Root URL* for this method can be one of the following:

- *rfc_5849_hmac_sha1*
- *rfc_5849_hmac_sha256*
- *rfc_5849_hmac_sha512*

Implementation

Implementing this method requires a few components, including a map or associative array data structure, query string handling, percent encoding according to RFC 3986, hashing libraries with HMAC support, base64 encoding, access to accurate time, and preferably a cryptographically secure pseudo random number generator.

Note, OAuth itself is a large specification, and a full implementation of all its features will require several more components and have more variables and behavior than defined here.

There are 5 parameters you will need to specify in order to implement this method:

- *oauth_signature_method* — specifies the algorithm used to sign/*MAC* the parameters. Officially, the algorithms are *PLAINTEXT*, *RSA-SHA1*, and *HMAC-SHA1*. The first doesn't provide any security, and while the second is quite good, it requires RSA support and has some additional challenges. MEETS SaaS supports *HMAC-SHA1*, *HMAC-SHA256*, and *HMAC-SHA512*.
- *oauth_consumer_key* — specifies the *key* (alternatively, username) for the conceptual user in question.
- *oauth_timestamp* — a UNIX timestamp for the current time. Specifying this using an inaccurate clock or some other kind of timestamp (with a different epoch, unit quantity other than seconds, or non-UTC timezone) will either fail to work or not provide the appropriate security qualities.
- *oauth_nonce* — is a client-side nonce you will need to generate. See above in the authentication method comparison section for additional information on what nonces are and important details to be aware of.
- *oauth_signature* — a *MAC* computed based on the 4 aforementioned parameters, as well as the HTTP method (HEAD, GET, POST, PUT, etc…), url, POST body iff it is a *query string* (*Content-Type* is *x-www-form-urlencoded*), and the *secret* (alternatively, password). The *MAC* computational method will be described below.

Once defined, these variables can either be added to the *url query string*, or the POST body iff it is a *query string*, or sent in the HTTP *Authorization* header. If using an HTTP *Authorization* header, it must appear as follows:

```
Authorization: OAuth oauth_signature_method="HMAC-SHA512", oauth_consumer_key="Jane Dough", oauth_timestamp="1368051341",
oauth_nonce="nhQ8AwkMHeISBsCUqeznqCMu1MeC3bUr",
oauth_signature="r4laPfsQ1xAajGiXxoWM7G2bkW44zAmi1NgS2BVv0BCNYAzecl9TBtI0aaO5wkqXO4qxwnz3LVtuho9FOWvSxg%3D%3D"
```

Note: Order of the 5 parameters sent via the *Authorization* header does not matter.

Note: Unlike *Digest Authentication*'s technique for the *Authorization* string, here all parameters must have their values be percent-encoded and enclosed in double quotes (").

It is suggested to use a high-resolution time function in order to get the current time when generating the *timestamp*. The amount of seconds for the current time are placed in the *oauth_timestamp*, whereas the remaining nanoseconds, microseconds, or milliseconds can be placed into *oauth_nonce* along with random data. Something like the following:

```
FUNCTION NONCE_GENERATE()
  SET ts = TIMESTAMP()
  SET seconds = TIMESTAMP_SECOND_QUANTITY(ts)
  SET remainder = BASE64_ENCODE(CONCATENATE(HEX(TIMESTAMP_REMAINING_QUANTITY(ts)), SECURE_RANDOM_BYTES_GENERATOR(32)))
  RETURN seconds, remainder //Assign return into oauth_timestamp and oauth_nonce respectively
END
```

Or in a more flat design:

```
SET ts = TIMESTAMP()
SET auth_params_array[oauth_timestamp] = TIMESTAMP_SECOND_QUANTITY(ts)
SET auth_params_array[oauth_nonce] = BASE64_ENCODE(CONCATENATE(HEX(TIMESTAMP_REMAINING_QUANTITY(ts)),
SECURE_RANDOM_BYTES_GENERATOR(32)))
```

In order to generate the *MAC*, the auth parameters as well as any relevant *query string* parameters will need to be *normalized* as defined in RFC 5849 section 3.4.1.3.2 and combined into a single string. The general outline for this process is to take all the parameters, sort them first by their *name*, then by their *value*, percent-encode both of them, combine each name-value pair (alternatively, key-value pair) with an equal sign (=) as a separator between them, and then combine all pairs using an ampersand (&) between them. If a name-value pair has a blank value, the right side of the equal sign will be blank. If a name-value pair has a blank name, the standard does not specify what to do, although we recommend to throw away that name-value pair as it has no definite meaning.

```
FUNCTION NORMALIZE_COMBINE(params_array)
  //Build the string
  SET s = ''
  FOREACH key, value IN params_array
    IF key NOT_EQUALS ''
      SET s = CONCATENATE(s, PERCENT_ENCODE(key), '=', PERCENT_ENCODE(value), '&')
    END
  END
  RETURN CHOP_RIGHT(s, 1) //Chop off the last character of ampersand
END
```

The *url* component of the *MAC* consists of the full URL until the *query string*. Meaning if a question mark (*?*) appears in a URL, the entire segment preceding it will be the *url* that appears in the *MAC*, while the component after the question mark will be *query string* parsed (typically known as *GET* parameters) and added to the parameter list.

```
//QUERY_STRING_PARSE converts a query string into an array of key-value pairs
FUNCTION URL_SPLIT(url)
  IF CONTAINS(url, '?')
    SET get_array = QUERY_STRING_PARSE(SUBSTRING_RIGHT_OF(url, '?'))
    SET url = SUBSTRING_LEFT_OF(url, '?')
  ELSE
    SET get_array = ARRAY_EMPTY()
  END
  RETURN url, get_array
END
```

The actual *MAC* computation is as follows:

```
//method is the HTTP method, an all uppercase string
//key_client for all intents and purposes is the authentication secret or password
//key_token is not used by a simple signature based authentication and is a blank value (empty string)
//NORMALIZE_COMBINE is combination of name-values described above
//HMAC is an HMAC function such as HMAC-SHA-1 or HMAC-SHA-256
FUNCTION COMPUTE(http_method, url, get_array, auth_array, post_array, key_client, key_token)
  SET params = NORMALIZE_COMBINE(SORT(MERGE_ARRAYS(get_array, auth_array, post_array)))
  SET data = CONCATENATE(PERCENT_ENCODE(http_method), '&', PERCENT_ENCODE(url), '&', PERCENT_ENCODE(params))
  SET key = CONCATENATE(PERCENT_ENCODE(key_client), '&', PERCENT_ENCODE(key_token))
  RETURN BASE64_ENCODE(HMAC(key, data))
END
```

Usage example for use within an HTTP *Authorization* header:

```
FUNCTION AUTHORIZATION(http_method, url, body, key, secret)
  SET auth_array = ARRAY_EMPTY()
  SET auth_array[oauth_signature_method] = 'HMAC-SHA512'
  SET auth_array[oauth_consumer_key] = key
  SET auth_array[oauth_timestamp],
      auth_array[oauth_nonce] = NONCE_GENERATE()
  //Note, URL_SPLIT returns two parameters and thus the required part of the URL and the get_array is passed onto COMPUTE
  SET auth_array[oauth_signature] = COMPUTE(http_method, URL_SPLIT(url), auth_array, QUERY_STRING_PARSE(body), secret, '')


  //Build the authorization string
```

```
  SET auth = 'OAuth '
  FOREACH key, value IN auth_array
    SET auth = CONCATENATE(auth, PERCENT_ENCODE(key), '="', PERCENT_ENCODE(value), '", ')
  END
  RETURN CHOP_RIGHT(auth, 2) //Chop off the last two characters of comma and space
END

//For each request, determine the method (GET, POST, etc...) and Content-Type for the body being sent, and then:
IF content_type EQUALS 'x-www-form-urlencoded'
  SET body = message_body
ELSE
  SET body = ''
END

HTTP_REQUEST_HEADER_APPEND('Authorization', AUTHORIZATION(http_method, url, body, key, secret))
```

C

C unlike higher level languages is not inherently suited to working with complex structured data in a simple matter. Due to requirements of parsing, encoding, mapping, and other required features to almost every aspect of RFC 5849 Signature, a simple example with this authentication method is not really possible. C lacks the built-in data structures, as well as built-in functions necessary for easily working with some of the technology this authentication method is built upon. Nor do the popular C libraries that one would likely be using include the required functionality for you.

