Eval really is dangerous

https://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html

None Sat Oct, 31 01:30

Wednesday 6 June 2012

Python has an eval() function which evaluates a string of Python code:

```
assert eval('2 + 3 * len('hello')') == 17
```

This is very powerful, but is also very dangerous if you accept strings to evaluate from untrusted input. Suppose the string being evaluated is "os.system('rm -rf /')" ? It will really start deleting all the files on your computer. (In the examples that follow, I'll use 'clear' instead of 'rm -rf /' to prevent accidental foot-shootings.)

Some have claimed that you can make eval safe by providing it with no globals. eval() takes a second argument which are the global values to use during the evaluation. If you don't provide a globals dictionary, then eval uses the current globals, which is why "os" might be available. If you provide an empty dictionary, then there are no globals. This now raises a NameError, "name 'os' is not defined":

```
eval('os.system('clear')', {})
```

But we can still import modules and use them, with the builtin function __import__. This succeeds:

```
eval('__import__('os').system('clear')', {})
```

The next attempt to make things safe is to refuse access to the builtins. The reason names like __import__ and open are available to you in Python 2 is because they are in the __builtins__ global. We can explicitly specify that there are no builtins by defining that name as an empty dictionary in our globals. Now this raises a NameError:

```
eval('__import__('os').system('clear')', {'__builtins__':{}})
```

Are we safe now? Some say yes, but they are wrong. As a demonstration, running this in CPython will segfault your interpreter:

```
s = '''
(lambda fc=(
    lambda n: [
        c for c in
            ().__class__.__bases__[0].__subclasses__()
            if c.__name__ == n
        ][0]
    ):
    fc('function')(
        fc('code')(
            0,0,0,0,'KABOOM',(),(),(),'','',0,''
        ),{}
    )()
)()
'''
eval(s, {'__builtins__':{}})
```

Let's unpack this beast and see what's going on. At the center we find this:

```
().__class__.__bases__[0]
```

which is a fancy way of saying "object". The first base class of a tuple is "object". Remember, we can't simply say "object", since we have no builtins. But we can create objects with literal syntax, and then use attributes from there.

Once we have object, we can get the list of all the subclasses of object:

```
().__class__.__bases__[0].__subclasses__()
```

or in other words, a list of all the classes that have been instantiated to this point in the program. We'll come back to this at the end. If we shorthand this as ALL_CLASSES, then this is a list comprehension that examines all the classes to find one named n:

```
[c for c in ALL_CLASSES if c.__name__ == n][0]
```

We'll use this to find classes by name, and because we need to use it twice, we'll create a function for it:

```
lambda n: [c for c in ALL_CLASSES if c.__name__ == n][0]
```

But we're in an eval, so we can't use the def statement, or the assignment statement to give this function a name. But default arguments to a function are also a form of assignment, and lambdas can have default arguments. So we put the rest of our code in a lambda function to get the use of the default arguments as an assignment:

```
(lambda fc=(
    lambda n: [
        c for c in ALL_CLASSES if c.__name__ == n
        ][0]
    ):
    # code goes here...
)()
```

Now that we have our "find class" function fc, what will we do with it? We can make a code object! It isn't easy, you need to provide 12 arguments to the constructor, but most can be given simple default values.

```
fc('code')(0,0,0,0,'KABOOM',(),(),(),'','',0,'')
```

The string "KABOOM" is the actual bytecodes to use in the code object, and as you can probably guess, "KABOOM" is not a valid sequence of bytecodes. Actually, any one of these bytecodes would be enough, they are all binary operators that will try to operate on an empty operand stack, which will segfault CPython. "KABOOM" is just more fun, thanks to lvh for it.

This gives us a code object: fc("code") finds the class "code" for us, and then we invoke it with the 12 arguments. You can't invoke a code object directly, but you can create a function with one:

```
fc('function')(CODE_OBJECT, {})
```

And of course, once you have a function, you can call it, which will run the code in its code object. In this case, that will execute our bogus bytecodes, which will segfault the CPython interpreter. Here's the dangerous string again, in more compact form:

```
(lambda fc=(lambda n: [c for c in ().__class__.__bases__[0].__subclasses__()
    if c.__name__ == n][0]): fc('function')(fc('code')(0,0,0,0,'KABOOM',(),
    (),(),'','',0,''),{})())()
```

So eval is not safe, even if you remove all the globals and the builtins!

We used the list of all subclasses of object here to make a code object and a function. You can of course find other classes and use them. Which classes you can find depends on where the eval() call actually is. In a real program, there will be many classes already created by the time the eval() happens, and all of them will be in our list of ALL_CLASSES. As an example:

```
s = '''
[
    c for c in
    ().__class__.__bases__[0].__subclasses__()
    if c.__name__ == 'Quitter'
][0](0)()
'''
```

The standard site module defines a class called Quitter, it's what the name "quit" is bound to, so that you can type quit() at the interactive prompt to exit the interpreter. So in eval we simply find Quitter, instantiate it, and call it. This string cleanly exits the Python interpreter.

Of course, in a real system, there will be all sorts of powerful classes lying around that an eval'ed string could instantiate and invoke. There's no end to the havoc that could be caused.

The problem with all of these attempts to protect eval() is that they are blacklists. They explicitly remove things that could be dangerous. That is a losing battle because if there's just one item left off the list, you can attack the system.

While I was poking around on this topic, I stumbled on Python's restricted evaluation mode, which seems to be an attempt to plug some of these holes. Here we try to access the code object for a lambda, and find we aren't allowed to:

```
>>> eval('(lambda:0).func_code', {'__builtins__':{}})
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
  File '<string>', line 1, in <module>
RuntimeError: function attributes not accessible in restricted mode
```

Restricted mode is an explicit attempt to blacklist certain "dangerous" attribute access. It's specifically triggered when executing code if your builtins are not the official builtins. There's a much more detailed explanation and links to other discussion on this topic on Tav's blog. As we've seen, the existing restricted mode it isn't enough to prevent mischief.

So, can eval be made safe? Hard to say. At this point, my best guess is that you can't do any harm if you can't use any double underscores, so maybe if you exclude any string with double underscores you are safe. Maybe...

Update: from a thread on Reddit about recovering cleared globals, a similar snippet that will get you the original builtins:

```
[
    c for c in ().__class__.__base__.__subclasses__()
    if c.__name__ == 'catch_warnings'
][0]()._module.__builtins__
```