The Case for Python in Web Security Testing

By:

Nathan Hamiel – FishNet Security
Marcin Wielgoszewski – Gotham Digital Science

Black Hat USA 2010

Abstract

It seems that everything is a web application nowadays. Whether the application is cloud-based, mobile, or even fat client they all seem to be using web protocols to communicate. Adding to the traditional landscape there is rise in the use of application programming interfaces, integration hooks, and next generation web technologies. What this means for someone testing web applications is that flexibility is the key to success. The Python programming language is just as flexible as today's web application platforms. The language is appealing to security professionals because it is easy to read and write, has a wide variety of modules, and has plenty of resources for help. This additional flexibility affords the tester greater depth than many of the canned tests that come with common tools they use on a daily basis. Greater familiarity plus flexible language equals tester win!

Write Your Own?

Someone tasked with testing modern web technologies needs to be flexible. There will be times when the standard tool kits are not effective for a testing situation. Testing tools are constantly lagging behind technology and their capabilities can be limited. Rather than throwing up your hands and admitting defeat there are steps you can take to ensure success in your testing activities. This success often comes with writing your own clients and performing your own tests.

With intimate knowledge of the applications that are being tested new doors begin to open that were previously closed. Say for instance that we have a highly dynamic site. Every time a page is laid out in the browser the links associated with menu items have some randomized values associated with them and these values are never the same twice. This can wreak havoc on standard testing tools because tests cannot be replayed, which is how many web testing tools work. Even though testing is difficult from a standard tool perspective, someone writing a client merely needs to be able to read and parse the content.

Why Python

Python is a byte compiled, object-oriented programming language that is easy to read and write. This makes Python a good fit for programs from just a few lines to thousands of lines of code. The language is great for security professionals because it allows for the rapid creation of tests as well as reusable items for future use. Many tools are written in Python. This offers many opportunities for extending and adding features to tools that are already written.

Being that there are many tools are writing in Python it also means that there are plenty of code examples for you to take advantage of. There is a search engine for Python source code called Nullege[1]. This search engine can be used to find examples of how to use modules or other Python code features in real world examples.

Building an HTTP Client

Building an HTTP client in Python couldn't be easier. Built in to the Python standard library are modules such as *httplib*, *urllib*, and *urllib2*. These modules have difference capabilities between all of them, but are useful for a majority of your web testing. There is also the 3rd party httplib2[2] module that provides some enhanced capabilities over the standard *httplib* module.

Parsing Content

Python has the capability built in to the standard library read and parse various data formats. The standard library has modules such as *xml*, *json*, and *HTMLParser*. There is also a wide variety of 3rd party modules that allow for parsing of data as well. For example the built in *HTMLParser* module does not deal well with malformed HTML. This is why it's necessary to rely on 3rd party module such as lxml[3] for parsing HTML. Not only is *lxml* a great parser for XML and HTML it's also highly efficient. The lxml module has performance advantages over other parsers such as *BeautifulSoup*, *html5lib*, as well as the built in HTMLParser. Ian Bicking did a comparison[4] of Python parser performance for his PyCon talk in 2008 which shows *lxml* coming out on top in most tests.

Even though the built-in xml module is sufficient for parsing XML you might want to consider using *lxml* to parse your XML data as well. It's a good idea whenever encountering HTML, XHTML, and XML to use lxml. Many people talk about *BeautifulSoup* to parse HTML and XML. The maintainer of *BeautifulSoup* has expressed interest in not continuing the project and recommends that people use *lxml* instead. Recent changes to the Python language has made *BeautifulSoup* a bit less tolerant than it was in the past.

Weaponize Your Tests

Python provides all of the modules necessary to communicate with modern web technologies so what is left is weaponizing your test cases in order to create accurate tests for vulnerabilities. Many tests for vulerabilities can be run by just sending GETs and POSTs.

pywebfuzz

One of our goals is to make the testing of web applications easier from the Python programming language. We want the logic to make different requests as

[1] http://nullege.com/
[2] http://code.google.com/p/httplib2/
[3] http://codespeak.net/lxml/
[4] http://blog.ianbicking.org/2008/03/30/python-html-parser-performance/

well as test values available directly in the language. That’s why we are introducing the pywebfuzz[5] project.

The *pywebfuzz* project provides values for testing as well as logic for requests and range generation. When it comes to providing values for testing, *pywebfuzz* implements values from the fuzzdb[6] project cleaned up and available through Python classes as list objects.

The following example would create a Python list that contains all of the values from *fuzzdb* for LDAP Injection.

```
from pywebfuzz import fuzzdb

ldap_values = fuzzdb.attack_payloads.ldap.ldap_injection
```

Now the ldap_values variable would be a Python dictionary containing the values from fuzzdb’s ldap_injection file. You could then iterate over the top of this variable with your tests.