You might want to consult the following books if your knowledge of C is limited:

- Understanding and Using C Pointers
- Expert C Programming
- Mastering Algorithms with C

Here's a full implementation of RFC 5849 Signature authentication in C which uses the libcurl library for HTTP communication, and the OpenSSL library for random data, *HMAC*, and *base64 encoding*. This example is TLS hardened, supports SHA512, and works with both GET and POST where the latter is a *query string*. You can tweak this further if you need additional features.

If you're looking to clean this up, or use a more intelligent string handling approach, consider working with The Better String Library.

```
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdarg.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
```

```
#include <openssl/rand.h>
#include <openssl/hmac.h>
#include <openssl/evp.h>
#include <openssl/bio.h>
#include <openssl/buffer.h>
#include <curl/curl.h>
#include <curl/easy.h>

#ifdef CLOCK_REALTIME //clock_gettime() is available
static void time_ns(struct timespec *ts)
{
  clock_gettime(CLOCK_REALTIME, ts);
}
#elif defined(TIME_UTC)
static void time_ns(struct timespec *ts)
{
  timespec_get(ts, TIME_UTC);
}
#elif defined(__WIN32__) || defined(_WIN32) || defined(_MSC_VER)
#include <windows.h>
struct timespec
{
  int64_t tv_sec;
  uint32_t tv_nsec;
};
static void time_ns(struct timespec *ts)
{
  union
  {
    uint64_t ns100; //Time since 1601-01-01 in 100ns units
    FILETIME ft;
  } u;
  GetSystemTimeAsFileTime(&u.ft);
  u.ns100 -= UINT64_C(116444736000000000); //Offset from 1601-01-01 to 1970-01-01 in 100ns units

  ts->tv_sec = u.ns100 / UINT64_C(10000000);
  ts->tv_nsec = (u.ns100 % UINT64_C(10000000)) * 100;
}
#else
#error Your system is not POSIX compliant, nor C11 compliant, nor a popular one like Windows, seriously, go get yourself a
better system
#endif

static size_t percent_encode(char *out, const char *in, size_t in_length)
{
  char *o = out;
```

```
  const char *end = in + in_length;
  for (; in < end; ++in)
  {
    if (isalnum(*in) || (*in == '-') || (*in == '.') || (*in == '_') || (*in == '~'))
    {
      *o++ = *in;
    }
    else
    {
      static const char hex[] = "0123456789ABCDEF";
      *o++ = '%';
      *o++ = hex[*in >> 4];
      *o++ = hex[*in & 0xF];
    }
  }
  return o - out;
}

static size_t percent_encoded_length(const char *s, size_t len)
{
  size_t out_size = 0;
  const char *end = s + len;
  for (; s < end; ++s)
  {
    out_size += (isalnum(*s) || (*s == '-') || (*s == '.') || (*s == '_') || (*s == '~')) ? 1 : 3;
  }
  return out_size;
}

static char *percent_decode(const char *in, size_t in_length, size_t *out_length)
{
  char *out;
  const char *s = in, *end = s + in_length;
  *out_length = 0;
  //Pass 1, determine output size
  while (s < end)
  {
    ++*out_length;
    if ((*s != '%') || (s+2 >= end) || !isxdigit(s[1]) || !isxdigit(s[2]))
    {
      ++s;
    }
    else
    {
      s += 3;
    }
```

```
  }
  //Pass 2, generate output
  out = malloc(*out_length);
  if (out)
  {
    char hex[3] = { 0 };
    char *o = out;
    s = in; //Reset pointer
    while (s < end)
    {
      if ((*s != '%') || (s+2 >= end) || !isxdigit(s[1]) || !isxdigit(s[2]))
      {
        *o++ = *s++;
      }
      else
      {
        hex[0] = s[1];
        hex[1] = s[2];
        *o++ = strtoul(hex, 0, 16);
        s += 3;
      }
    }
  }
  return out;
}

typedef struct
{
  const char *key; size_t key_length;
  const char *value; size_t value_length;
} charmap_entry;

typedef struct
{
  charmap_entry *array;
  size_t length;
} charmap;

static void map_free(charmap *cm)
{
  if (cm)
  {
    if (cm->array)
    {
      size_t i;
      for (i = 0; i < cm->length; ++i)
```

```
      {
        free((char *)cm->array[i].key);
        free((char *)cm->array[i].value);
      }
      free(cm->array);
    }
    free(cm);
  }
}

static void map_free_copy(charmap *cm)
{
  if (cm)
  {
    free(cm->array);
    free(cm);
  }
}

#define FREE(p) free((char *)p); p = 0;
#define MAP_FREE(cm) map_free(cm); cm = 0;
#define MAP_FREE_COPY(cm) map_free_copy(cm); cm = 0;

static charmap *map_merge(size_t amount, ...)
{
  charmap *out = malloc(sizeof(*out));
  if (out)
  {
    size_t i;
    va_list ap;
    //Pass 1, determine output size
    va_start(ap, amount);
    out->length = 0;
    for (i = 0; i < amount; ++i)
    {
      charmap *cm = va_arg(ap, charmap *);
      if (cm)
      {
        out->length += cm->length;
      }
    }
    va_end(ap);

    //Pass 2, generate output
    out->array = calloc(out->length, sizeof(*out->array));
    if (out->array)
```

```
    {
      charmap_entry *o = out->array;
      va_start(ap, amount);
      for (i = 0; i < amount; ++i)
      {
        charmap *cm = va_arg(ap, charmap *);
        if (cm)
        {
          memcpy(o, cm->array, cm->length * sizeof(*cm->array));
          o += cm->length;
        }
      }
      va_end(ap);
    }
    else
    {
      MAP_FREE(out);
    }
  }
  return out;
}

#ifndef MIN
#define MIN(a, b) (((a) <= (b)) ? (a) : (b))
#endif

static int map_sort_compare(const charmap_entry *e1, const charmap_entry *e2)
{
  //Find differences in keys
  int r = memcmp(e1->key, e2->key, MIN(e1->key_length, e2->key_length));
  if (r == 0) //So far equal
  {
    r = e1->key_length - e2->key_length; //Find difference in length
    if (r == 0) //So far equal
    {
      //Find differences in values
      int r = memcmp(e1->value, e2->value, MIN(e1->value_length, e2->value_length));
      if (r == 0) //So far equal
      {
        r = e1->value_length - e2->value_length; //Find difference in length
      }
    }
  }
  return r;
}
```

```
static charmap *map_sort(charmap *cm)
{
  qsort(cm->array, cm->length, sizeof(*cm->array), (int (*)(const void *, const void *))map_sort_compare);
  return cm;
}

static charmap *map_from_string(const char *s)
{
  charmap *out = malloc(sizeof(*out));
  if (out)
  {
    const char *in = s;
    out->length = 1; //There is always at least one entry
    //Pass 1, determine output entries
    while (*s)
    {
      if (*s++ == '&')
      {
        ++out->length;
      }
    }
    //Pass 2, generate output
    out->array = calloc(out->length, sizeof(*out->array));
    if (out->array)
    {
      const char *end = s, *pos = 0, *sep = 0;
      size_t i = 0;
      for (s = in; s < end; s = pos + 1, ++i)
      {
        pos = strchr(s, '&');
        sep = strchr(s, '=');
        if (!pos)
        {
          pos = end;
        }

        if (sep && sep < pos) //This parameter is key, value
        {
          out->array[i].key = percent_decode(s, sep - s, &out->array[i].key_length);
          if (out->array[i].key)
          {
            out->array[i].value = percent_decode(sep + 1, (pos - sep) - 1, &out->array[i].value_length);
            if (!out->array[i].value)
            {
              FREE(out->array[i].key);
            }
```

```
          }
        }
        else //This parameter is key only
        {
          out->array[i].key = percent_decode(s, pos - s, &out->array[i].key_length);
        }

        if (!out->array[i].key) //Allocation failed somewhere
        {
          MAP_FREE(out);
          break;
        }
      }
    }
    else
    {
      MAP_FREE(out);
    }
  }
  return out;
}

static char *map_to_string(const charmap *cm, size_t *out_size, bool include_values)
{
  char *out;
  size_t i;
  //Pass 1, determine output entries
  *out_size = 0;
  for (i = 0; i < cm->length; ++i)
  {
    if (cm->array[i].key_length)
    {
      if (*out_size)
      {
        ++*out_size; //&
      }
      *out_size += percent_encoded_length(cm->array[i].key, cm->array[i].key_length);
      if (include_values)
      {
        ++*out_size; //=
        *out_size += percent_encoded_length(cm->array[i].value, cm->array[i].value_length);
      }
    }
  }
  //Pass 2, generate output
  out = malloc(*out_size);
```

```
  if (out)
  {
    char *o = out;
    for (i = 0; i < cm->length; ++i)
    {
      if (cm->array[i].key_length)
      {
        if (o > out)
        {
          *o++ = '&';
        }
        o += percent_encode(o, cm->array[i].key, cm->array[i].key_length);
        if (include_values)
        {
          *o++ = '=';
          o += percent_encode(o, cm->array[i].value, cm->array[i].value_length);
        }
      }
    }
  }
  return out;
}

static char *map_to_string_for_authorization(const charmap_entry *ce, size_t length)
{
  const char auth[] = "Authorization: OAuth ";
  char *out;
  //Pass 1, determine output entries
  size_t out_size = sizeof(auth)-1, i;
  for (i = 0; i < length; ++i)
  {
    if (ce[i].key_length)
    {
      out_size += 4; //="",
      out_size += percent_encoded_length(ce[i].key, ce[i].key_length);
      out_size += percent_encoded_length(ce[i].value, ce[i].value_length);
    }
  }
  //Pass 2, generate output
  out = malloc(out_size);
  if (out)
  {
    char *o = out;
