Usage of SAML in Web Authentication

JOHAN BERG
jberg8@kth.se

April 28, 2022

Abstract

This report briefly explains how Security Assertion Markup Language (SAML) 2.0 works, and shows a Python module that simulates the browser's behavior in the KTH Canvas sign-in process, allowing users to fetch login-restricted web pages from the platform. These pages can then be parsed to extract the desired data. A summary of the SAML messages handled by the script is also showcased.

Contents

List of Acronyms and Abbreviations

API application programming interface

CSRF cross-site request forgery

IdP Identity Provider

LMS Learning Management System

SAML Security Assertion Markup Language

SP Service Provider

SSO single sign-on

1 Introduction

This section introduces the problem statement with the background of what led to choosing this topic, provides a theoretical framework for the reader, and the research question that this report strives to answer.

1.1 Problem statement

The Canvas Learning Management System (LMS) [1], from now on ”Canvas”, is a platform used by KTH Royal Institute of Technology, providing teachers and students with learning services such as course pages, assignments, and quizzes. The platform is usually accessed through its web interface, using a web browser. In order to authenticate users in the web interface, the KTH Canvas instance is configured to make use of KTH’s single sign-on (SSO) service, which uses SAML 2.0, from now on ”SAML”, in the authentication requests and responses. Canvas also provides an application programming interface (API) [2], where a user with an API Access Token retrieved from the website can programmatically access the most essential parts of the Canvas service. Although the endpoints available are well documented and defined, the amount of data accessible through the API is limited. This means that some parts of the Canvas service can only be accessed through the web interface. Now, if a program that uses the API wants to access some of this limited data, how does the developer solve this?

1.2 Theoretical framework

This section explains the relevant SAML concepts to understand this report. A more detailed analysis of the SAML messages handled in this study can be seen in Section 4.

OASIS SAML is a standard that defines an XML-based framework for exchanging security information between on-line services [3]. The technical overview document [3] describes relevant concepts, use cases, and architecture of the SAML V2.0 standard. Additionally, several other documents, that can be found in its references, specify certain aspects in greater detail, such as assertions and protocols [4] and bindings to other protocols [5].

The most important use case, according to [3, sec 3], is Multi-Domain SSO, where a Service Provider (SP) can ask a trusted Identity Provider (IdP) if a user is authenticated, in order to give this user access to the site’s content, allowing the user to reuse their login across several websites. This can also be taken one step further to achieve Identity Federation, also known as *account linking*, where the SP and IdP agree on a pseudonymous user ID to refer to the same user. This is useful when the accounts on each side have different names.

The SAML bindings document [5] defines how SAML messages can be transferred over HTTP, which is highly relevant for authenticating web users. Two ways of doing it are:

- using HTTP Redirects, where the SAML message is set as an encoded query parameter in the destination of the redirect, and
- using HTTP POST requests, where the SAML message is an encoded value in an XHTML form delivered to the user. By submitting that form (usually done automatically by JavaScript in the browser), an HTTP POST request with the SAML message in the body is sent to the destination.

Note that in both of these cases, the two services do not contact each other directly. It is the User Agent, typically a browser, that delivers the messages back and forth.

1.3 Research question

What this report strives to answer is: Can a user authenticate in a SAML sequence programmatically without using a browser?

2 Method

Firstly, the Canvas login process was analyzed in Chrome DevTools, while signing in and out a few times. This revealed the usage of HTTP redirects and the Canvas session cookies needed to access the resources. The Chrome extension *EditThisCookie* [6] was used as a tool for quickly viewing and managing cookies. An apparent requirement for the solution was that it had to support holding state, i.e. cookies, between requests, so that the Canvas session cookie obtained could be used in subsequent requests.

The language chosen for writing the solution was Python. This led to the choice of the *Requests* package [7] for doing HTTP requests due to its ability to keep state between requests when using a `Session` object, making it simulate a browser pretty well. Another relevant feature is that it automatically follows redirects.