Even though it’s early in the projects life you can perform some basic requests with the module as well. The *pywebfuzz* has a make_request function that allows you to make requests for content. The following would create a GET request for python.org

---

```
from pywebfuzz import utils

location = "http://python.org"

headers, content = utils.make_request(location)
```

The module also has an encoder library that can aid in the encoding and decoding of values in various formats. The following is an example of URL encoding value.

```
from pywebfuzz import EncoderLib

urlencoded = EncoderLib.url_encode(value)
```

---

[5] http://code.google.com/p/pywebfuzz/
[6] http://code.google.com/p/fuzzdb/

Browser Integration

Python has integration in a couple of browser frameworks including Mozilla Firefox and Webkit. Having integration with the browser can be an important part of testing and give greater capabilities to the code that you write. Some of these capabilities can be rendering of content, inspection of the DOM, as well as integration with plug-ins.

Firefox

It's possible to integrate with the Firefox web browser using the Python programming language. Python can be used to write plug-ins for Firefox and even standalone XULRunner applications. The easiest way to get up and running with Python integrated is to use the Python Extension pyxpcomext[7]. This extension allows for the use of cross platform communication components to integrate the Mozilla components with your Python code.

You can also use XULRunner to create a standalone application that uses Python logic. The following is a good tutorial on how to get up and running creating standalone XULRunner applications with Python[8]. Even though there is a good amount of integration with Python there are still some items that require the use of JavaScript in order to interact with it.

Webkit

Python also has integration in the Webkit browser. This can come from GUI frameworks such as PyGTK or PyQT. Even simple tasks like rendering responses returned from other libraries can be done using a Webview. The following is an example of making a request with the *httplib2* module for python.org and rendering it in a webview.

```
from PyQt4.QtGui import *
from PyQt4.QtWebKit import *
import httplib2

http = httplib2.Http()
headers, content = http.request("http://python.org", "GET")

app = QApplication(sys.argv)

web = QWebView()
web.setHtml(content)
web.show()

sys.exit(app.exec_())
```

[7] http://pyxpcomext.mozdev.org/
[8] http://pyxpcomext.mozdev.org/no_wrap/tutorials/pyxulrunner/python_xulrunner_about.html

The Webkit implementation inside of PyQT also has some interesting features either currently or slated for the future. For example you can open an inspector on the content being rendered and have integration of browser plug-ins such as Flash and Silverlight as well.

Working with Binary Protocols

Throughout your testing experience, you will most likely run into more than a few applications that utilize a custom or proprietary protocol in one or more of its components. Often, the application architects decide on such formats for simplicity, though most of all for their efficiency. Binary protocols are often compact, and less verbose than their HTTP or even SOAP/XML equivalents. These protocols generally adhere to a strict protocol specification, so while flipping random bytes may shake out some potential vulnerabilities in the underlying parser, you will not fully explore the capabilities of the applications' components, resulting in inadequate test coverage.

Adobe Flex Application Testing and PyAMF

One such binary protocol is Action Message Format[9] (AMF), commonly used in Adobe Flex applications. Luckily, most of the heavy lifting writing a protocol parser has been done for you and is available in the popular PyAMF[10] module. PyAMF, in addition to providing encoders and decoders, exposes a number of API for those wishing to write their own clients and gateways for a variety of existing Python frameworks.

The following example illustrates a simple AMF client that calls a method "getLanguages" on the "service" destination available on a remote endpoint hosted at http://demo.pyamf.org/gateway/recordset/amf.

```
from pyamf.remoting.client import RemotingService

client = RemotingService("http://demo.pyamf.org/gateway/recordset")
service = client.getService("service")

print service.getLanguages()
```

PyAMF supports serialization of Python data types to most AMF data types. For example, if a remote method expects a parameter of *java.util.Date*, PyAMF will accept and serialize a Python datetime.datetime() object to AMF.

[9] http://opensource.adobe.com/wiki/download/attachments/1114283/amf3_spec_05_05_08.pdf
[10] http://pyamf.org/

Similar to Java RMI[11], Flex provides the ability for developers to pass objects from client to server and vice-versa. In this scenario, the client binds an ActionScript Value Object with a server-side Java object. It is easy to identify methods expecting custom objects, as the server will complain about invalid types – something along the lines of:

```
"Cannot convert type java.lang.String with value 'marcin' to an instance of
class flex.samples.crm.employee.Employee"
```

In such a scenario, the client is binding an ActionScript Value Object with a server-side “flex.samples.crm.employee.Employee” object. We can create such an object using a simple object factory shown below:

```
import pyamf

class Factory(object):
    def __init__(self, *args, **kwargs):
        self.__dict__.update(kwargs)

pyamf.register_class(Factory, "flex.samples.crm.employee.Employee")

employee = Factory(**{'firstName': 'Marcin'})
```

