# How to use LTI services

Laszlo Csirmaz
Central European University, Budapest

December 10, 2014

## 1 Introduction

Starting from the end of 2015, Turnitin plans to terminate support for the legacy API interface, and offers its services as an LTI Tool Provider. In this note we explain the basic model behind using those services (Section 2), discuss the connection between the old and new terminology (Section 3), and give examples for some of the new calls (Section 5 and 6).

Material in this note was composed exclusively from publicly available information. The most important source was the Moodle integration module by John McGettrick [3]. The tutorial [6] explains what parameters a Tool Provider should expect.

As always, there is no guarantee for the correctness of any information or statement in this note.

## 2 The Learning Management System model

In its simplest form, the LMS model has three components, see [2]. The first one is the *Tool Costumer*, which is represented here by $C$. These consumers are actually computer systems; our example is a Learning Management System (LMS), more specifically, a Moodle server. The second component, the *Tool Provider* $T$ (Turnitin) offers services which are either used directly by the Tool Consumer $C$, or are forwarded to its *users*, represented here by $U$ (see Figure 1). A typical user $U$ is a real person using a web browser on a desktop computer or on a hand-held device. We use $U$ to note both the browser and the person behind the interface.

The communication between the tool consumer $C$ and the service provider $T$ uses the SOAP protocol [4]. Requests and responses are well-formed XML messages defined in several WSDL files [5]. The user $U$ is a web browser communicating with $C$ using the standard HTTP protocol. $U$ can connect directly to $T$ using LTI calls over another HTTP or (preferably) HTTPS connection. Such connections are initiated by snippets embedded into the HTML code sent by $C$ to $U$. According to the IMS recommendation, these LTI calls are `POST` requests.

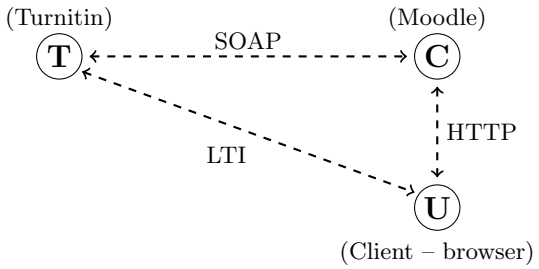


Figure 1: Components and information flow

**Remark**

The only way the client web browser can forward a `POST` request is using a `<form>` tag in the HTML code. To complete the forwarding, either a user intervention is necessary ("push the submit button"), or running JavaScript should be permitted in the client browser. `GET` requests, however, can be redirected transparently, making a conceptually cleaner (and simpler) solution.

Interestingly, Turnitin's response to such `POST` LTI calls are redirects. Rather than letting $U$ send the LTI request, $C$ can send it on behalf of $U$, capture the redirect, and send the redirect command to $U$, which can then get the requested resource from $T$ without any extra user intervention or running any JavaScript.

# 3 Turnitin and LMS terminology

The standard for Learning Management System (LMS) created its own terminology. This table contains commonly used concepts roughly corresponding to LMS terms.

| Turnitin | LMS |
|---|---|
| course | context |
| class | courseSection |
| assignment | lineItem |
| submission | result |
| student | Learner |
| teacher | Instructor |
| primary id | consumer key |
| shared key | shared secret |

# 4 Security, authorization, authenticity

Security is an intricate issue. Serious flaws are frequently discovered in systems used by millions of users after several years of wide deployment. To avoid such problems, security measures should be designed carefully, and with expertise.

Unfortunately, it seems that in the LMS standard security did not get the necessary attention; rather, solutions routinely used by other companies were accepted seemingly without proper evaluation.

Traditionally, Tool Providers talk to Tool Consumers using secure connections (HTTPS), and both parties are interested in keeping the joint secret, which is used to authenticate requests and responses, under tight control. Time stamps and randomly set nonces prevent replay attacks, where the malicious party attempts to re-send a previous request or response. Hashing messages with the join secret achieves *integrity*, which means that messages cannot be altered "en route" by anyone controlling the transmission line. However, ensuring integrity is unnecessary if the connection is already secure, as the secure channel automatically guarantees this property. Rather, the sole role of the joint secret is to *authenticate* the message, ensuring for both parties that the messages are coming from someone who knows the secret.