    memcpy(o, auth, sizeof(auth)-1);
    o += sizeof(auth)-1;
    for (i = 0; i < length; ++i)
```

```
    {
      if (ce[i].key)
      {
        o += percent_encode(o, ce[i].key, ce[i].key_length);
        *o++ = '=';
        *o++ = '"';
        o += percent_encode(o, ce[i].value, ce[i].value_length);
        *o++ = '"';
        *o++ = ',';
      }
    }
    *--o = 0;
  }
  return out;
}

static void base64_encode(const void *in, size_t len, char *out) //Ensure out is large enough beforehand
{
  BIO *b64 = BIO_new(BIO_f_base64());
  *out = 0;
  if (b64)
  {
    BIO *bio = BIO_new(BIO_s_mem());
    if (bio)
    {
      BUF_MEM *mem;
      BIO_push(b64, bio);
      BIO_set_flags(b64, BIO_FLAGS_BASE64_NO_NL);
      BIO_write(b64, in, len);
      BIO_flush(b64);
      BIO_get_mem_ptr(bio, &mem);
      memcpy(out, mem->data, mem->length);
      BIO_free(bio);
    }
    BIO_free(b64);
  }
}

bool rfc5849_compute(char *output, const char *method, const char *url, const char *input, const charmap_entry auth[4] /* Yes,
we know it's 4 */, const char *secret)
{
  int ret = false; //Failed
  charmap *get = 0;
  size_t url_length;

  //Check if $url contains a query string
```

```
const char *pos = strchr(url, '?');
if (pos)
{
  get = map_from_string(pos + 1);
  url_length = pos - url;
}
else
{
  url_length = strlen(url);
}

if (!pos || get) //No query string, or we successfully parsed it
{
  charmap *post = map_from_string(input);
  if (post)
  {
    const charmap a = { (charmap_entry *)auth, 4 }; //Yes, we know it's 4
    charmap *merged = map_merge(3, get, &a, post);
    if (merged)
    {
      size_t params_length;
      char *params = map_to_string(map_sort(merged), &params_length, true);
      if (params)
      {
        char *str;
        size_t str_len;
        const charmap_entry str_entry[] =
        {
          { method, strlen(method), 0, 0 },
          { url, url_length, 0, 0 },
          { params, params_length, 0, 0 }
        };
        const charmap str_map = { (charmap_entry *)str_entry, sizeof(str_entry) / sizeof(*str_entry) };
        str = map_to_string(&str_map, &str_len, false);
        if (str)
        {
          char *secret_encoded;
          size_t secret_length = strlen(secret);
          size_t secret_encoded_length =  percent_encoded_length(secret, secret_length);
          secret_encoded = malloc(secret_encoded_length + 1);
          if (secret_encoded)
          {
            uint8_t mac[64]; //SHA512 as its name implies output 512 bits, which is 64 bytes

            percent_encode(secret_encoded, secret, secret_length);
            secret_encoded[secret_encoded_length++] = '&';
```

```
              HMAC(EVP_sha512(), secret_encoded, secret_encoded_length, (const uint8_t *)str, str_len, mac, 0);
              base64_encode(mac, sizeof(mac), output);
              if (*mac)
              {
                ret = true; //Succeeded
              }

              FREE(secret_encoded);
            }
            FREE(str);
          }
          FREE(params);
        }
        MAP_FREE_COPY(merged);
      }
      MAP_FREE(post);
    }
    MAP_FREE(get);
  }
  return ret;
}

//Caller must free returned value
const char *rfc5849_authorization(const char *method, const char *key, const char *secret, const char *url, const char *input)
{
  const char *ret = 0;
  struct timespec ts;
  uint32_t ns; //Nanoseconds
  char time_human[20]; //Buffer for human readable time, largest is 9223372036854775807
  uint8_t nonce_raw[24];
  char nonce[32]; //We use 24 bytes raw, then base64 encode it which is ceil(24 * 1.33-), which is 32
  char signature[86]; //We are using SHA512 which is 64 bytes, then base64 encode it which is ceil(64 * 1.33-), which is 86

  time_ns(&ts);
  sprintf(time_human, "%ld", ts.tv_sec); //Convert seconds to human readable
  ns = ts.tv_nsec; //Ensure ns is in a 4 byte quantity to avoid endian issues copying empty bytes
  memcpy(nonce_raw, &ns, sizeof(ns)); //Place nanoseconds into nonce
  RAND_pseudo_bytes(nonce_raw + sizeof(ns), sizeof(nonce_raw) - sizeof(ns)); //Fill the rest with random
  base64_encode(nonce_raw, sizeof(nonce_raw), nonce); //Encode into nonce
  if (*nonce)
  {
    #define STR_C(s) s, sizeof(s)-1
    const charmap_entry auth[] = //Variables for authorization
    {
      { STR_C("oauth_signature_method"), STR_C("HMAC-SHA512") },
```

```
      { STR_C("oauth_consumer_key"), key, strlen(key) },
      { STR_C("oauth_timestamp"), time_human, sprintf(time_human, "%ld", ts.tv_sec) },
      { STR_C("oauth_nonce"), nonce, sizeof(nonce) },
      { STR_C("oauth_signature"), signature, sizeof(signature) }
    };
    #undef STR_C
    if (rfc5849_compute(signature, method, url, input, auth, secret))
    {
      ret = map_to_string_for_authorization(auth, sizeof(auth) / sizeof(*auth));
    }
  }
  return ret;
}

typedef struct
{
  uint8_t *buffer;
  size_t length;
} output_allocator;

//This function is used by curl to handle writing HTTP response bodies
//Same semantics as fwrite(), except the last parameter isn't necessarily a FILE *
static size_t output_callback(const void *ptr, size_t size, size_t nmemb, output_allocator *oa)
{
  size_t amount = size * nmemb;
  size_t amount_written = 0;
  //Increase the buffer size to the new size, returns new buffer pointer on success (freeing the old if need be), 0 otherwise
  void *p = realloc(oa->buffer, oa->length + amount);
  if (p) //Successfully enlarged
  {
    //Assign new address for buffer
    oa->buffer = p;
    //Append new data to buffer
    memcpy(oa->buffer + oa->length, ptr, amount);
    //Update new buffer size
    oa->length += amount;
    amount_written = amount;
  }
  return amount_written;
}

//Perform a request, optionally including a body, returning the HTTP response code directly and response body via the output
pointers
//A return of 0 indicates that the request could not be completed for some reason
//If returned value is not 0, caller must be sure to free the output buffer
long request(const char *key, const char *secret, const char *url, const char *input, uint8_t **output, size_t *output_length)
```

```
{
  //Set initial response code to 0, meaning nothing even remotely succeeded
  long response = 0;
  CURL *curl = curl_easy_init(); //Initilize curl
  if (curl)
  {
    const char *auth = rfc5849_authorization(input ? "POST" : "GET", key, secret, url, input ? input : "");
    if (auth)
    {
      struct curl_slist *headers = curl_slist_append(0, auth);
      if (headers)
      {
        CURLcode request_status;
        output_allocator oa = { 0, 0 }; //Initilize output allocator

        //Use above defined function for handling output
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, output_callback);
        //Use our output allocator as the parameter to pass to output_callback()
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &oa);

        //Turn on debugging (if you need it), comment out if you don't require it
        curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);

        curl_easy_setopt(curl, CURLOPT_HTTP_VERSION , CURL_HTTP_VERSION_1_1);

        //Turn on required SSL/TLS verification
        curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 1L);
        curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 2L); //Maximum verification
        //Force TLS 1.2 because MEETS SaaS supports it
        curl_easy_setopt(curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2);
        //Force strong hash algorithms because MEETS SaaS support them (string below is for OpenSSL / LibreSSL only)
        curl_easy_setopt(curl, CURLOPT_SSL_CIPHER_LIST, "-ALL:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-
RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256");

        curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);

        curl_easy_setopt(curl, CURLOPT_URL, url);

        if (input)
        {
          curl_easy_setopt(curl, CURLOPT_POST, 1L);
          curl_easy_setopt(curl, CURLOPT_POSTFIELDS, input);
          curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, strlen(input));
        }

        request_status = curl_easy_perform(curl);
```

```
        if (request_status == CURLE_OK) //If no errors were encountered processing this request
        {
          curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response); //Get HTTP response code

          //Point supplied output pointers to allocated buffers containing read data
          *output = oa.buffer; //Caller is responsible for freeing this buffer
          *output_length = oa.length;
        }
        else
        {
          //Free buffer just in case any was allocated. Note, it is not an error to call free(0).
          free(oa.buffer);
          fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(request_status));
        }
        curl_slist_free_all(headers);
      }
      free((char *)auth);
    }
    curl_easy_cleanup(curl); //Cleanup curl
  }
  return response;
}
```