The development of the program, shown in Appendix A, started with writing code for requesting the `canvas.kth.se/login` endpoint to get into the chain of redirects. When a `200 OK` response was encountered, the response body was examined. Some of the responses in this chain turned out to contain nothing but a hidden HTML form with some values to POST to the next destination. If a browser with JavaScript received this type of response, the form would submit itself due to the one line of JavaScript in the response. But for the program, this had to be done manually. A function was written that took the names and values of all `<input>` elements and constructed a POST request similar to what a browser would have done. Sending this POST request with the same `Session` as before allowed the chain of redirects to continue.

This process was repeated a few times, needing only slight adjustments for the different forms encountered, with one exception: On the KTH login page, the user's credentials clearly needed to be inserted into the POST request before sending it off. After a quick and simple check that the login seemed valid, the program was pretty much done at this point. The redirect chain had ended up back at Canvas and a session cookie was present.

Verifying the functionality of the program was straight-forward: Simply using the same `Session`, a request was made to fetch a login-protected page. Verifying that the page had been correctly delivered deemed the solution complete. The final work involved refactoring, type-hinting, and documenting the code.

3 Results

The Python module, shown in Appendix A, was integrated with a script from the collection *Canvas-tools* by Maguire [8] in order to fetch results of many quiz submissions. After a successful run, he wrote "I was able to process [...] 12,000 submissions in just a little more than 22 minutes. A great improvement over my manually saving web pages at the rate of 4 per minute" [9].

4 Analysis of SAML messages

For a better insight into how the solution works, the SAML messages from a test run will be shown and analyzed below. Outtakes shown will contain the symbol "`[...]`" to indicate that data has been omitted for brevity.

4.1 Overview

Figure 1 shows the order and content of messages passed when the SP uses HTTP Redirects and the IdP uses XHTML forms. This is the setup that is used by KTH. The *steps* mentioned hereafter refer to the numbered messages in Figure 1.

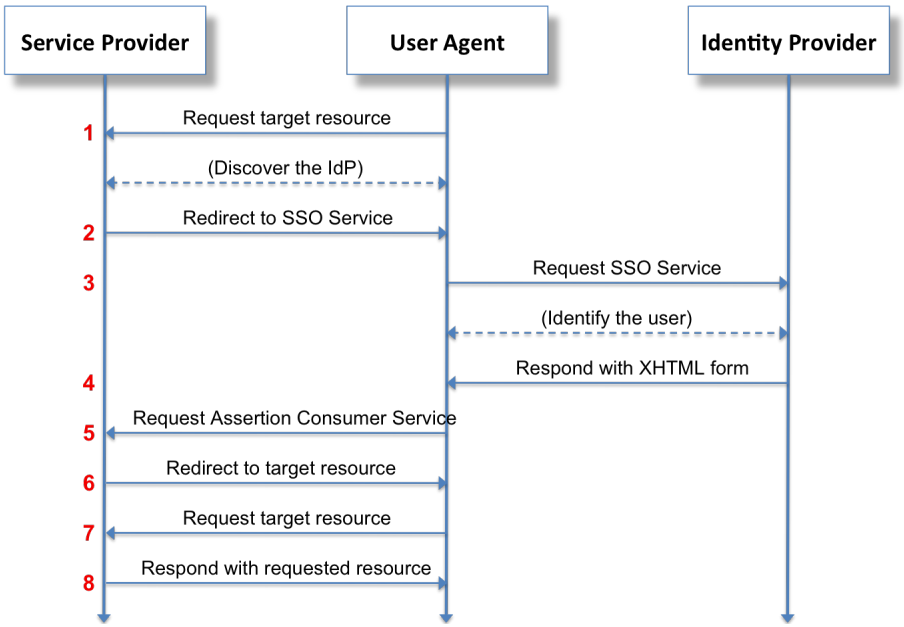

Figure 1: SAML 2.0 Web Browser SSO (SP Redirect Bind/ IdP POST Response) [10], appears here under CC BY-SA 3.0.

