[UIUCTF 2023] Rattler Read

Robin Jadoul Jul 02, 2023 uiuctf, misc, pyjail

> All these poisonous snakes keep biting me. Good think I remembered to bring restrictedpython to keep me safe!
> **Challenge Author:** Pete Stenger

Points: 390 (16 solves)

Challenge overview

We're given a pyjail, but rather than the standard, more or less ad-hoc ways of restricting python execution that we commonly see in CTF challenges, this one is using an existing third-party module from the zope project to constrict us.

```
from RestrictedPython import compile_restricted
from RestrictedPython import Eval
from RestrictedPython import Guards
from RestrictedPython import safe_globals
from RestrictedPython import utility_builtins
from RestrictedPython.PrintCollector import PrintCollector

def exec_poisonous(code):
    """Makes sure your code is safe to run"""

    def no_import(name, *args, **kwargs):
        raise ImportError("Don't ask another snake for help!")
    code += "\nresults = printed"
    byte_code = compile_restricted(
        code,
        filename="<string>",
        mode="exec",
    )
    policy_globals = {**safe_globals, **utility_builtins}
    policy_globals['__builtins__']['__metaclass__'] = type
    policy_globals['__builtins__']['__name__'] = type
    policy_globals['__builtins__']['__import__'] = no_import
    policy_globals['_getattr_'] = Guards.safer_getattr
    policy_globals['_getiter_'] = Eval.default_guarded_getiter
    policy_globals['_getitem_'] = Eval.default_guarded_getitem
    policy_globals['_write_'] = Guards.full_write_guard
    policy_globals['_print_'] = PrintCollector
    policy_globals['_iter_unpack_sequence_'] = Guards.guarded_iter_unpack_sequence
    policy_globals['_unpack_sequence_'] = Guards.guarded_unpack_sequence
    policy_globals['enumerate'] = enumerate
    exec(byte_code, policy_globals, None)
    return policy_globals["results"]

if __name__ == '__main__':
    print("Well, well well. Let's see just how poisonous you are..")
    print(exec_poisonous(input('> ')))
```

git checkout protections --empirically

We could absolutely start off our exploration of the protections offered by RestrictedPython by diving into the source code, and analysing every step in `compile_restricted` and all introduced globals in minute detail, but where's the fun in that? Instead, we can just try to input some assorted pieces of python code and see how the system reacts to it.

One of the important things to notice then, is that attribute access seems to be restricted in an ast transformer. We're unable to access attributes with a name staring with `_`. Looking at the source code just a tiny bit, we also see that we've got some replacement `getattr` function that seems to impose the same restriction, and denies access to the `format` method for string values. We've got some globals and builtins that are considered "safe" by the RestrictedPython authors, and apparently also a few modules that provide utilities (`string`, `math` and `random`).

There's also all sorts of messing about with iteration and indexing going on, but let's focus on the getattr situation first. After all, with a sufficiently unrestricted getattr and a few function calls, we should be able to apply standard pyjail escape techniques and then hopefully we can remain blissfully unaware of all other potential holes in our jail.

git blame Guards.safer_getattr

```
def safer_getattr(object, name, default=None, getattr=getattr):
    """Getattr implementation which prevents using format on string objects.

    format() is considered harmful:
    http://lucumr.pocoo.org/2016/12/29/careful-with-str-format/

    """
    if isinstance(object, str) and name == 'format':
        raise NotImplementedError(
            'Using format() on a %s is not safe.' % object.__class__.__name__)
    if name.startswith('_'):
        raise AttributeError(
            '"{name}" is an invalid attribute name because it '
            'starts with "_"'.format(name=name)
        )
    return getattr(object, name, default)
```

As a first quick observation, and to illustrate once more the dangers of blacklisting, and the versatility of python, let's look at two ways to bypass the first check here:

- Replace `s.format(...)` by `str.format(s, ...)`, that's pretty much what calling a method will do in the first place.
- Ever since python 3.2, there's another method, `str.format_map` that does essentially the same thing, but without doing splatting or turning things into a dictionary.

Unfortunately, to the best of my knowledge, `str.format` can at best lead to information leakage, and not directly to code execution, so we'll have to look a bit further. The underscore check itself doesn't look like it's trivially bypassed, but as we'll explore in the next section, not all juicy and desirable member variables are locked behind underscores. For now, let's have a look at that linked blog post about string formatting however.

> You can use the somewhat undocumented internals to change the behavior.

Undocumented internals, you say, in the `string` module? Diving into python standard library source code is always so much more fun than third-party libraries, so let's go!

```
# in the class string.Formatter
    # given a field_name, find the object it references.
    #  field_name:   the field being looked up, e.g. "0.name"
    #                 or "lookup[3]"
    #  used_args:    a set of which args have been used
    #  args, kwargs: as passed in to vformat
    def get_field(self, field_name, args, kwargs):
        first, rest = _string.formatter_field_name_split(field_name)

        obj = self.get_value(first, args, kwargs)

        # loop through the rest of the field_name, doing
        #  getattr or getitem as needed
        for is_attr, i in rest:
            if is_attr:
                obj = getattr(obj, i)
            else:
                obj = obj[i]

        return obj, first
```

Now this looks interesting, won't you say? We just need to check if that `getattr` isn't restricted in the same way we are. With a bit of fiddling in a python repl, we can check out the value `string.Formatter.get_field.__globals__["__builtins__"]` within the jail (you can e.g. inject the real `getattr` as it's own utility variable into the jail to try this out in the right context) to notice that this is the jackpot we were looking for. We now have an arbitrary getattr, we can still call functions and methods, so we can get our pyjail payloads in. I personally like to go through the `BuiltinImporter` class, which in this case turned out to be at index 84 (just print stuff out and count, to find that).

```
string.Formatter().get_field("a.__class__.__base__.__subclasses__", [], {"a": ""})[0]()[84].load_module("
```

Conveniently enough, this doesn't even need to be in an `exec` context, `eval` is more than enough here.

git ls-tree generator_expression

Now, to come back to our remark about important members not being hidden behind underscores, consider the following:

```
for f in (g := (g.gi_frame.f_back.f_back for _ in [1])): print(f.f_builtins)
```

For those unaware, `g` is something knows as a "generator", created from a "generator expression". Since it is in essence a sort of coroutine, it needs to keep track of its internal state during execution, and the frame in which it is executing can be accessed by the `gi_frame` attribute. From there, we can traverse the python stack up until we're outside of this pesky and protected `exec` context, and kindly ask for the builtins, which again contain a regular and unrestricted version of `getattr`. The one important thing to take into account when doing this is that we need the generator to be running at the time we try to access the stack frames, hence the loop surrounding it. Making a full exploit out of this is left as an (easy, since it's pretty much the same as before) exercise to the reader.

git pull

I just felt the need to end this writeup by noticing that the techniques used to solve this challenge still work just fine on the latest python/RestrictedPython combo. I guess it only goes to reinforce that notion – as it's been so many times by an avalanche of pyjail CTF challenges – that trying to safely evaluate user-provided python code is a *bad idea*. Here be dragons!