C++

Here's a full implementation of RFC 5849 Signature authentication in C++ which uses the libcurl library for HTTP communication, and the OpenSSL library for random data, *HMAC*, and *base64 encoding*. This example is TLS hardened, supports SHA512, and works with both GET and POST where the latter is a *query string*. You can tweak this further if you need additional features.

This example makes heavy use of the STL, and also is written in modern C++. If you find yourself lost, the following resources are recommended:

- The C++ Standard Library
- Effective STL
- Effective C++ Third Edition
- Effective Modern C++

This example aims for clarity over using highly optimized algorithms for parsing and encoding. If you feel you want to optimize things further, obtain and read Optimized C++.

```
#include <sstream>
#include <string>
#include <map>
#include <deque>
```

```
#include <memory>
#include <chrono>
#include <utility>
#include <cstdint>
#include <cstdlib>
#include <cstring>
#include <cctype>
#include <openssl/rand.h>
#include <openssl/hmac.h>
#include <openssl/evp.h>
#include <openssl/bio.h>
#include <openssl/buffer.h>
#include <curl/curl.h>
#include <curl/easy.h>

namespace
{
  std::string percent_encode(const std::string &s)
  {
    std::string r;
    r.reserve(s.size());
    for (auto i = s.begin(); i != s.end(); ++i)
    {
      if (std::isalnum(*i) || (*i == '-') || (*i == '.') || (*i == '_') || (*i == '~'))
      {
        r += *i;
      }
      else
      {
        static const char hex[] = "0123456789ABCDEF";
        r += '%';
        r += hex[*i >> 4];
        r += hex[*i & 0xF];
      }
    }
    return r;
  }

  std::string percent_decode(const std::string &s)
  {
    std::string r;
    r.reserve(s.size());

    char hex[3] = { 0 };
    for (auto i = s.begin(); i != s.end(); )
    {
```

```
    if ((*i != '%') || (i+2 >= s.end()) || !std::isxdigit(i[1]) || !std::isxdigit(i[2]))
    {
      r += *i++;
    }
    else
    {
      hex[0] = i[1];
      hex[1] = i[2];
      r += static_cast<char>(std::strtoul(hex, 0, 16));
      i += 3;
    }
  }
  return r;
}

template <typename T>
std::deque<std::string> tokens(const std::string &s, const T &delimiters)
{
  std::deque<std::string> r;
  auto pos_last = s.find_first_not_of(delimiters, 0);
  auto pos = s.find_first_of(delimiters, pos_last);
  while ((pos != std::string::npos) || (pos_last != std::string::npos))
  {
    r.push_back(s.substr(pos_last, pos - pos_last));
    pos_last = s.find_first_not_of(delimiters, pos);
    pos = s.find_first_of(delimiters, pos_last);
  }
  return r;
}

std::map<std::string, std::string> map_from_string(const std::string &s)
{
  std::map<std::string, std::string> r;
  auto parameters(::tokens(s, '&'));
  for (auto i = parameters.begin(); i != parameters.end(); ++i)
  {
    auto pos = i->find('=');
    if (pos !=  std::string::npos)
    {
      r.insert(std::make_pair(::percent_decode(i->substr(0, pos)), percent_decode(i->substr(pos+1))));
    }
    else
    {
      r.insert(std::make_pair(::percent_decode(*i), std::string()));
    }
  }
```

```
  return r;
}

std::string map_to_string(const std::map<std::string, std::string> &m)
{
  std::string r;
  for (auto i = m.begin(); i != m.end(); ++i)
  {
    r.append(::percent_encode(i->first));
    r += '=';
    r.append(::percent_encode(i->second));
    r += '&';
  }
  if (!r.empty())
  {
    r.resize(r.size() - 1);
  }
  return r;
}

std::string map_to_string_for_authorization(const std::map<std::string, std::string> &m)
{
  std::string r("Authorization: OAuth ");
  for (auto i = m.begin(); i != m.end(); ++i)
  {
    r.append(::percent_encode(i->first));
    r.append("=\"");
    r.append(::percent_encode(i->second));
    r.append("\",");
  }
  r.resize(r.size() - 1);
  return r;
}

std::string base64_encode(const std::string &s)
{
  std::unique_ptr<BIO, int (*)(BIO *)> b64_handle(BIO_new(BIO_f_base64()), BIO_free),
                                       bio_handle(BIO_new(BIO_s_mem()), BIO_free);
  BIO *b64 = b64_handle.get(),
      *bio = bio_handle.get();
  if (b64 && bio)
  {
    BIO_push(b64, bio);
    BIO_set_flags(b64, BIO_FLAGS_BASE64_NO_NL);
    BIO_write(b64, s.data(), s.size());
    (void)BIO_flush(b64);
```

```
      BUF_MEM *mem;
      BIO_get_mem_ptr(bio, &mem);
      std::string r;
      r.append(mem->data, mem->length);
      return r;
    }
    throw std::bad_alloc();
  }

  std::string hmac_sha512(const std::string &key, const std::string &data)
  {
    std::string r;
    r.resize(64); //SHA512 as its name implies output 512 bits, which is 64 bytes
    HMAC(EVP_sha512(), key.data(), key.size(), reinterpret_cast<const uint8_t *>(data.data()), data.size(),
reinterpret_cast<uint8_t *>(&r[0]), 0);
    return r;
  }

  std::string rfc5849_compute(const char *method, const std::string &url, const std::string &input, const std::map<std::string,
std::string> &auth, const char *secret)
  {
    std::string url_base;
    std::map<std::string, std::string> get;

    //Check if url contains a query string
    auto pos = url.find('?');
    if (pos != std::string::npos)
    {
      get = ::map_from_string(url.substr(pos + 1));
      url_base = url.substr(0, pos);
    }
    else
    {
      url_base = url;
    }

    std::map<std::string, std::string> post(::map_from_string(input));
    post.insert(auth.begin(), auth.end());
    post.insert(get.begin(), get.end());
    std::string params(::map_to_string(post));
    std::ostringstream str; str << ::percent_encode(method) << '&' << ::percent_encode(url_base) << '&'
<< ::percent_encode(params);
    std::ostringstream key; key << ::percent_encode(secret) /* key_client */ << '&'; // << ::percent_encode(key_token)
    return ::base64_encode(::hmac_sha512(key.str(), str.str()));
  }
```

```
  std::string rfc5849_authorization(const char *method, const char *key, const char *secret, const std::string &url, const
std::string &input)
  {
    std::string nonce;
    nonce.resize(24);

    auto now(std::chrono::high_resolution_clock::now());
    auto duration(now.time_since_epoch());
    auto sec = std::chrono::duration_cast<std::chrono::seconds>(duration);
    std::ostringstream time_human; time_human << sec.count(); //Convert seconds to human readable
    uint32_t ns = std::chrono::duration_cast<std::chrono::nanoseconds>(duration - sec).count();
    std::memcpy(&nonce[0], &ns, sizeof(ns)); //Place nanoseconds into nonce
    RAND_pseudo_bytes(reinterpret_cast<uint8_t *>(&nonce[sizeof(ns)]), nonce.size() - sizeof(ns)); //Fill the rest with random

    std::map<std::string, std::string> auth;
    auth["oauth_signature_method"] = "HMAC-SHA512";
    auth["oauth_consumer_key"] = key;
    auth["oauth_timestamp"] = time_human.str();
    auth["oauth_nonce"] = ::base64_encode(nonce);
    auth["oauth_signature"] = ::rfc5849_compute(method, url, input, auth, secret);
    return ::map_to_string_for_authorization(auth);
  }

  //This function is used by curl to handle writing HTTP response bodies
  size_t output_callback(const char *ptr, size_t size, size_t nmemb, std::string *buffer)
  {
    auto amount = size * nmemb;
    buffer->append(ptr, amount);
    return amount;
  }
}

//Perform a request, optionally including a body, returning the HTTP response code and response body or error message
//A return with the first value of 0 indicates that the request could not be completed for some reason and the second may
contain an error message
//Be prepared to catch std::bad_alloc in case of allocation failure
std::pair<long, std::string> request(const char *key, const char *secret, const std::string &url, const std::string &input =
std::string())
{
  //Set initial response code to 0 and empty string, meaning nothing even remotely succeeded
  std::pair<long, std::string> response(0, std::string());
  //We will use RAII to ensure automatic cleanup of curl, even if an exception is thrown
  //curl_handle will automatically be cleaned up by curl_easy_cleanup() when its destructor from std::unique_ptr is called
  std::unique_ptr<CURL, void (*)(CURL *)> curl_handle(curl_easy_init(), curl_easy_cleanup); //Initialize curl
  CURL *curl = curl_handle.get(); //Actual pointer managed by curl_handle
  if (curl)
```

```
  {
    std::string auth(::rfc5849_authorization(input.empty() ? "GET" : "POST", key, secret, url, input));
    std::unique_ptr<curl_slist, void (*)(curl_slist *)> headers_handle(curl_slist_append(0, auth.c_str()), curl_slist_free_all);
    curl_slist *headers = headers_handle.get();
    if (headers)
    {
      //Use above defined function for handling output
      curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, ::output_callback);
      //Use our response string as the buffer to pass to output_callback()
      curl_easy_setopt(curl, CURLOPT_WRITEDATA, &response.second);

      //Turn on debugging (if you need it), comment out if you don't require it
      curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);

      curl_easy_setopt(curl, CURLOPT_HTTP_VERSION , CURL_HTTP_VERSION_1_1);

      //Turn on required SSL/TLS verification
      curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 1L);
      curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 2L); //Maximum verification
      //Force TLS 1.2 because MEETS SaaS supports it
      curl_easy_setopt(curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2);
      //Force strong hash algorithms because MEETS SaaS support them (string below is for OpenSSL / LibreSSL only)
      curl_easy_setopt(curl, CURLOPT_SSL_CIPHER_LIST, "-ALL:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-
RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256");

      curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);

      curl_easy_setopt(curl, CURLOPT_URL, url.c_str());

      if (!input.empty())
      {
        curl_easy_setopt(curl, CURLOPT_POST, 1L);
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS, input.data());
        curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, input.length());
      }

      CURLcode request_status = curl_easy_perform(curl);
      if (request_status == CURLE_OK) //If no errors were encountered processing this request
      {
        curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response.first); //Get HTTP response code
      }
      else
      {
        response.second = "curl_easy_perform() failed: ";
        response.second.append(curl_easy_strerror(request_status));
      }
```

```
    }
    else
    {
      throw std::bad_alloc();
    }
  }
  return response;
}
```

PHP

Method 1

PHP offers an OAuth library which can get a basic job done, but unfortunately does not allow for *TLS hardening*, nor does it work properly with POST. It currently supports the SHA-1 and SHA256 hash algorithms.

```
<?php

//Perform a request, optionally including a body, returning an object containing the HTTP response code and response body
//A return code of 0 indicates that the request could not be completed for some reason
function request($key, $secret, $url)
{
  $response = array('data' => false, 'error' => '', 'code' => 0);
  try
  {
    $oauth = new OAuth($key, $secret, OAUTH_SIG_METHOD_HMACSHA256);

    //Turn on debugging (if you need it), comment out if you don't require it
    $oauth->enableDebug();

    $oauth->fetch($url);

    $ri = $oauth->getLastResponseInfo();
    $response['code'] = $ri['http_code'];
    $response['data'] = $oauth->getLastResponse();
  }
  catch(OAuthException $e)
  {
    $response['code'] = $e->getCode();
    $response['data'] = $e->lastResponse;
    $response['error'] = $e->getMessage();
  }
  return (object)$response;
}
```

