16:12 This PDF is a Shell Script That Runs a Python Webserver That Serves a Scala-Based JavaScript Compiler With an HTML5 Hex Viewer; or, Reverse Engineer Your Own Damn Polyglot

by Evan Sultanik

This PDF starts a web server that displays an annotated hex view of itself, ripe with the potential for reverse enginerding.

```
$ sh pocorgtfo16.pdf 8080
Listening on port 8080...
```

Warning: Spoilers ahead! Stop reading now if you want the challenge of reverse engineering this polyglot on your own!

The General Method

First, let's talk about the overall method by which this polyglot was accomplished, since it's slightly different than that which we used for the Ruby webserver polyglot in PoC‖GTFO 11:9. After that I'll give some further spoilers on the additional obfuscations used to make reversing this polyglot a bit more challenging.

The file starts with the following shell wizardry:

```
!  read -d '' String <<"PYTHONSTART"
```

This uses *here document* syntax to slurp up all of the bytes after this line until it encounters the string "`PYTHONSTART`" again. This is piped into `read` as stdin, and promptly ignored. This gives us a place to insert the PDF header in such a way that it does not interfere with the shell script.

Inside of the here document goes the PDF header and the start of a PDF stream object that will contain the Python webserver script. This is our standard technique for embedding arbitrary bytes into a PDF and has been detailed numerous times in previous issues. Python is bootstrapped by storing its code in yet another here document, which is passed to `python`'s stdin and run via Python's `exec` command.

```
!  read -d '' String <<"PYTHONSTART"
%PDF-1.5
%0x25D0D4C5D8
9999 0 obj
<</Length # bytes in the stream
>>
stream
PYTHONSTART
python -c 'import sys;
exec sys.stdin.read()' $0 $* <<"ENDPYTHON"
Python webserver code
ENDPYTHON
exit $?
endstream
endobj
Remainder of the PDF
```

Obfuscations

In actuality, we added a second PDF object stream *before* the one discussed above. This contains some padding bytes followed by 16 KiB of MD5 collisions that are used to encode the MD5 hash of the PDF (*cf.* 14:12). The padding bytes are to ensure that the collision occurs at a byte offset that is a multiple of 64.

Next, the "Python webserver code" is actually base64 encoded. That means the only Python code you'll see if you open the PDF in a hex viewer is `exec sys.stdin.read().decode("base64")`.

The first thing that the webserver does is read itself, find the first PDF stream object containing its MD5 quine, decode the MD5 hash, and compare that to its actual MD5 hash. If they don't match, then the web server fails to run. In other words, if you try and modify the PDF at all, the webserver will fail to run unless you also update the MD5 quine. (Or if you remove the MD5 check in the webserver script.)

From where does the script serve its files? HTML, CSS, JavaScript, ... they need to be *somewhere*. But where are they?

The observant reader might notice that there is a particular file, "`PoC.pdf`",[38] that was purposefully omitted from the feelies index. It sure is curious that that PDF—whose vector drawing should be no more than a few hundred KiB—is in fact 6.5 MiB! Sure enough, that PDF is an encrypted ZIP polyglot!

The ZIP password is hard-coded in the Python script; the first three characters are encoded using the symbolic regression trick from 16:09 (*q.v.* page 47), and the remaining characters in the password are encoded using Python reflection obfuscation that simply amounts to a ROT13 cipher. In summary, the web server extracts itself in-memory, and then decrypts and extracts the encrypted ZIP.

[38] Here, "PoC" stands for "Pictures of Cats", because the PDF contains a picture of Micah Elizabeth Scott's cat Tuco.