Using an authentication method which does not require transmitting the join secret (even over a secure channel) is preferred, as it could thwart attacks against the channel (and not, in particular, against the security of the message exchange). Security mechanisms used in the legacy Turnitin API achieved the necessary security requirements in this way.

The situation is entirely different when LTI services are used. The Tool Consumer $C$ *delegates* its authenticity to a third, typically untrusted party, $U$, who, using the delegated credentials, approaches the Tool Provider $T$ to get the granted services. The *digital ticket* which contains these credentials, should satisfy different, and more stringent security measures. Some of the main differences are:

- The client $U$ is not interested in keeping the credentials secret.
- The client $U$ cannot be assumed to be honest, as the client might try to extract as much information from the ticket as possible.
- While with timestamps and nonces, it can be ensured that the ticket can be used only once, and for a limited time only, there is no time limit on *breaking the security of the ticket.*

Authenticity of the delegated tickets are ensured by the OAuth 1.0 authentication mechanism [1], which digitally signs the text of the ticket using the common secret shared by $C$ and $T$ as the key. The ticket cannot be assumed to remain within a controllable environment, thus mounting a dictionary attack *to find the signing key* is a very probable scenario. This attack would be able to find the *common secret*, and thus would let the attacker use all sources that $C$ is entitled to use.

### A possible scenario for exploiting the flaw

In a LMS, a student has to upload her paper using a delegated "submission form". The student then may capture the packet (simply by using the "View

document source" feature, or using web developer tools to investigate the document structure). Using the data there, she can mount an attack to find out the shared secret. She has unlimied time to succeed, and once she recovered the secret key, she could pretend to be her teacher, give and modify grades, look at materials prepared but not to be shown to students, upload late submissions, etc.

#### Countermeasures to prevent the attack

There are several possible countermeasures to prevent this kind of attack. One possibility is to use *temporary keys* to sign tickets; temporary keys are negotiated by $C$ and $T$ and have a tight expiration date. Another possibility is to make sure that the common secret has high entropy, thus it cannot be guessed easily.

To have a temporary remedy until a satisfactory solution is reached, **it would be advisable** if the common secret was generated by the Tool Provider (and not the Tool Consumer), and was a long (at least 40 hexadecimal digits) random number.

## 5 Sample LTI calls

All LTI calls go to the URI starting with `https://HOSTNAME/api/lti/1p0`; the ending is defined below. The requests are composed by the Tool Consumer $C$, and are intended for submission to the Tool Provider $P$ by the web browser $U$. The LTI calls are authenticated using a method inherited from the SOAP authentication, and should use the POST method.

The following parameters together with the request-specific parameters should be set and sent with the request:

| | |
|---|---|
| `lti_message_type` | should be `basic-lti-launch-request`, but seems to be ignored by Turnitin |
| `lti_version` | the version number; this is "`LTI-1p0`" |
| `lang` | for example, "`en_us`" |
| `resource_link_id` | a random hex number identifying the request, see below; |
| `custom_source` | the Turnitin integration code, presently this is `12`. |

The `resource_link_id` is a 8+4+4+4+12 hexadecimal digit-long random value; the parts are separated by hyphens (`-`). In the Moodle integration, five bits of this 128-bit random string are fixed: The first digit of the third number is 4, and the first digit of the fourth number is 8 or bigger.

The following parameters, while defined in the LTI standard, seem to be ignored:

| | |
|---|---|
| `launch_presentation_return_url` | the full URL where the control returns when the service is over |
| `launch_presentation_css_url` | the full URL of a CSS resource to be included. |

Parameters of the OAuth authentication:

| | |
|---|---|
| `oauth_version` | it should be `1.0` |
| `oauth_nonce` | 40 hexadecimal digits; a unique number |
| `oauth_timestamp` | the standard unix time in seconds |
| `oauth_consumer_key` | the integration number supplied by Turnitin |
| `oauth_signature_method` | it should be "`HMAC-SHA1`" |
| `oauth_signature` | the URL-encoded signature of all arguments of the request. |

### Remark on the number of random bits

One can argue that there is too much "randomness" involved in the message. It is encoded in the `oauth_nonce` and in the `resource_link_id` fields. Their lengths are 160 and 128 bits, respectively. For correct functioning, only the *uniqueness* of these values is necessary over a reasonable period of time (say, one day). Using only 32 random bits, both fields could be padded with fixed values without loosing any security.