Method 2

Here's a full implementation of RFC 5849 Signature authentication in PHP which makes use of curl for HTTP communication. It is *TLS hardened*, supports SHA512, and unlike the above method, this works with both GET and POST. You can tweak this further if you need additional features.

```
<?php

//Perform a request, optionally including a body, returning an object containing the HTTP response code and response body
//A return code of 0 indicates that the request could not be completed for some reason
function request($key, $secret, $url, $input = null)
{
  $response = array('data' => false, 'error' => '', 'code' => 0);
  $curl = curl_init();
  if (is_resource($curl))
  {
    function array_sort(array $q)
    {
      ksort($q);
      return $q;
    }

    function array_from_string($query_string)
    {
      $a = array();
      parse_str($query_string, $a); //This function really needs different semantics
      return $a;
    }

    function array_to_string(array $q)
    {
      $r = '';
      foreach ($q as $key => $value)
      {
        $r .= rawurlencode($key).'='.rawurlencode($value).'&';
      }
      $r = substr($r, 0, -1);
      return $r;
    }

    function array_to_string_for_authorization(array $q)
    {
      $r = '';
      foreach ($q as $key => $value)
      {
        $r .= rawurlencode($key).'="'.rawurlencode($value).'", ';
      }
```

```
  $r = substr($r, 0, -2);
  return $r;
}

function compute($method, $url, $input, $auth, $key_client = '', $key_token = '')
{
  function hmac_sha512($key, $data, $hex_output = false)
  {
    return hash_hmac('sha512', $data, $key, !$hex_output);
  }

  //Check if $url contains a query string
  $pos = strpos($url, '?');
  if ($pos !== false)
  {
    //Split $url into the get query and the preceeding part
    $get = array_from_string(substr($url, $pos+1));
    $url = substr($url, 0, $pos);
  }
  else
  {
    $get = array();
  }
  $post = array_from_string($input);

  $params = array_to_string(array_sort(array_merge($get, $auth, $post)));
  $str = rawurlencode($method).'&'.rawurlencode($url).'&'.rawurlencode($params);
  $key = rawurlencode($key_client).'&'.rawurlencode($key_token);
  return base64_encode(hmac_sha512($key, $str));
}

function authorization($method, $key, $secret, $url, $input)
{
  if (!function_exists('random_bytes'))
  {
    function random_bytes($amount)
    {
      $r = false;
      if (function_exists('openssl_random_pseudo_bytes'))
      {
        $r = openssl_random_pseudo_bytes($amount);
      }
      if ($r === false)
      {
        $r = '';
        while ($amount--)
```

```
          {
            //This is really weak
            $r .= chr(mt_rand(0, 255));
          }
        }
        return $r;
      }
    }

    $time = gettimeofday();
    $auth = array(); //Variables for authorization
    $auth['oauth_signature_method'] = 'HMAC-SHA512';
    $auth['oauth_consumer_key'] = $key;
    $auth['oauth_timestamp'] = $time['sec'];
    $auth['oauth_nonce'] = base64_encode(pack('H*', sprintf('%05x', $time['usec'])) . random_bytes(21));
    $auth['oauth_signature'] = compute($method, $url, $input, $auth, $secret);
    return 'Authorization: OAuth ' . array_to_string_for_authorization($auth);
  }


  //Turn on debugging (if you need it), comment out if you don't require it
  curl_setopt($curl, CURLOPT_VERBOSE, true);

  curl_setopt($curl, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_1);

  curl_setopt($curl, CURLOPT_SSL_VERIFYPEER, true);
  curl_setopt($curl, CURLOPT_SSL_VERIFYHOST, 2); //Maximum verification
  //Force TLS 1.2 because MEETS SaaS supports it
  curl_setopt($curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2);
  //Force strong hash algorithms because MEETS SaaS support them (string below is for OpenSSL / LibreSSL only)
  curl_setopt($curl, CURLOPT_SSL_CIPHER_LIST, '-ALL:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256');

  $method = 'GET';
  if ($input !== null)
  {
    $method = 'POST';
    curl_setopt($curl, CURLOPT_POST, true);
    curl_setopt($curl, CURLOPT_POSTFIELDS, $input);
  }

  curl_setopt($curl, CURLOPT_HTTPHEADER, array(authorization($method, $key, $secret, $url, $input)));

  curl_setopt($curl, CURLOPT_URL, $url);

  curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);
```

```
    $response['data'] = curl_exec($curl);
    if ($response['data'] !== false)
    {
      $response['code'] = curl_getinfo($curl, CURLINFO_HTTP_CODE);
    }
    else
    {
      $response['error'] = curl_error($curl);
    }
    curl_close($curl);
  }
  return (object)$response;
}
```

Java

Here's a full implementation of RFC 5849 Signature authentication in Java which uses *HttpsURLConnection*. *This example is TLS hardened, supports SHA256 and SHA512, works with both GET and POST where the latter is a* query string. *You can tweak this further if you need additional features.*