4.2 SAML Request

The test run is started in step 1 by requesting `https://canvas.kth.se/login` (perhaps not a login-protected resource, but at least the page that begins the login process). Since no valid security context exists, Canvas initiates a SAML login. In this case, the *Discover the IdP* is skipped, since the IdP is pre-determined.

Now onto step 2 and 3. After first being redirected to `https://canvas.kth.se/login/saml`, we arrive at

```
https://saml-5.sys.kth.se/
    idp/profile/SAML2/Redirect/SSO?SAMLRequest=fZJfT8[...]
```

What happened here is the HTTP Redirect mechanism described in [5, sec 3.4]. Canvas generated a SAML request, used the three encoding steps *Deflate* → *Base64 Encode* → *URL Encode*, and placed it in the `SAMLRequest` URL parameter in the redirect to KTH's SAML SSO endpoint. By using *URL Decode* → *Base64 Decode* → *Inflate*, the SAML message is revealed (some XML attributes omitted):

```
<samlp:AuthnRequest
    ID="_f8e8369a[...]"
    Destination="https://saml-5.sys.kth.se/idp/profile/SAML2/Redirect/SSO"
    AssertionConsumerServiceURL="https://canvas.kth.se/login/saml"
    ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
>
  <saml:Issuer>http://kth.instructure.com/saml2</saml:Issuer>
  <samlp:NameIDPolicy AllowCreate="true"/>
</samlp:AuthnRequest>
```

Among other things, we find out the Issuer of the request, the Assertion Consumer Service for step 5, and that the HTTP POST binding [5, sec 3.5] should be used to send the response.

4.3 CSRF

What happens next is a security mechanism unrelated to SAML, but it will be mentioned anyway. `saml-5.sys.kth.se` responds with a form containing a cross-site request forgery (CSRF) token (see Section 4.5 for an example of its structure). This is a security check that helps ensure the requests we are sending cannot be exploited by an attacker [11]. The form is simply submitted as described in Section 2.

4.4 Nested SAML Request and Response

At this point, the `saml-5.sys.kth.se` server *could* have been responsible for authenticating the user with credentials, but that is not the case here. What happens is that `saml-5.sys.kth.se`, that thus far has acted as IdP, now acts SP and sends a new SAML request to `login.ug.kth.se`, the server responsible for authentication. The request and response of this exchange are very similar to the ones shown in Sections 4.2 and 4.5. Therefore, this ”nested” request and response is now shown here in Section 4.4.

During the trip to the login server, a form for the login credentials is served by the login server. This form is submitted in the same fashion as the other forms, the only difference being that the username and password are inserted into the correct fields.

4.5 SAML Response

After `saml-5.sys.kth.se` has received a valid SAML response from `login.ug.kth.se` (the nested response), the main response can now be sent back to Canvas (steps 4 and 5 in figure 1). This response is given as a form with a value to submit, in accordance with the HTTP POST binding [5, sec 3.5]. The form looks roughly like this:

```
<form method="POST" action="https://canvas.kth.se/login/saml">
  <input type="hidden" name="SAMLResponse" value="PD94bWwg[...]"></input>
  [...]
</form>
```