We also called the pyamf.register_class() method to register the Factory object with a class alias. Whenever PyAMF encounters an instance of the Factory object, it will look up its alias, (e.g., flex.samples.crm.employee.Employee), and serialize it as such. This capability allows for us to successfully send our “employee” object as the appropriate object type the server expects. You can inspect the contents of the class cache by calling pyamf.CLASS_CACHE:

```
>>> pyamf.CLASS_CACHE
{<class 'pyamf.ASObject'>: <ClassAlias alias= class=<class 'pyamf.ASObject'>
@ 0xd5de50>, <class '__main__.Factory'>: <ClassAlias
alias=flex.samples.crm.employee.Employee class=<class '__main__.Factory'>
@ 0x7f778447ef10>, 'flex.samples.crm.employee.Employee': <ClassAlias
alias=flex.samples.crm.employee.Employee class=<class '__main__.Factory'>
@ 0x7f778447ef10>}
```

Custom Protocols

While PyAMF above was included as an example you may commonly run into, you might not end up so lucky and will have to resort to writing a client on the fly. It is in this scenario Python shines as language and you find writing a client to be

[11] Java Remote Method Invocation, http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

both simple and time saving. Best of all, the underlying components can be reused, especially in cases where you find yourself testing different protocols that serialize to a common format.

When working with custom protocols, the preferred modules to work with are StringIO and struct. StringIO (and its faster cousin, cStringIO), provides a file-like object interface to a string buffer. This allows us to write() to and read() from string buffers. Using struct, we can convert Python values to C structs. This is useful when a particular data type such as a Python Integer, needs to be encoded to an 8-byte IEEE-754[12] double precision floating point in network byte-order.

Implementing an example protocol

Take for example the following protocol specified below:

```
U8          = unsigned 8-byte integer
U16         = unsigned 16-byte integer
UTF-8       = U16 * (UTF8-char) ; as defined in RFC3629
DOUBLE      = 8-byte IEEE-754 double precision
            ; floating point in network byte order

msg                 = message-count parameters
message-count       = U16
parameters          = number-type | boolean-type | string-type
number-marker       = 0x00
boolean-marker      = 0x01
string-marker       = 0x02
number-type         = number-marker DOUBLE
boolean-type        = boolean-marker U8
string-type         = string-marker UTF-8
```

In this particular protocol, we construct a message by a) specifying the number of parameters in the message followed by b) its parameters. Each parameter can be a number, a Boolean or a string. Each parameter must be prefixed with its appropriate type marker. Strings must be encoded in UTF8-char as defined in RFC3629[13] and must be prefixed by an unsigned 16-byte integer designating the length of the string. Numbers must be 8-byte IEEE-754 double precision floating points in network byte-order.

Let's take the following parameters and encode it according to the protocol specification:

```
param1 = "sing"
param2 = "I've got ? problems but your app ain't one."
param3 = 99
```

[12] http://en.wikipedia.org/wiki/IEEE_754-1985
[13] http://www.ietf.org/rfc/rfc3629.txt

```
param4 = True
```

According to the protocol specification, the message must first begin with a message-count, a 16-byte unsigned integer that specifies how many parameters are in our message. Since a 16-byte unsigned Integer is also known as an “unsigned short”, we’ll use the unsigned short format specifier, “H” when calling struct.pack().

```
>>> buf = StringIO.StringIO()
>>> buf.write(struct.pack("H", 4))
```

After our message-count, we can start adding our parameters. Recall a string must be specified with a string-marker, 0x02, followed by the length of the string, and then the string in UTF8-char format.

```
>>> buf.write("\x02")     # string-marker
>>> buf.write(struct.pack("H%ds" % len(param1),
                          len(param1), param1))
```

Our buffered string now contains the following value:

```
>>> buf.getvalue()
"\x04\x00\x02\x04\x00sing"
```

Next, we encode param2, another string, the same way:

```
>>> buf.write("\x02")
>>> buf.write(struct.pack("H%ds" % len(param2),
                          len(param2), param2))
```

Afterwards, our buffer string looks like:

```
>>> buf.getvalue()
"\x04\x00\x02\x04\x00sing\x02+\x00I've got ? problems but your app ain't one."
```

For param3, we must encode it as a IEEE-754 Double in network byte-order. To write a Double, we use “!d” format specifier to struct.pack. The exclamation mark specifies that the format be encoded in network-byte order.

```
>>> buf.write("\x00")
>>> buf.write(struct.pack("!d", 99))
```

Our encoded message now looks like:

```
>>> buf.getvalue()
"\x04\x00\x02\x04\x00sing\x02+\x00I've got ? problems but your app ain't
one.\x00@X\xc0\x00\x00\x00\x00\x00"
```

We're not done just yet; we still need to add our last parameter, a Boolean to the message:

```
>>> buf.write("\x02")
>>> buf.write(struct.pack("?", True))
```

After all is said and done, our encoded message looks like so:

```
>>> buf.getvalue()
"\x04\x00\x02\x04\x00sing\x02+\x00I've got ? problems but your app ain't
one.\x00@X\xc0\x00\x00\x00\x00\x00\x01\x01"
```

Conclusion

Python is a great language for creating cases for testing modern web applications. There are libraries to address every major web technology even items such as AMF and HTML5. When encountering a difficult testing situation being able to write your own code to properly test the application may be the difference between success and failure. Python is the perfect language to fill the void left where modern testing tools are ineffective.