Note: With the interface below, sending a query string as a string without specifically specifying the *Content-Type* is *application/x-www-form-urlencoded* is an error. Use the Map version of Post to send key-pair parameters instead of raw query strings.

```
import java.io.*;
import java.net.*;
import java.security.*;
import java.util.*;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import javax.net.ssl.HttpsURLConnection;

public class HttpsRfc5849Client
{
  private Mac mac;
  private String key, secret, method;
  private int lastResponseCode;
  private HashMap<String, String> lastResponseHeaders;
  private String lastResponse;

  public HttpsRfc5849Client(String suppliedKey, String suppliedSecret, String signatureMethod) throws NoSuchAlgorithmException
  {
    switch (signatureMethod)
    {
      case "HMAC-SHA1": this.mac = Mac.getInstance("HmacSHA1"); break;
```

```
      case "HMAC-SHA256": this.mac = Mac.getInstance("HmacSHA256"); break;
      case "HMAC-SHA512": this.mac = Mac.getInstance("HmacSHA512"); break;
      default: throw new NoSuchAlgorithmException();
    }

    this.key = suppliedKey;
    this.secret = suppliedSecret;
    this.method = signatureMethod;
    this.lastResponseCode = 0;
  }

  public HttpsRfc5849Client(String suppliedKey, String suppliedSecret) throws NoSuchAlgorithmException
  {
    this.mac = Mac.getInstance("HmacSHA512");
    this.key = suppliedKey;
    this.secret = suppliedSecret;
    this.method = "HMAC-SHA512";
    this.lastResponseCode = 0;
  }


  public void Get(String urlString, Map<String, String> requestHeaders) throws IOException, MalformedURLException
  {
    this.Request("GET", urlString, "", requestHeaders);
  }

  public void Get(String urlString) throws IOException, MalformedURLException
  {
    this.Request("GET", urlString, "", new HashMap<String, String>());
  }

  public void Post(String url, String body, Map<String, String> requestHeaders) throws MalformedURLException, IOException
  {
    this.Request("POST", url, body, requestHeaders);
  }

  //Don't use this function for application/x-www-form-urlencoded, use either of the two below
  public void Post(String url, String body) throws MalformedURLException, IOException
  {
    Map<String, String> requestHeaders = new HashMap<String, String>();
    this.Post(url, body, requestHeaders);
  }

  public void Post(String url, Map<String, String> parameters, Map<String, String> requestHeaders) throws MalformedURLException,
IOException
  {
```

```
  StringBuilder sb = new StringBuilder();
  for (Map.Entry<String, String> entry : parameters.entrySet())
  {
    if (sb.length() != 0)
    {
      sb.append("&");
    }
    sb.append(PercentEncode(entry.getKey()));
    sb.append("=");
    sb.append(PercentEncode(entry.getValue()));
  }
  requestHeaders.put("Content-Type", "application/x-www-form-urlencoded");
  this.Request("POST", url, sb.toString(), requestHeaders);
}

public void Post(String url, Map<String, String> parameters) throws MalformedURLException, IOException
{
  Map<String, String> requestHeaders = new HashMap<String, String>();
  this.Post(url, parameters, requestHeaders);
}


public int responseCode()
{
  return this.lastResponseCode;
}

public String responseBody()
{
  return this.lastResponse;
}

public String responseHeader(String key)
{
  return this.lastResponseHeaders.get(key);
}

public Map<String, String> responseHeaders()
{
  return lastResponseHeaders;
}


private String ComputeSignature(String httpMethod, String urlString, TreeMap<String, String> parameters, String secret)
{
  StringBuilder encodeBuilder = new StringBuilder();
```

```
    for (Map.Entry<String, String> entry : parameters.entrySet())
    {
      encodeBuilder.append(String.format("%1$s=%2$s&", PercentEncode(entry.getKey()), PercentEncode(entry.getValue())));
    }
    String encodePairs = encodeBuilder.length() > 0 ? encodeBuilder.substring(0, encodeBuilder.length() - 1) : "";

    if (urlString.contains("?"))
    {
      urlString = urlString.substring(0, urlString.indexOf("?"));
    }

    String data = String.format("%1$s&%2$s&%3$s", PercentEncode(httpMethod), PercentEncode(urlString),
PercentEncode(encodePairs));

    String signatureString = "";
    try
    {
      this.mac.init(new SecretKeySpec(String.format("%1$s&", PercentEncode(secret)).getBytes("UTF-8"),
this.mac.getAlgorithm()));
      byte[] signatureBytes;
      signatureBytes = this.mac.doFinal(data.getBytes("UTF-8"));
      signatureString = Base64.getEncoder().encodeToString(signatureBytes);
    }
    catch (InvalidKeyException | UnsupportedEncodingException e)
    {
      System.out.println("An error occurred while generating the signature: " + e.getMessage());
      System.exit(0);
    }

    return signatureString;
  }

  private String GenerateNonce(Long milliseconds)
  {
    SecureRandom sr = new SecureRandom();
    byte[] nonce = new byte[24];
    sr.nextBytes(nonce);

    String concat = milliseconds.toString() + nonce.toString();
    String encoded = null;

    try
    {
      encoded = Base64.getEncoder().encodeToString(concat.getBytes("UTF-8"));
    }
    catch (UnsupportedEncodingException e)
```

```
  {
    System.out.println("An error occurred while encoding the nonce. " + e.getMessage());
    System.exit(0);
  }

  return encoded;
}

private void ParseArgs(String tokens, Map<String, String> parameterMap)
{
  String[] keyValuePair = tokens.split("=", 2);
  if (keyValuePair.length == 2)
  {
    if (!keyValuePair[0].isEmpty())
    {
      parameterMap.put(keyValuePair[0], keyValuePair[1]);
    }
  }
  else if (!keyValuePair[0].isEmpty())
  {
    parameterMap.put(keyValuePair[0], "");
  }
}

private void ParseWords(String options, Map<String, String> parameterMap)
{
  while (options.contains("&"))
  {
    ParseArgs(options.substring(0, options.indexOf("&")), parameterMap);
    options = options.substring(options.indexOf("&") + 1, options.length());
  }
  ParseArgs(options, parameterMap);
}

private void ParseURLGetParams(String url, Map<String, String> parameterMap)
{
  if (url.contains("?"))
  {
    ParseWords(url.substring(url.indexOf("?") + 1, url.length()), parameterMap);
  }
}

private String PercentEncode(String word)
{
  String encodedWord = "";
```

```
    try
    {