### Sample Turnitin LTI calls

1. *Show a submission*

   There are four calls in this group: the first one opens the submission page, where there are three tabs: report, peermark, grademark. The other three calls automatically open a specific tab.

   Endpoints: `/dv/default`, `/dv/report`, `/dv/peermark`, `/dv/grademark`.

   Arguments:

   | | |
   |---|---|
   | `lis_person_sourcedid` | the viewer |
   | `roles` | either Instructor or Learner |
   | `lis_result_sourcedid` | the submission ID |

2. *Messages, rubric manager, quickmark*

   View messages, use rubric or quickmark

   Endpoints: `/user/messages`, `/user/rubric`, `/user/quickmark`.

   Arguments:

   | | |
   |---|---|
   | `lis_person_sourcedid` | the user's ID |
   | `roles` | either Instructor or Learner |

3. *Upload a document for similarity checking*

   A document can be uploaded as a new request, or as a replacement for an earlier submission. In the latter case it is accepted only if the assignment is configured to allow resubmissions. When "`custom_xmlresponse`" is set to 1, an XML response is sent back containing the assignment ID, and an extract of the submitted document. The *content* of the uploaded document goes under the parameter name "`custom_submission_data`." When

generating the OAuth signature, this parameter *should not be included* in the computation. This means that the content of the uploaded data is not authenticated; and the signature allows whoever has access to the ticket to upload any content for a given student $X$, under the title $Y$, and assignment $Z$.

Endpoints: `/upload/submit`, `/upload/resubmit`.

Arguments:

| | |
|---|---|
| `lis_person_sourcedid` | the submitter's ID |
| `roles` | either the "`Instructor`" or the "`Learner`" |
| `custom_submission_title` | the title of the paper, maximum 100 characters |
| `custom_submission_author` | the paper's author. The author *must* be enrolled in the class which contains this submission |
| `custom_xmlresponse` | either 0 or 1, according to whether an XML response is requested |
| `lis_lineitem_sourcedid` | the ID of the assignment, only when submitting a new request |
| `lis_result_sourcedid` | the ID of the previous submission to be replaced, only when resubmitting |
| `custom_submission_data` | the document to be submitted for similarity checking. |

When `custom_xmlresponse=1` is used, the returned XML response contains the submission ID under `lis_result_sourcedid`.

4. *End User License Agreement* (EULA)

   The user can accept the Turnitin's EULA on this form.

   Endpoint: `/user/eula`.

   Argument:

   | | |
   |---|---|
   | `lis_person_sourcedid` | the ID of the person who will accept the EULA. |

# 6 SOAP calls

SOAP calls are used for direct communication between the Tool Provider and Tool Consumer. The messages are well-formed XML sent and received via HTTPS with appropriately set headers. The XML for requests and responses is defined by WSDL resources defined in [5]. In the examples below only the relevant arguments are indicated. Requests are sent to `https://HOSTNAME/api/soap/1p0/lis-<service>`, where `<service>` is one of `result`, `person`, `membership`, `coursesection`, and `lineitem`. The XML reply should be parsed; the head contains information about success or failure, and the body contains the requested data.

The header of the HTTPS `POST` request should contain the following:

`Content-type: text/xml; charset="utf-8"`

`Accept: text/xml`

`SOAPAction: "http://<soap action>"`
the URL of the soap action

`Content-length:` the length of the request data

`Source: 12` Turnitin integration code

`Authorization: OAuth <authorization data>`

`Cache-Control: no-cache`

As an example, the full xml request for retrieving a person's full record with record number `22222222` is like the one shown here.

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:ns1="http://www.imsglobal.org/services/lis/
                                    pms2p0/wsdl11/sync/imspms_v2p0">
 <SOAP-ENV:Header><ns1:imsx_syncRequestHeaderInfo>
    <ns1:imsx_version>V1.0</ns1:imsx_version>
    <ns1:imsx_messageIdentifier>97d5792c-13c4-4641-8f7a-
                            ea6d43279a02</ns1:imsx_messageIdentifier>
 </ns1:imsx_syncRequestHeaderInfo></SOAP-ENV:Header>
 <SOAP-ENV:Body><ns1:readPersonRequest>
    <ns1:sourcedId>22222222</ns1:sourcedId>
 </ns1:readPersonRequest></SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

### Sample SOAP calls

The description below indicates what data is required by the function, and it is not indicated under which XML node should it be inserted in the complete XML request. This task can be done easily by inspecting the official WSDL specification [5].

