

# 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 engineering.

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



The screenshot shows a web browser window with the address bar containing `http://localhost:8080/`. The main content area displays the following text:

### PoC||GTFO Issue 0x16

In Which a PDF is a Shell Script that Runs a Python Webserver Serving a Scala-Based JavaScript Compiler with an HTML5 Hex Viewer that Can Help You Reverse Engineer Itself

Neighbor, as you read this, your web browser is downloading the dozens of megabytes constituting `pocorgtfo16.pdf`. From itself. Depending on your endowment of RAM, you may notice your operating system start to resist. Please be patient, as this may take a couple minutes to load.

The hex viewer used for this polyglot is Kaitai Struct's WebIDE, which is freely available under the GPL v3. The only modifications we made to it were to display this dialog and to auto-load `pocorgtfo16.pdf`. All of the modified source code is available in the feelies.

Despite where you may stand in The Great Editor Schism, Pastor Manul Laphroaig urges you to put aside your theological differences and celebrate this great licensing achievement of Saint IGNUcius—which is not so much different than our own самиздат license—, without which this polyglot would have likely been impossible. Sanctity can be found in all manner of hackery. In any event, we hear that the good Saint runs Vim from inside of Emacs, which is not so much different than our own polyglots.

This is a fully functional hex viewer and reverse engineering tool, with which you can load any other file from your filesystem. We have annotated the PDF using Kaitai Struct, which should be sufficient for you to figure it all out. You might even be tempted to edit the PDF to make your own PoC, but be careful! We've included some tricks to make modifications more of a challenge for you. But most importantly: Have fun!

Close

**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 *some-where*. But where are they?

The observant reader might notice that there is a particular file, "PoC.pdf",<sup>38</sup> 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.

---

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