This `value` field is once again Base64 encoded, but neither Deflated or URL encoded this time. Base64 decoding gives us the full SAML response, shown below. Due to it’s length, many parts have been omitted or truncated here.

```
<saml2p:Response
    Destination="https://canvas.kth.se/login/saml"
    ID="_16cad3d4[...]"
    InResponseTo="_f8e8369a[...]"
>
  <saml2:Issuer>https://saml.sys.kth.se/idp/shibboleth</saml2:Issuer>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
      [...]
    </ds:SignedInfo>
    <ds:SignatureValue>luwQrtzs[...]</ds:SignatureValue>
    <ds:KeyInfo>
      <ds:X509Data>
        <ds:X509Certificate>MIIDLzCC[...]</ds:X509Certificate>
      </ds:X509Data>
    </ds:KeyInfo>
  </ds:Signature>
  <saml2p:Status>
    <saml2p:StatusCode
      Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
  </saml2p:Status>
  <saml2:EncryptedAssertion>
    <xenc:EncryptedData>
      <xenc:EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <xenc:EncryptedKey Recipient="http://kth.instructure.com/saml2">
          [...]
```

```
        </xenc:EncryptedKey>
      </ds:KeyInfo>
      <xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
        <xenc:CipherValue>KlH9mqfX[...]</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </saml2:EncryptedAssertion>
</saml2p:Response>
```

The main points of interest here are:

- The `InResponseTo` and `Destination` attributes of the response match the `ID` and `AssertionConsumerServiceURL` attributes of the request respectively.
- The `StatusCode` tag shows that the authentication was successful.
- The `Issuer` tag suggests that the KTH SAML service uses the software *Shibboleth* [12] for handling SSO with SAML.
- There is encrypted data being sent back to the SP as part of the response. Since the SP does not contact the IdP directly, the message itself has to have an XML Signature in order for the SP to verify the authenticity of the message [3, sec 4.6].

After receiving this response (in a POST request), the Canvas server now sets a session cookie that allows access to login-protected pages. The session cookie is:

```
<Cookie canvas_session=J_RWmKr2[...] for canvas.kth.se/>
```

It is now possible to perform steps 7 and 8 in Figure 1.

5 Discussion

The result shows that a Python implementation for authenticating users through SAML is possible. This report only shows an implementation for KTH Canvas, but it can serve as proof of concept for implementations for other services. The solution is far from ”complete” and can be developed further.

References

[1] “Overview | Canvas,” Apr. 2022, accessed 2022-04-22. [Online]. Available: https://www.instructure.com/canvas

[2] “Canvas LMS REST API Documentation,” Apr. 2022, accessed 2022-04-22. [Online]. Available: https://canvas.instructure.com/doc/api/

[3] N. Ragouzis, J. Hughes, R. Philpott, E. Maler, P. Madsen, and T. Scavo, “Security Assertion Markup Language (SAML) V2.0 Technical Overview,” Mar. 2008, accessed 2022-04-19. [Online]. Available: https://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf

[4] S. Cantor, J. Kemp, R. Philpott, and E. Maler, “Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0,” Mar. 2005, accessed 2022-04-18. [Online]. Available: https://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf

[5] S. Cantor, F. Hirsch, J. Kemp, R. Philpott, and E. Maler, “Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0,” Mar. 2005, accessed 2022-04-20. [Online]. Available: https://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf

[6] "EditThisCookie," Apr. 2022, accessed 2022-04-22. [Online]. Available: https://www.editthiscookie.com/

[7] "Requests: HTTP for Humans™ — Requests 2.27.1 documentation," Apr. 2022, accessed 2022-04-22. [Online]. Available: https://docs.python-requests.org/en/latest/

[8] Gerald Q. Maguire Jr., "Canvas-tools," Apr. 2022, accessed 2022-04-22. [Online]. Available: https://github.com/gqmaguirejr/Canvas-tools

[9] ——, "SAML vs Canvas API," e-mail, Apr. 2022.