1. *Create a record for a person*

   A new record for a person is created, if it does not exists. Supplied data:

   | | |
   |---|---|
   | `person` | the service to be called |
   | `createByProxyPerson` | the function |
   | `email` | the primary e-mail address; this is used to uniquely identify the person |
   | `firstname` | first and middle names |
   | `lastname` | last name |
   | `role` | Instructor, Learner, Admin, etc. |

   The returned record is either empty, or, if the record has been created successfully, it contains the record number (`sourcedId`).

2. *Get a full record of a person*

The returned XML record contains all data stored under a specific person's record number.

| | |
|---|---|
| `person` | the service to be called |
| `readPerson` | the function |
| `personid` | argument, the record number, the `sourcedId`. |

3. *Find persons based on e-mail address*

   This call returns all personal record IDs where the primary e-mail address matches the given e-mail address.

| | |
|---|---|
| `person` | the service to be called |
| `discoverPersonIds` | the function |
| `email` | the email address to search for |

   The search is successful even if there is no match.

4. *Create a class*

   Classes are called "course section"s in LMS. Only the course title is necessary to create the class.

| | |
|---|---|
| `coursesection` | the service to be called |
| `createByProxyCourseSection` | the function |
| `title` | the title of the course |

   The returned record contains the ID of the new course.

5. *Enroll a student in a class*

   Only enrolled students are allowed to submit papers; thus students should be enrolled into classes.

| | |
|---|---|
| `membership` | the service to be called |
| `createByProxyMemembership` | the function |
| `classid` | the ID of the class |
| `personid` | the ID of the person to be enrolled |
| `role` | either Instructor or Learner, the person's role |

   The returned record contains the ID of a *membership* record which connects the class and the person. One can search membership records containing a certain person to get all things the person belongs to.

6. *Create an assignment*

   Assignments are dubbed as "line items". Every assignment has a label, a corresponding class, and three dates: start, due, and feedback. Several custom parameters should be set as well, such as whether late submission is allowed, whether every submission is final or resubmission is allowed, what is the scale of grading, etc. Only a sample subset of possibilities are shown here.

| | |
|---|---|
| `lineitem` | the service to be called |

| | |
|---|---|
| `createByProxyLineItem` | the function |
| `classid` | the ID of the class this assignment belongs to |
| `title` | the title of the assignment |
| `startdate` | when the assignment starts accepting submissions |
| `duedate` | closing date |
| `feedbacdate` | when the grades are revealed to the students |
| `LateSubmissionAllowed` | yes or no |
| `MaxGrade` | the value of the maximum grade for this assignment |
| `AnonymousMarking` | yes or no, whether the name of submitter should be shown or not, etc. |

The returned record contains the ID of the newly created lineItem (i.e., the assignment).

7. *Obtain the score of a submitted paper*

   After uploading a paper to an assignment, a "result ID" is returned. Also, using a search ("discover") function, the ID's of all submissions in an assignment can be retrieved.

| | |
|---|---|
| `result` | the service to be called |
| `readResult` | the function |
| `paperid` | the ID of the submission – typically a result ID. |

   The XML record returned contains the total score and its breakdown to sub-scores depending on sources, the title, the author's ID, etc.

# References


[1] The OAuth 1.0 Protocol (RFC 5849),
`http://tools.ietf.org/html/rfc5849`

[2] John Tibbetts, *lti2 introduction*,
`http://developers.imsglobal.org/tutorials.html`

[3] John McGettrick, Turnitin Moodle Direct version 2,
`https://github.com/jmcgettrick/MoodleDirectV2`

[4] *SOAP Specifications* – World Wide Web Consortium,
`http://www.w3.org/TR/`

[5] IMS Global Learning Consortium, *Learning Information Services*,
`http://www.imsglobal.org/lis/`

[6] `https://www.edu-apps.org/code.html`