      encodedWord = URLEncoder.encode(word, "UTF-8");
      encodedWord = encodedWord.replace("+", "%20");
    }
    catch (UnsupportedEncodingException e)
    {
      System.out.println("Encoding error: " + e.getMessage());
      encodedWord = "";
    }

    return encodedWord;
  }

  /**
  * @param method The HTTP method to use: GET, POST, PUT, etc.
  * @param urlString The URL to connect to.
  * @param body Optional, submitted as part of the Request.
  * @param requestHeaders A map of key/value pairs to be sent as part of the Request.
  */
  private void Request(String method, String urlString, String body, Map<String, String> requestHeaders) throws IOException,
MalformedURLException
  {
    this.lastResponseCode = 0;
    this.lastResponseHeaders = new HashMap<String, String>();
    this.lastResponse = new String();

    Calendar currentTime = Calendar.getInstance();
    Long milliseconds = currentTime.getTimeInMillis(); //Milliseconds since the Epoch
    Long seconds = milliseconds / 1000; //Used in the timestamp
    Long remainder = milliseconds % 1000; //Used in the nonce
    String nonce = GenerateNonce(remainder);

    Map<String, String> oauthHeaders = new HashMap<String, String>();
    oauthHeaders.put("oauth_signature_method", this.method);
    oauthHeaders.put("oauth_consumer_key", key);
    oauthHeaders.put("oauth_timestamp", seconds.toString());
    oauthHeaders.put("oauth_nonce", nonce);

    //Building a TreeMap out of the Map guarantees that the keys are sorted, which is a requirement of OAuth.
    TreeMap<String, String> parameters = new TreeMap<String, String>(oauthHeaders);
    ParseURLGetParams(urlString, parameters);
    if (!body.isEmpty())
    {
      String contentType = requestHeaders.get("Content-Type");
      if (contentType != null && contentType.equals("application/x-www-form-urlencoded"))
```

```
      {
        ParseWords(body, parameters);
      }
    }

    oauthHeaders.put("oauth_signature", ComputeSignature(method, urlString, parameters, secret));

    StringBuilder auth_sb = new StringBuilder();
    auth_sb.append("OAuth ");
    for (Map.Entry<String, String> entry : oauthHeaders.entrySet())
    {
      auth_sb.append(String.format("%1$s=\"%2$s\", ", entry.getKey(), PercentEncode(entry.getValue())));
    }
    String authorization_string = auth_sb.substring(0, auth_sb.length() - 2); //Removing excess ", ".

    System.setProperty("https.protocols", "TLSv1.2");
    System.setProperty("https.cipherSuites",
"TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE
_RSA_WITH_AES_128_GCM_SHA256");
    System.setProperty("jdk.certpath.disabledAlgorithms", "MD2, MD5, DSA, RSA keySize< 2048");

    HttpsURLConnection http = (HttpsURLConnection) new URL(urlString).openConnection();
    http.setRequestMethod(method);
    //Writing the authorization header MUST happen before attaching an output stream.
    http.setRequestProperty("Authorization", authorization_string);
    for (Map.Entry<String, String> entry : requestHeaders.entrySet())
    {
      http.setRequestProperty(entry.getKey(), entry.getValue());
    }

    if (!body.isEmpty())
    {
      http.setDoOutput(true);
      http.setRequestProperty("Content-Length", Integer.toString(body.length()));
      DataOutputStream writer = new DataOutputStream(http.getOutputStream());
      writer.write(body.getBytes("UTF-8"));
    }

    http.connect();

    this.lastResponseCode = http.getResponseCode();
    for (int i = 0; ; ++i)
    {
      String key = http.getHeaderFieldKey(i);
      String value = http.getHeaderField(i);
      if (key != null)
```

```
      {
        if (value != null)
        {
          this.lastResponseHeaders.put(key.trim(), value.trim());
        }
        else
        {
          this.lastResponseHeaders.put(key.trim(), "");
        }
      }
      else if (value == null)
      {
        break;
      }
    }

    InputStream is = (this.lastResponseCode >= 0 && this.lastResponseCode< 400) ? http.getInputStream() : http.getErrorStream();
    BufferedReader br = new BufferedReader(new InputStreamReader(is));
    StringBuilder sb = new StringBuilder();
    String output;
    while ((output = br.readLine()) != null)
    {
      sb.append(output);
    }
    br.close();
    this.lastResponse = sb.toString();
  }
}
```

C#

Here's a full implementation of RFC 5849 Signature authentication in C# which uses *HttpWebRequest*. This example supports SHA256 and SHA512, and works with both GET and POST. You can tweak this further if you need additional features.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Net.Security;
using System.Text;
using System.Threading.Tasks;
using System.Security.Cryptography;

namespace RFC5849
```

```
{
  class RFC5849Client
  {
    private String key, secret;
    private HttpStatusCode lastResponseCode;
    private WebHeaderCollection lastResponseHeaders;
    private String lastResponse;

    public void Get(String urlString, Dictionary<String, String> requestHeaders, String hashScheme)
    {
      this.Request("GET", urlString, "", requestHeaders, hashScheme);
    }

    public void Get(String urlString, Dictionary<String, String> requestHeaders)
    {
      this.Request("GET", urlString, "", requestHeaders, "HMAC-SHA512");
    }

    public void Get(String urlString, String hashScheme)
    {
      this.Request("GET", urlString, "", new Dictionary<String, String>(), hashScheme);
    }

    public void Get(String urlString)
    {
      this.Request("GET", urlString, "", new Dictionary<String, String>(), "HMAC-SHA512");
    }

    public void Post(String urlString, String body, Dictionary<String, String> requestHeaders, String hashScheme)
    {
      this.Request("POST", urlString, body, requestHeaders, hashScheme);
    }

    public void Post(String urlString, String body, String hashScheme)
    {
      this.Request("POST", urlString, body, new Dictionary<String, String>(), hashScheme);
    }

    public void Post(String urlString, String body, Dictionary<String, String> requestHeaders)
    {
      this.Request("POST", urlString, body, requestHeaders, "HMAC-SHA512");
    }

    public void Post(String urlString, String body)
    {
      this.Request("POST", urlString, body, new Dictionary<String, String>(), "HMAC-SHA512");
```