[10] Ayungn, "Saml2-browser-sso-redirect-post.png," Oct. 2021. [Online]. Available: https://commons.wikimedia.org/wiki/File:Saml2-browser-sso-redirect-post.png

[11] "What is CSRF (Cross-site request forgery)? Tutorial & Examples | Web Security Academy," Apr. 2022, accessed 2022-04-22. [Online]. Available: https://portswigger.net/web-security/csrf

[12] "Shibboleth Consortium - Shaping the future of Shibboleth Software," Apr. 2022, accessed 2022-04-22. [Online]. Available: https://www.shibboleth.net/

A Source code

This is the file `kth_canvas_login.py`, written for Python 3.8 or higher. It is also available on GitHub:
https://github.com/jonaro00/Canvas-tools/blob/44f2bdf5f0ad79502a5a625ad6dd684c2df23ef3/kth_canvas_saml.py
A few extra line feeds appear below due to document formatting.

```
#!/usr/bin/python3

"""
Toolkit for signing in to KTH's Canvas LMS (through SAML) directly in Python.
This allows you to make scripts that access login-protected pages.
This extends the range of possibilities beyond what the Canvas LMS API offers,
allowing the user to automate even more tasks involving Canvas.

Example usage:
```py
>>> from kth_canvas_saml import kth_canvas_login
>>> s = kth_canvas_login('<user>@ug.kth.se', '<password>')
>>> r =
 ↪  s.get('https://canvas.kth.se/courses/<c_id>/quizzes/<q_id>/history?quiz_submission_id=<s_id>&version=1')
>>> print(r.text)
```

Running this script alone will prompt you for credentials and test the sign in with verbose printouts:
```text
$ ./kth_canvas_saml.py
Testing KTH Canvas login.
KTH username: test
Password for user test@ug.kth.se:
Got response from https://canvas.kth.se/login : code 302
Got response from https://canvas.kth.se/login/saml : code 302
... [rest of output hidden]
```

Remember to handle your password with care!
If you do not wish to have your password stored in plain text,
you may use `kth_canvas_login_prompt`, or make something yourself.

The two dependencies can be installed with: `pip install requests beautifulsoup4`

Author: Johan Berg, 2022-04-09
"""

from urllib.parse import urljoin
from getpass import getpass

# The requests library for keeping state between HTTP requests
from requests import Request, Response, Session, codes # pip install requests

# Beautiful Soup for parsing HTML
from bs4 import BeautifulSoup # pip install beautifulsoup4