```
    }

    public RFC5849Client(String suppliedKey, String suppliedSecret)
    {
      this.key = suppliedKey;
      this.secret = suppliedSecret;
      this.lastResponseCode = 0;
    }

    public HttpStatusCode responseCode()
    {
      return this.lastResponseCode;
    }

    public String responseBody()
    {
      return this.lastResponse;
    }

    public String responseHeader(String key)
    {
      return this.lastResponseHeaders[key];
    }

    public WebHeaderCollection responseHeaders()
    {
      return this.lastResponseHeaders;
    }

    String ComputeSignature(String httpMethod, String urlString, Dictionary<String, String> oauth_headers, Dictionary<String,
String> parameters, String secret, String hashScheme)
    {
      SortedDictionary<String, String> options = new SortedDictionary<String, String>();
      oauth_headers.ToList().ForEach(x => options.Add(x.Key, x.Value));
      parameters.ToList().ForEach(x => options.Add(x.Key, x.Value));

      StringBuilder encodeBuilder = new StringBuilder();
      foreach (KeyValuePair<String, String> kvPair in options)
      {
        encodeBuilder.AppendFormat("{0}={1}&", Uri.EscapeDataString(kvPair.Key), Uri.EscapeDataString(kvPair.Value));
      }
      String encode_pairs = encodeBuilder.ToString().Substring(0, encodeBuilder.Length - 1);  // Remove trailing &.

      if (urlString.Contains("?"))
      {
        urlString = urlString.Substring(0, urlString.IndexOf("?"));
```

```
      }
      byte[] data = Encoding.UTF8.GetBytes(String.Format("{0}&{1}&{2}", Uri.EscapeDataString(httpMethod),
Uri.EscapeDataString(urlString), Uri.EscapeDataString(encode_pairs)));
      byte[] secretBytes = Encoding.UTF8.GetBytes(Uri.EscapeDataString(secret) + "&");
      byte[] outputBytes;

      switch(hashScheme)
      {
        case "HMAC-SHA1":
          HMACSHA1 sha1 = new HMACSHA1(secretBytes);
          outputBytes = sha1.ComputeHash(data);
          sha1.Dispose();
          break;
        case "HMAC-SHA256":
          HMACSHA256 sha256 = new HMACSHA256(secretBytes);
          outputBytes = sha256.ComputeHash(data);
          sha256.Dispose();
          break;
        case "HMAC-SHA512":
          HMACSHA512 sha512 = new HMACSHA512(secretBytes);
          outputBytes = sha512.ComputeHash(data);
          sha512.Dispose();
          break;
        default:
          throw new System.Security.Cryptography.CryptographicException("An invalid hashing scheme was selected; valid values
are HMAC-SHA1, HMAC-SHA256, and HMAC-SHA512.");
      }

      return Convert.ToBase64String(outputBytes);
    }

    private void Request(String httpMethod, String urlString, String body, Dictionary<String, String> requestHeaders, String
hashScheme)
    {
      this.lastResponseCode = 0;
      this.lastResponse = "";

      Dictionary<String, String> oauth_headers = new Dictionary<String, String>();
      Dictionary<String, String> parameters = new Dictionary<String, String>();

      System.Net.ServicePointManager.SecurityProtocol = System.Net.SecurityProtocolType.Tls12;

      oauth_headers.Add("oauth_signature_method", hashScheme);
      oauth_headers.Add("oauth_consumer_key", key);

      DateTime now = DateTime.UtcNow;
```

```
      DateTime epoch = new DateTime(1970, 01, 01, 0, 0, 0, DateTimeKind.Utc);
      long ticks = now.Ticks - epoch.Ticks;
      const long tenmillion = 10000000; // 10,000,000 ticks in a second
      long seconds = ticks / tenmillion;
      long remainder = ticks % tenmillion;

      oauth_headers.Add("oauth_timestamp", seconds.ToString());
      String nonce = GenerateNonce(remainder);
      oauth_headers.Add("oauth_nonce", nonce);

      ParseURLGetParams(urlString, parameters);

      if (!String.IsNullOrEmpty(body))
      {
        ParseWords(body, parameters);
      }

      oauth_headers.Add("oauth_signature", ComputeSignature(httpMethod, urlString, oauth_headers, parameters, secret,
hashScheme));

      StringBuilder auth_sb = new StringBuilder();
      auth_sb.Append("OAuth ");
      foreach (String keyName in oauth_headers.Keys)
      {
        auth_sb.AppendFormat("{0}=\"{1}\", ", keyName, Uri.EscapeDataString(oauth_headers[keyName]));
      }
      String authorization_string = auth_sb.ToString().Substring(0, auth_sb.Length - 2); // Trim off the tailing ", ".

      HttpWebRequest webRequest =(HttpWebRequest)WebRequest.CreateHttp(urlString);
      webRequest.Headers.Add("Authorization", authorization_string);
      parameters.ToList().ForEach(x => webRequest.Headers.Add(x.Key, x.Value));

      if (!String.IsNullOrEmpty(body))
      {
        byte[] postData = Encoding.ASCII.GetBytes(body);

        webRequest.Method = "POST";
        webRequest.ContentType = "application/x-www-form-urlencoded";
        webRequest.ContentLength = postData.Length;

        Stream postStream = webRequest.GetRequestStream();
        postStream.Write(postData, 0, postData.Length);
        postStream.Close();
      }

      HttpWebResponse response;
```

```
  try
  {
    response = (HttpWebResponse)webRequest.GetResponse();
  }
 /* C#'s HttpWebRequest class likes throwing exceptions on 4xx(and 5xx?) responses.
  * So, instead of simply throwing, we can cast the WebException's response field as an HttpWebResponse,
  * and if it is non-null, we can use its information.
  */
  catch(WebException ex)
  {
    response = ex.Response as HttpWebResponse;
    if (response == null)
    {
      throw;
    }
  }

  this.lastResponseCode = response.StatusCode;
  this.lastResponseHeaders = response.Headers;

  StreamReader sr = new StreamReader(response.GetResponseStream());
  this.lastResponse = sr.ReadToEnd();
  sr.Close();
}

String GenerateNonce(long remainder)
{
  RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
  byte[] rngBytes = new byte[24];
  rng.GetBytes(rngBytes);

  return Convert.ToBase64String(Encoding.ASCII.GetBytes(remainder.ToString() + rngBytes.ToString()));
}

void ParseArgs(String tokens, Dictionary<String, String> output)
{
  String[] keyValuePair = tokens.Split('=');
  if (keyValuePair.Length == 2)
  {
    if (!String.IsNullOrEmpty(keyValuePair[0]))
    {
      output.Add(keyValuePair[0], keyValuePair[1]);
    }
  }
  else if (!String.IsNullOrEmpty(keyValuePair[0]))
  {
```

```
        output.Add(keyValuePair[0], "");
      }
    }

    void ParseURLGetParams(String urlString, Dictionary<String, String> output)
    {
      if (urlString.Contains("?"))
      {
        ParseWords(urlString.Substring(urlString.IndexOf("?")), output);
      }
    }

    void ParseWords(String options, Dictionary<String, String> output)
    {
      while(options.Contains("&"))
      {
        ParseArgs(options.Substring(0, options.IndexOf("&")), output);
        options = options.Substring(options.IndexOf("&") + 1);
      }
      ParseArgs(options, output);
    }
  }
}
```

Amazon Web Services Signature Version 4

CirQlive CryptoAuth