```

```python

KTH_CANVAS_LOGIN = 'https://canvas.kth.se/login'
KTH_EMAIL_SUFFIX = '@ug.kth.se'
DEFAULT_TIMEOUT = 10


def kth_canvas_login_prompt(user: str = '', password: str = '', **kwargs) -> Session:
    """Prompt for username and password if not provided, then send them and kwargs to `kth_canvas_login`."""
    user = user or input('KTH username: ')
    if '@' not in user:
        user += KTH_EMAIL_SUFFIX
    password = password or getpass(f'Password for user {user}: ')
    return kth_canvas_login(user, password, **kwargs)


def kth_canvas_login(
        user: str,
        password: str,
        login_url: str = KTH_CANVAS_LOGIN,
        session: Session = None,
        timeout: float = DEFAULT_TIMEOUT,
        verbose: bool = False,
        ) -> Session:
    """
    Goes through the chain of redirects and logs in to Canvas with the provided credentials,
    returning a `Session` with the proper cookies needed to access Canvas pages.

    Parameters:
      `user`: Email address of user.
      `password`: The password.
      `login_url`: A starting point that redirects to the sign in process.
      `session`: If a session is provided, the requests will continue from there, otherwise creates a new one.
      `timeout`: Max number of seconds to wait for each response.
      `verbose`: If True, print more information.

    Returns:
      `Session` with cookies allowing access to Canvas pages.

    Raises:
      `ValueError`: When the login fails.
    """
    if verbose and not user.endswith(KTH_EMAIL_SUFFIX):
        print(f'Warning, username {user} seems to be invalid')

    # Use the provided session or create a new one
    s = session or Session()

    if verbose:
        # Add callback printouts in verbose mode
        s.hooks['response'].append(lambda r, *a, **kw: print(f'Got response from {r.url} : code {r.status_code}'))
        print(f'Authenticating at {login_url}')

    # Try to access Canvas, get redirected to KTH SAML
    r = s.get(login_url, timeout=timeout)
    verify_ok(r)

    # Send in first form, continue to KTH login
    req = html_form_to_request(r.text, r.url, {'name': 'form1', 'method': 'post', 'action': True})
    r = s.send(s.prepare_request(req), timeout=timeout)
    verify_ok(r)

    # Send KTH login credentials
    req = html_form_to_request(r.text, r.url, {'id': 'loginForm', 'method': 'post', 'action': True})
    req.data['UserName'] = user
    req.data['Password'] = password
    r = s.send(s.prepare_request(req), timeout=timeout)
    verify_ok(r)
    # Assumes a correct login causes a 302 redirect, therefore raising the error if we were not redirected here.
    # (A very naive check for login validity)
    if not r.history:
        raise ValueError('Sign in failed. Incorrect credentials?')

    # Send SAML token back to KTH SAML
    req = html_form_to_request(r.text, r.url, {'name': 'hiddenform', 'method': 'POST', 'action': True})
    r = s.send(s.prepare_request(req), timeout=timeout)
    verify_ok(r)

    # Send SAML token back to Canvas
    req = html_form_to_request(r.text, r.url, {'method': 'post', 'action': True})
    r = s.send(s.prepare_request(req), timeout=timeout)
    verify_ok(r)

    if verbose:
        print('Authentication successful')
```

```
131
132     # Session should now contain a valid Canvas session cookie
133     return s
134
135
136 def html_form_to_request(html: str, url: str, attributes: dict = {}, default_input_value: str = 'false')
    ↪  -> Request:
137     """
138     Parses HTML to find a <form> element and constructs a `Request` that
139     contains (roughly) the POST request that a browser would send if the form was submitted.
140     If multiple forms match, the first one is used.
141
142     Parameters:
143       `html`: String of HTML to parse.
144       `url`: URL where the HTML was fetched from. This is to calculate the resulting requests'
    ↪  destination.
145       `attributes`: Dict where keys are names of attributes the form must have, and values are "filters",
146                     see https://beautiful-soup-4.readthedocs.io/en/latest/#kinds-of-filters .
147       `default_input_value`: This value is used when an <input> does not have a 'value' field.
148
149     Returns:
150       A `Request` that can be prepared with or without a `Session`, and then sent.
151
152     Raises:
153       `ValueError`: If no matching form was found.
154     """
155     # Parse HTML
156     soup = BeautifulSoup(html, 'html.parser')
157
158     # Find the first <form> element with the correct attributes.
159     # Example:
160     #   <form method="post" id="loginForm" action="..."> ... </form>
161     form = soup.find('form', attributes)
162     if not form:
163         raise ValueError('No form found in HMTL')
164
165     # Check action field to calculate destination of the POST request
166     destination = urljoin(url, form['action'])
167     # Find all <input> elements with 'name' attributes,
168     # then fill a dict with their pre-filled values.
169     # Example:
170     #   <input id="userNameInput" name="UserName" type="email" value=""/>
171     inputs = form.find_all('input', {'name': True})
172     data = {i['name']: i.get('value', default_input_value) for i in inputs}
173
174     return Request('POST', destination, data=data)
175
176
177 def verify_ok(r: Response) -> None:
178     """Raises `ValueError` if response code is not OK."""
179     if r.status_code != codes.ok:
180         raise ValueError(f'{r.url} responded with code {r.status_code}.')
181
182
183 def test() -> None:
184     """Tests the login process and prints the cookies gathered."""
185     print('Testing KTH Canvas login.')
186     try:
187         s = kth_canvas_login_prompt(verbose=True)
188     except ValueError as e:
189         print(e)
190         return
191     print('Got cookies:')
192     for c in s.cookies:
193         print(c)
194
195
196 if __name__ == '__main__':
197     test()
```